Änderungspropagation für autonome und heterogene Informationssysteme

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der
Universität Stuttgart zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Uwe Heinkel

aus Hannover

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr.-Ing. Norbert Ritter
Tag der mündlichen Prüfung:	07.03.2011

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2011

Vorwort

Ganz besonders möchte ich mich bei meinem Doktorvater Prof. Dr. Mitschang bedanken: Ich habe viel gelernt von Ihnen, als ich an der Universität Stuttgart am Institut für Parallele und Verteilte Systeme (IPVS) beschäftigt war. Vielen Dank für unsere Diskussionen über die Forschung zu meiner Doktorarbeit. Ihre Tipps werden mich auch im weiteren beruflichen Leben voranbringen.

Weiterhin möchte ich meinem Mitberichter Prof. Dr. Ritter dafür danken, dass er sich Zeit nahm, um meine Dissertation zu lesen und zu evaluieren.

Was wäre die schöne Zeit an der Universität Stuttgart ohne meine Kollegen gewesen. Ich möchte mich deshalb bei allen Angestellten des IPVS bedanken. Besonderer Dank gilt Dr. Carmen Constantinescu, Clemens Dorda, Fabian Kaiser, Sylvia Radeschütz und Dr. Ralf Rantzau. Ebenfalls möchte ich meinen Dank an meine Kollegen im Transferbereich 059 und Sonderforschungsbereich 467 aussprechen, insbesondere an Olga Kernbach, Dr. Ralf Kapp, Ralph Winkler, Benno Löffler, Jan le Blond und Lars Illenberger.

Für die große Unterstützung der Abteilung Infrastruktur am IPVS bedanke ich mich ebenfalls herzlich. Besonders hervorheben möchte ich Ralf Aumüller, Heike Kniehl, Manfred Rasch und Christine Reissner.

Großen Dank für die große Unterstützung möchte ich an meine Familie aussprechen, insbesondere an meine Eltern Dieter und Regina, an meine Schwestern Sonja und Bettina sowie an meine Oma Erna.

Uwe Heinkel
Stuttgart, den 07.03.2011

Inhaltsverzeichnis

Abkürzungsverzeichnis 9

Zusammenfassung in deutscher Sprache 11

Zusammenfassung in englischer Sprache (Abstract) 13

1 Einleitung 21
- 1.1 Problemstellung und Motivation . . . 21
- 1.2 Zielsetzung . . . 24
- 1.3 Gliederung . . . 25

2 Grundlagen 27
- 2.1 Informationssysteme . . . 27
  - 2.1.1 Definition . . . 27
  - 2.1.2 Architektur . . . 29
  - 2.1.3 Modelle und Geschäftsobjekte . . . 30
- 2.2 Enterprise-Resource-Planning-Systeme . . . 31
- 2.3 Enterprise Application Integration . . . 33
  - 2.3.1 Unternehmenssicht . . . 34
  - 2.3.2 Klassifikationen . . . 34
  - 2.3.3 Technologien . . . 37
- 2.4 Datenintegration im Unternehmen . . . 37
  - 2.4.1 Globales Schema und homogene Systemlandschaft . . . 37
  - 2.4.2 Föderierte Datenbanken . . . 39
  - 2.4.3 Lokale Modelle, Geschäftsprozesse und einheitliche Benutzerschnittstelle . . . 41
- 2.5 Peer-Data-Management . . . 41
- 2.6 Workflows . . . 42
  - 2.6.1 Grundlagen . . . 42

2.6.2 Workflow-Managementsysteme (WFMS) 43
2.6.3 Workflow-Beschreibungen 43
2.6.4 Datenintegration mit Workflows 45
2.7 XML Technologien 46
2.8 Message Oriented Middleware 47
2.9 Ereignissysteme 49
2.10 Model-Management 50
2.10.1 Übersicht 50
2.10.2 Automatic Schema Matching 51
2.11 Schlussfolgerungen 51

3 Grundlegende Konzeption 55
3.1 Lösung für Replikation der Informationssystemdaten 55
3.2 Basiskonzepte 58
3.2.1 Abhängigkeiten und Propagationsprozesse 58
3.2.2 Änderungsbeschreibung 60
3.3 Transaktionen 61
3.4 XML als Basis für Änderungspropagation 62
3.4.1 XML zur Definition von Zustandsbeschreibungen 62
3.4.2 Technologie für eine XML-basierte Änderungspropagation . . . 63
3.5 Sprache für die Definition von Abhängigkeiten 64
3.5.1 Deklaration der Eingabe 66
3.5.2 Kontrollfluss 66
3.5.3 Verarbeitungs- und Output-Befehle 69
3.6 Pfadausdrücke für Änderungsbeschreibungen 72
3.6.1 Propagation Condition Language (PCL) 72
3.6.2 XPath-Bibliothek 74
3.7 Komponenten 75
3.7.1 Repository 76
3.7.2 Propagationsmanager 81
3.7.3 Abhängigkeitsmanager 87
3.8 Konflikterkennung und Auflösung 90
3.9 Reihenfolgeeinhaltung von propagierten Änderungsbeschreibungen . . . 93
3.10 Fehlerbehandlung 95
3.10.1 Fehlerklassifikation 95
3.10.2 Fehlerbehandlung im Prozessmanager 95
3.10.3 Fehlerbehandlung eines Propagationsprozesses 96
3.11 Adapter 96
3.11.1 Genereller Adapter 96
3.11.2 Adapter für relationale Datenbanken 98
3.12 Zusammenfassung 101

4 Komplexe Propagation 103
4.1 Einbindung von Daten aus Drittsystemen 104
4.1.1 Problemstellung 104
4.1.2 Verwendung eines Datendienstes 105
4.1.3 Zugriffsarten 107
4.1.4 Von der Definition zur Nutzung eines Datendienstes 107
4.1.5 Dienstbeschreibung 109
4.1.6 Realisierung 111
4.1.7 Beispiel 115
4.2 Verarbeitung mehrerer Änderungen 115
4.2.1 Problemstellung 116
4.2.2 Implementierungskonzept der M-zu-N-Erweiterung 117
4.2.3 Erweiterung von XPDL 122
4.2.4 Erweiterung von PCL 123
4.2.5 Schlussfolgerungen 123
4.3 Verteilte Propagation 124
4.3.1 Problemstellung 125
4.3.2 Einschränkungen der Lastverteilung 126
4.3.3 Load-Manager-Ansatz 127
4.3.4 Selbstorganisierter Ansatz 130
4.4 Zusammenfassung 131

5 Evaluation des Propagationssystems 133
5.1 Praxistest 133
5.1.1 Integrationsszenario 133
5.1.2 Digitale Fabrik und ihre Werkzeuge 134
5.1.3 Integrationsplattform 135
5.1.4 Integration der Digitalen Fabrik und des Planungstisches 137
5.1.5 Integration der Digitalen-Fabrik und des Montage-Konfigurators 139
5.1.6 Schlussfolgerungen 139
5.2 Evaluierung der Performance 140
5.2.1 Messmethodik 140
5.2.2 Testumgebung 141
5.2.3 Realisierung der zuverlässigen Multicast-Warteschlange 141
5.2.4 Testfälle 142
5.2.5 Zusammenfassung 156
5.2.6 Vergleich mit Anforderungen aus der Industrie 156
5.3 Vergleich mit EAI-Produkten 157
5.3.1 BizTalk 158
5.3.2 Oracle SOA Suite 159
5.3.3 Websphere Message Broker 159
5.3.4 Schlussfolgerung 160

6 Schlussfolgerung und Ausblick **161**
6.1 Schlussfolgerungen 161
6.2 Ausblick 163

Literaturverzeichnis **165**

Abkürzungsverzeichnis

API	Application Programming Interface
EAI	Enterprise Application Integration
ERP	Enterprise Resource Planning
JMS	Java Message Service
MOM	Message-oriented Middleware
PCL	Propagation Condition Language
RPC	Remote Procedure Call
XML	Extensible Markup Language
XPDL	XML Propagation Definition Language
XPath	XML Path Language
XSLT	Extensible Stylesheet Language Transformation

Zusammenfassung in deutscher Sprache

Heutzutage müssen Unternehmen sich schnell an neue Situationen anpassen. Die Gründe hierfür sind vielfältig: Kundenanforderungen ändern sich, Konkurrenten entwickeln neue Produkte bzw. Strategien oder neue Gesetze werden verabschiedet. Die Anpassungsfähigkeit von Unternehmen wird als Wandlungsfähigkeit bezeichnet. Damit Unternehmen diese Wandlungsfähigkeit erreichen können, müssen sie aus Einheiten bestehen, die weitestgehend autonom sind. Durch die Autonomie wird erreicht, dass Entscheidungen schnell getroffen werden können, weil jede Einheit selbstständig reagieren kann. Die Unternehmenseinheiten wurden im Sonderforschungsbereich 467 „Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienproduktion“, in dessen Rahmen auch diese Arbeit entstand, Leistungseinheiten genannt. Leistungseinheiten brauchen unter anderem eine Unterstützung durch Informationssysteme, welche Informationen bereitstellen und verwalten. Damit sich die Leistungseinheiten an neue Situationen anpassen können, müssen auch deren Informationssysteme so weit wie möglich autonom bleiben. Dennoch muss der Austausch von Daten zwischen den Informationssystemen garantiert sein, da Daten teilweise von vielen verschiedenen Leistungseinheiten und ihren Informationssystemen verwendet werden. Besonders deutlich wird das bei Kundendaten, die oftmals in vielen Unternehmensbereichen bzw. Informationssystemen benötigt werden. Daten, die von mehreren Informationssystemen benötigt und gespeichert werden, liegen oft redundant im Unternehmen und meist in heterogener Form vor. Werden redundante Daten in einem Informationssystem geändert, entsteht ein inkonsistenter Zustand, da an anderer Stelle noch die alten Daten gespeichert sind. Um diese Inkonsistenz zu verhindern, müssen die Informationssysteme integriert und die redundanten Daten synchronisiert werden. Replizierte Datenbanken haben ein ähnliches Problem: es müssen ebenfalls Daten synchronisiert werden. Hier sind die Daten aber meistens homogen und die partizipierenden DBMS sind nicht autonom. Des Weiteren ändern replizierte Datenbanken ihre Daten nur über ihre bereitgestellte Schnittstelle in der Datenschicht, in einem Informationssystem sollten sie hingegen in der Anwendungsschicht geändert werden, weil dort die Anwendungslogik liegt und oftmals wichtige Konsistenzregeln geprüft werden müssen.

Um diesen Anforderungen gerecht zu werden, wurde in dieser Arbeit ein XML-basiertes Datenintegrationssystem konzipiert und entwickelt, das Änderungspropagation verwendet, um redundante Daten von Geschäftsobjekten zu synchronisieren. Ein Geschäftsobjekt besteht aus einem oder mehreren Implementierungsobjekten, beispielsweise hat ein Kundenauftrag einen Auftragskopf und mehrere Auftragspositionen. Aufgetretene Änderungen werden in einer sogenannten Änderungsbeschreibung pro Geschäftsobjektänderung propagiert, die alle wichtigen Daten einer Änderung enthält. Besonders wichtig sind die zwei Zustände von Geschäftsobjekten, vor und nach der Änderung, und die Änderungsart (*create*, *update*, *delete*) des Geschäftsobjektes. Die Verwendung von zwei Zuständen ermöglicht die Erkennung der Änderungsarten bei den Implementierungsobjekten sowie die Ermittlung von Änderungsdeltas innerhalb des Integrationssystems. Änderungsbeschreibungen werden entlang von definierten Abhängigkeiten propagiert, die von einem Quellsystem zu mehreren Zielsystemen gehen. Um diese Abhängigkeiten flexibel gestalten zu können, wurde eine XML-basierte Sprache entwickelt, die den Namen *XML Propagation Definition Language* (XPDL) trägt. Des Weiteren wurde eine XPath-basierte Sprache (*Propagation Condition Language*, PCL) entworfen, die zustandsübergreifende Bedingungen ermöglicht, um Filter für Abhängigkeiten zu definieren. Besonders wichtige Eigenschaften eines Datenintegrationssystems sind die Einhaltung der Änderungsreihenfolge und die Erkennung von Änderungskonflikten. Beide Punkte wurden in dieser Arbeit umgesetzt. Für die Erkennung von Änderungskonflikten wurde eine zustandsbasierte Methode entwickelt, die eine feingranulare Erkennung von Änderungskonflikten ermöglicht. XPDL und PCL ermöglichen eine weitgehend abstrakte Beschreibung von Änderungspropagationen. Damit können dann recht unterschiedliche Informationssysteme unterstützt werden und auch Drittsysteme, die zusätzliche Daten bereitstellen, eingebunden werden.

Zusammenfassung in englischer Sprache (Abstract)

In order to stay competitive enterprises need to adapt themselves constantly to new situations like new products in the market, changes of the competitors' strategies, new laws, changing of customers' needs, and so on. For coping with these turbulent situations, the enterprise has to have a fast decision making, which is facilitated by autonomous organization units inside the enterprise. In the project SFB 467 with the title "Transformable Business Structures for Multiple-Variant Series Production", a kind of organization units was developed that are called transformable business units. These business units are characterized by processes that have products and/or information as input and output; these processes have a customer value as well. The customers are internal (other business units) or external customers (enterprise customers). On the other hand, the business units need to act in concert and achieve common enterprise goals. Therefore, the business units must be integrated among each other, which affects also the integration of their information systems, since they are part of the information exchange between the units.

Due to the fact that data of these information systems needs to be integrated and locally stored, some data is redundant in the enterprise. Changes of the redundant data lead involuntarily to inconsistent data inside the enterprise and problems arise in handling enterprise wide processes. This inconsistency can be anticipated by a flexible data integration that copes with heterogeneity. Heterogeneity represents a big topic for integrations due to the fact that IT infrastructures are comprised of many different products and individual software.

The integration of replicated data can be compared with the integration of replicated databases, since the replicated databases store redundant data as well. On the other hand, the redundant data in replicated databases exists in a homogenous form and the update propagation - used to update redundant data - changes the data via the data layer. Information systems consist usually of multiple layers, e.g. the data layer, the application layer, and user interface layer. Above the data layer is the application layer that realizes application logic, and thus consistency rules may also be realized in this layer. A data integration of independent information systems should use the

application layer interface and not that of the data layer, so that all the consistency rules will be considered. Moreover, the application logic can trigger processes, which will not be started if changes are done directly in the data.

Furthermore, there exist a lot of EAI-Products to integrate information systems inside enterprises that apply data in the application layer. Nonetheless, they lack of order control of changes handled by the integration system and the detection of change conflicts. These EAI products are mostly made to integrate and implement business processes, and thus not for the pure data integration without the involvement of business processes. Furthermore, with the help of such EAI-Products so-called integration applications are realized, which have their own communication channels. On the other hand, a pure data integration system should have a central channel independent of the existing integration applications. This channel provides the involved information systems with a transparency of the realized integration applications. Hence such applications can easily be removed, added or changed without the adaption of the information system resulting in a more flexible way of data integration in an enterprise.

This thesis was part of the mentioned project SFB467. It proposes a data integration system called change propagation system that overcomes these shortages and is especially developed for integrating data of autonomous and heterogeneous information systems. The data integration is based on forwarding changes of business objects. A business object represents an object with a value to the enterprise that can consist of multiple implementation objects, e.g. a customer order with order header and order positions as implementation objects. For representing states of business objects XML is adequate, since it has a hierarchical structure, which facilitates the inclusion of multiple implementation objects inside a document, and thus complete business objects. Furthermore, the combination of meta data and data provides a human readable form of the business objects states. Not only the business object states are needed as information for propagation; all needed information is subsumed under the term change description that represents a propagated object. These change descriptions are propagated along dependencies from one system (source system) to multiple systems (destination systems). For the flexible constitution of dependencies an XML based language was developed, which is called XML Propagation Definition Language (XPDL). Furthermore, for the definition of conditions between the two states an XPath-based language was developed, called Propagation Condition Language (PCL). A requirement for such a change propagation system is the guaranteed transmission of change descriptions from the source system to destination systems. Thus persistent message queues are used. Moreover, these queues facilitate a FIFO communication.

Important features of such a propagation system are the consideration of the correct order of change descriptions and the detection and resolution of change conflicts. For the conflict detection we propose a method based on the business object states, instead of using timestamps, which will be later explained in more detail. Due to the heterogeneity we support transformations and integration of data from third systems. Therefore we introduce data services - services offered by third systems - that provide additional data for the destination system. These data services can be queried by a Remote Procedure Call (RPC), SQL or XQuery, depending on the interface and stored

data.

The remainder of the abstract is organized as follows. In the first section we introduce the change description. The second section gives an overview of the XML Propagation Definition Language, which is followed by the Propagation Condition Language. Afterwards the architecture of the Change Propagation System is sketched. Finally, in the last two sections the conflict detection and data services are introduced.

CHANGE DESCRIPTIONS

First, there is a need for a complex object that contains all the necessary information, which is needed by a destination system to adapt its data correspondently. This complex object is called change description and is defined by the following tuple:

$$CD = (S, BOT, CT, B, A, TS)$$

S system where the change has occurred

BOT type of business object (e.g. Customer Order)

CT type of change (create, update or delete)

B state of the business object before the change

A state of the business object after the change

TS timestamp, when the change has occurred.

The usage of two states instead of only employing the after state facilitates the detection of change types CT of implementation objects inside a business object. The CT of an implementation object can differ from the CT of the business object. For example a customer order could have been updated, while positions have been added, updated, or deleted. Furthermore, change deltas can be calculated inside the change propagation system, e.g. the movement vector of a production resource after the change of its position in the factory layout. The two-state propagation facilitates the state-based conflict detection as well.

We use two states (B and A) instead of a combination of change delta with one state, since it makes the processing of the change descriptions easier. The main advantages of using two states are that only one XML schema for validation is needed, only one transformation script for adapting the states, and states can be easier described than change deltas.

XML Propagation Definition Language (XPDL)

Another important feature of a change propagation system is the definition of dependencies between business objects stored in different information systems. These

dependencies should be as flexible as possible. A language can offer a great deal for this purpose. We created a language called XML Propagation Definition Language (XPDL). The XPDL artifacts are called propagation scripts and the executed propagation scripts are named propagation processes. XPDL is a language that consists of following statements:

I. Input Declaration
The input declaration consists of an input statement that selects the source system (the system, where the change occurred) and the business object type.

II. Controll Flow

   a. Sequence
   The sequence statement allows the execution of subsequent statements in order how they are specified.

   b. Parallel
   The parallel statement allows the execution of subsequent statements concurrently.

   c. Condition
   The condition statement has two branches: the true and the false branch, which will be executed depending on the evaluation of a Boolean expression.

III. Regular Statements

   a. Transform
   The statement adapts the states (B and A) of a change description. Therefore it uses transformation scripts which can be either written in XSLT or XQuery. The transform statement is used transparently to the number of existing states in the change description; one transformation script is written for one state, resulting in a transformation script that exactly transforms one business object type.

   b. Propagate
   The propagate statement sends the change description to a destination system by using its queue.

We came to the conclusion that there is no need for an iteration control flow, since a business object should be handled as one business object and not separated into many implementation objects inside the propagation system. This task should be handled by an adapter of the information system. Furthermore, we don't include filter statements, since it is a better programming style to filter by condition statements. Condition statements use control flow for filtering and filter statements use data flow. The latter statement would make it difficult to analyze the behavior of the programmed propagation script, because it would not be clear which statements will be executed.

Propagation Condition Language (PCL)

Sometimes it is necessary to define conditions between the two states of a change description, e.g. we only want to propagate to a certain system, when the name of a person has changed. Therefore we developed a language called Propagation Condition Language, which is an extension of XPath. Another feature of this language is the evaluation of the change type in a change description, which facilitates conditional execution depending on the change type. The language consists of the following:

- before
  The statement returns the before state.

- after
  It returns the after state.

- beforeOrAfter
  It represents the after state in case the before state is null. Can be used when the before state can be replaced by the after state in case of none existence, e.g. the order volume.

- afterOrBefore
  The statement is similar to beforeOrAfter, but here the after state is preferred.

- chgType
  It provides the change type of the change.

- timestamp
  It returns the time when the change occurred. It facilitates the implementation of time based propagations.

The statements are enclosed by the percentage marks and can be combined with XPath expressions.

If we describe a condition for executing the succeeding statements based on the fact that the last name of a customer has changed, the PCL condition looks like the following:

```
%before%/Customer/Lastname != %after%/Customer/Lastname
```

Architecture

The change propagation system consists of three important components: the repository, the dependency manager, and the propagation manager. The repository stores all the meta data needed for propagation: system descriptions, XML Schemas for the validation of business object states contained in change descriptions, propagation scripts (XPDL artifacts) and transformation scripts required for the XPDL transform statement. The dependency manager is a graphical tool to enter this meta data into the

repository. The third component - the propagation manager - propagates change descriptions from source systems to destination systems by executing propagation scripts. It consists of several components: the queue manager is used to receive change descriptions from source systems and to send processed change descriptions to destination systems, the process manager for managing the propagation processes, the XPDL engine for the execution of XPDL statements, an XML parser for the translation of states into the internal format (DOM) and the validation of correctness due to a schema, as well as two transformation engines (XSLT and XQuery).

Conflict Detection

A conflict can occur then the same business object is changed in more than one system concurrently. The concurrent change of a business object needs to be detected, since it would lead to inconsistent data in the involved systems. Furthermore, the detected conflicts need to be evaluated and resolved, so that the information systems are consistent again.

As we already mentioned, we suggest to use states for detecting change conflicts. The state-based approach makes it unnecessary to adapt the data model of an information system for the storage of timestamps, which are needed for timestamp-based approaches. The biggest advantage of the approach is the flexible definition of zones inside the business object, which facilitates the possibility of concurrent changes in independent zones, e.g. the customer object could have independent zones with the address and the account information. We call this approach a fine granular detection approach based on states.

A schema annotated with zone information is illustrated in the following example:

```
<xs:schema ...>
    <xs:element name="Customer">
        <xs:complexType>
            <xs:sequence>
                <xs:element cd:id="true" name="ID"/>
                <xs:element cd:group="name" name="Firstname"/>
                <xs:element cd:group="name" name="Lastname"/>
                <xs:element cd:group="address" name="Address">
                </xs:element>
                   ...
                <xs:element cd:group="account" name="AccountInfo">
                   ...
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

We introduced two attributes, which can be associated with element types: id and group. Id defines the identifier of the business object to differentiate the business objects. This identifier is needed to retrieve the local state of the business object. The group attributes introduce independent zones inside the business object.

Data Services

The destination system may need data which is neither stored in the destination system itself nor in a source system. An example is the integration of a factory layout system with a digital factory solution. A factory layout system plans the positions of resources in the factory, while the digital factory solution is used for the complete planning process including production processes and resources. Thus, both systems should be integrated with each other. The factory layout system stores production resources just with a name and their positions (X, Y, and Z). In contrast, the digital factory solution needs many attributes for resources. A resource can be created in the layout system and then synchronized with the digital factory. Hence, it lacks of the additional attributes required by the digital factory solution. These attributes can be provided by a third system, e.g. a service of the resource manufacturer, which acts as data service.

The change propagation system needs to be extended to fulfill this purpose. First, there is a need for a language to describe data services which is called Data Service Definition Language (DSDL). The second extension is a new statement in XPDL for the call of a data service. The approach provides the possibility of integration of three types of data services: relational databases, XML stores, and RPC access. We made the approach extensible so that the system can be extended by further types. The only requirement is that the extension takes a set of parameters and provides an XML document as result. The existing types use parameters for procedure calls (RPC) or for parameterized queries (relational databases and XML stores). These parameters are bound during runtime in the propagation system using PCL expressions for extracting values out of the states included in a change description. Therefore, we extended the XPDL with a data service call statement, which reads the DSDL definition of the data service from the repository, binds the parameter, and sends via SOAP the call to the data service. The retrieved result is integrated into the change description via standard transformations.

KAPITEL 1

Einleitung

1.1 Problemstellung und Motivation

Heutzutage sind Unternehmen einem immer größer werdenden Wettbewerbsdruck ausgesetzt: Lieferzeiten werden kürzer, Kundenwünsche ändern sich, Konkurrenten verändern ihre Strategie und ihre Produktpaletten, oder es werden neue Rahmenbedingungen geschaffen (z.B. Auflagen oder Gesetze). Dies sind nur einige Beispiele, wie sich die Situation eines Unternehmens verändern kann. Diese Veränderungen erfordern von den Unternehmen, dass sie sich nicht nur in einem beschränkten flexiblen Rahmen bewegen, sondern sich auch neuen und unvorhersehbaren Situationen anpassen können. Sie sollten wandlungsfähig sein. Um diese Wandlungsfähigkeit zu ermöglichen, wurde im Sonderforschungsbereich 467 (SFB 467) „Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienfertigung", in dessen Umfeld auch diese Arbeit entstand, das Konzept der Leistungseinheiten (vgl. Abbildung 1.1) [WZ09, Wes06] entwickelt.

Leistungseinheiten sind organisatorische Einheiten in einem Unternehmen, die Produkte oder Informationen als Output haben (vgl. Abbildung 1.1). Leistungseinheiten werden über mehrdimensionale Ziele koordiniert, die in Verhandlungs- und Abstimmungsprozessen festgelegt werden. Die Mitarbeiter einer Leistungseinheit versuchen dann mit dem Einsatz von Ressourcen, diese Ziele zu erreichen. Die Hauptaufgabe einer solchen Leistungseinheit ist, Produkte durch Ausführungsprozesse zu erzeugen, die den Materialinput umwandeln. Handelt es sich um keine produzierende Leistungseinheit, sind Input und Output Informationen. Produzierende Leistungseinheiten stellen ebenfalls Informationen bereit, die von anderen Leistungseinheiten benötigt bzw. weiterverarbeitet werden. Diese Informationen können beispielsweise Kundeninformationen oder Stadien von Produktionsaufträgen sein. Sie werden oftmals durch Informationssysteme verwaltet. Um die Leistungseinheiten zu integrieren ist ein Informationsaustausch zwischen den einzelnen Informationssystemen notwendig, damit die Leistungseinheiten zusammenarbeiten und übergeordnete, gemeinsame Ziele erreichen können.

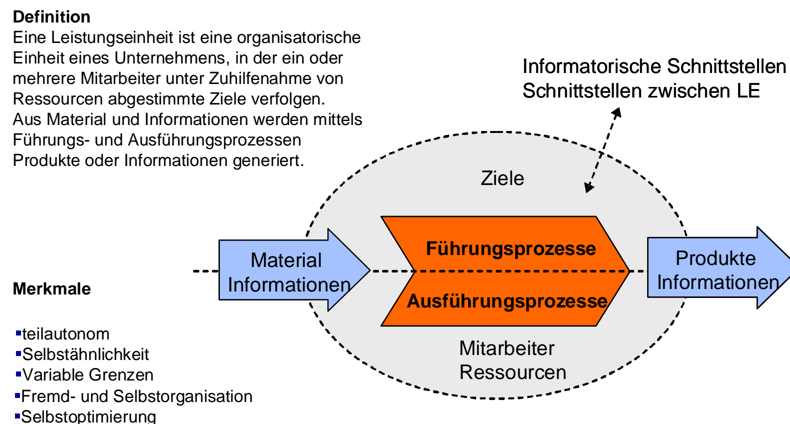


Abbildung 1.1: Leistungseinheiten als Grundbaustein eines wandlungsfähigen Unternehmens [WZ09, Wes06]

Die Leistungseinheiten sind hierarchisch organisiert, d.h. eine Leistungseinheit kann wiederum aus Leistungseinheiten bestehen (Selbstähnlichkeit). Durch die variablen Grenzen und Teilautonomie wird die Wandlungsfähigkeit möglich, da Entscheidungen unter Berücksichtigung der Zielvorgaben schnell gefällt werden können. Die variablen Grenzen ermöglichen eine flexible Gestaltung der Leistungseinheit anhand des Bedarfs, d.h. untergeordnete Leistungseinheiten können hinzugenommen oder abgegeben werden.

Die Teilautonomie wirkt sich auch auf die IT-Infrastruktur des Unternehmens aus. Die Informationssysteme, die Bestandteil der IT-Infrastruktur sind, sollten ebenfalls weitestgehend autonom sein, da sie sonst die geforderte Wandlungsfähigkeit der Leistungseinheiten behindern würden. Die Autonomie sagt dabei aus, dass die einzelnen Informationssysteme möglichst lose gekoppelt seien sollen. Außerdem werden durch die Autonomie der einzelnen Leistungseinheiten häufig Informationssysteme beschafft, die sich sehr von denen anderer Leistungseinheiten unterscheiden. Das bedeutet, es entsteht eine heterogene Infrastruktur im Unternehmen.

Die Heterogenität der Informationssysteme kann durch unterschiedliche Rechnerarchitekturen, Betriebssysteme und/oder Datenbanksysteme entstehen. Den Datenbanksystemen können unterschiedliche Datenmodelle zugrunde liegen, z.B. das relationale Datenmodell. Die Schemata der Datenbanken können unterschiedlich modelliert sein. Hierbei unterscheidet man zwischen struktureller und semantischer Heterogenität [Her03]. Zur strukturellen Heterogenität gehören unterschiedliche Datenmodelle, verschiedene Beziehungsmöglichkeiten zwischen den Daten (z.B. Generalisierung oder Assoziation) und die Möglichkeit der Modellierung von komplexen Objekten. Semantische Heterogenität ist begründet in unterschiedlichen Bezeichnungen, die beispielsweise durch Synonyme oder Abkürzungen hervorgerufen werden.

Eine im Unternehmen entstandene heterogene Systemlandschaft könnte beispielsweise wie in Abbildung 1.2 dargestellt aussehen. Sie enthält eine Menge von Informationssystemen, wie zum Beispiel ein Enterprise-Resource-Planning-System oder eine Lagerverwaltung. Diese müssen über ein Integrationssystem integriert werden, um den

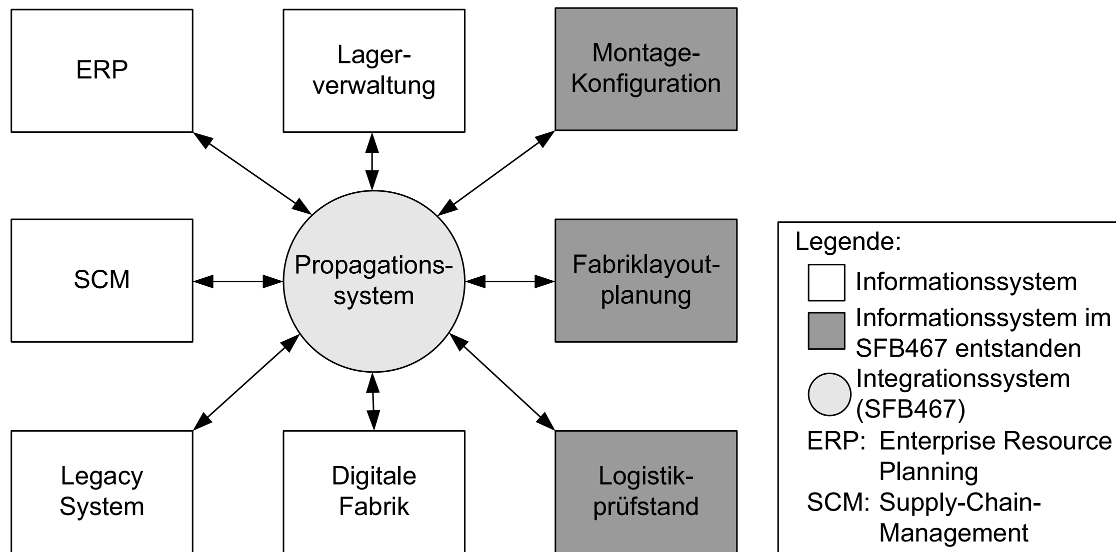


Abbildung 1.2: Informationssysteme in einem Unternehmen

angesprochenen Informationsaustausch zu realisieren. Ein mögliches Integrationssystem ist ein Propagationssystem, über den diese Arbeit handelt.

Die Integration von Informationssystemen bildet, wie in Abbildung 1.3 dargestellt, einen Graphen mit gerichteten und ungerichteten Kanten. Der Aufbau des Graphen ist stark davon abhängig, wie die Kommunikationsbeziehungen zwischen den einzelnen Leistungseinheiten und zwischen Informationssystemen innerhalb von Leistungseinheiten definiert sind. Der Informationsaustausch kann sowohl in eine Richtung gehen (gerichtete Kante), als auch bidirektional sein (ungerichtete Kante). Informationssysteme sind in der Regel in drei Schichten aufgebaut: einer Benutzerschnittstelle, einer Anwendungsschicht und einer Datenschicht. Benutzer können über die Benutzerschnittstelle Daten eingeben, pflegen und abfragen. Dafür verwendet die Benutzerschnittstelle Anwendungslogiken, die in der Anwendungsschicht realisiert sind. Die Daten werden schließlich in einer Datenschicht persistent vorgehalten. Diese Schicht kann zum Beispiel durch Datenbanksysteme realisiert werden.

Um einen Informationsaustausch zwischen den einzelnen Informationssystemen der Leistungseinheiten zu ermöglichen, müssen diese miteinander verbunden werden. Der hier angestrebte Ansatz ist ein gesteuerter Informationsaustausch, der bei Änderungen der Informationen eines Informationssystems diese an andere interessierte Informationssysteme mitteilt. Genauer gesagt handelt es sich um Änderungen von Daten, da Daten erst durch ihre Interpretation zu Informationen werden. Wenn man die einzelnen Informationssysteme betrachtet, so sind die Daten teilweise repliziert und zwar genau dort, wo ein Informationsaustausch notwendig ist. Das heißt, Objekte sind sowohl in dem einen System, als auch in anderen Systemen vorhanden. Diese Objekte müssen nicht gleich sein, sondern können unterschiedlich repräsentiert sein: die Objekte können sich anhand ihrer Struktur und Repräsentation der Daten unterscheiden. Beispielsweise kann ein System ausgeschriebene Ländernamen (z.B. Deutschland) und ein anderes System Ländercodes (z.B. D) verwenden. Dies ist bedingt durch die Heterogenität in

den Informationssystemen. Die replizierten Daten müssen für den angesprochenen Informationsaustausch integriert werden. Klassische Lösungen für replizierte Daten sind replizierte Datenbanken. Bei diesen werden gleiche Daten auf unterschiedlichen Rechnern vorgehalten. Dadurch kann eine höhere Performanz und Verfügbarkeit erreicht werden. Allerdings müssen die Daten bei Änderungen synchronisiert werden, sodass alle Replikate auf dem gleichen Stand sind. Dies erfolgt mittels Update-Propagation, welche die Konsistenz der Daten garantiert. Diese Systeme sind hauptsächlich für ein homogenes Umfeld gemacht und spielen Änderungen auf der Datenschicht von Informationssystemen ein. Dies hat zum Nachteil, dass Konsistenzregeln und evtl. vorhandene Anwendungslogiken umgangen werden. Stärken von solchen Systemen sind die Vermeidung oder Erkennung von Änderungskonflikten. Ob nun Konflikte nur erkannt oder vermieden werden, hängt zum Teil von der Art der Replikation (z.B. *Eager Replication* oder *Lazy Replication*) ab. Ein Änderungskonflikt tritt dann auf, wenn das gleiche reale Objekt in zwei unterschiedlichen Systemen gleichzeitig geändert wird.

Eine weitere Möglichkeit, Informationssysteme zu integrieren, stellen EAI-Produkte (*Enterpise Application Integration*) dar, die Informationen nachrichtenbasiert austauschen. Vorteile von solchen EAI-Produkten sind eine flexible Gestaltung von Integrationsprozessen und die Möglichkeit der Transformation von Nachrichten. Sie haben allerdings Schwächen in der Konfliktbehandlung und zumeist in der Einhaltung von der korrekten Reihenfolge von Änderungen.

Um diese Schwächen zu beheben, wird in dieser Arbeit eine Mischung aus beiden Ansätzen konzipiert, welche die Vorteile der beiden Welten vereinen soll. Dabei soll ein Update-Propagationssystem für ein heterogenes Umfeld entstehen, was die Synchronisation von geänderten Daten zwischen Informationssystemen ermöglicht. Dieser Ansatz soll dabei für die Integration von heterogenen und autonomen Informationssystemen maßgeschneidert sein. Die Änderungen sollen soweit wie möglich nicht in der Datenschicht eingespielt werden, sondern über Anwendungsschnittstellen in der Anwendungsschicht, damit Konsistenzregeln und Anwendungslogiken der Anwendungsschicht nicht umgangen werden. Wichtige Eigenschaften eines solchen Propagationssystems sind die oben beschriebene Erkennung oder Vermeidung von Änderungskonflikten. Werden diese nur erkannt, müssen Änderungskonflikte schließlich auch aufgelöst werden, d.h. es wird versucht die korrekten Daten wieder zu rekonstruieren. Da die Reihenfolge von verarbeiteten Änderungen eine große Rolle spielt, muss dafür gesorgt werden, dass diese auch eingehalten wird.

1.2 Zielsetzung

Das Hauptziel der vorliegenden Arbeit war die Entwicklung eines Änderungspropagationssystems, das für heterogene und autonome Informationssysteme geeignet ist und deren Anforderungen berücksichtigt. Um die Wandlungsfähigkeit des Unternehmens zu unterstützen, sollte die angestrebte Lösung flexibel sein. Dies bedeutet vor allem, dass neue Informationssysteme hinzugenommen oder nicht mehr benötigte entfernt werden können. Außerdem soll die Änderung von Informationsmodellen (Schemas) der einzelnen Informationssysteme möglich sein. Die angebundenen Informationssysteme

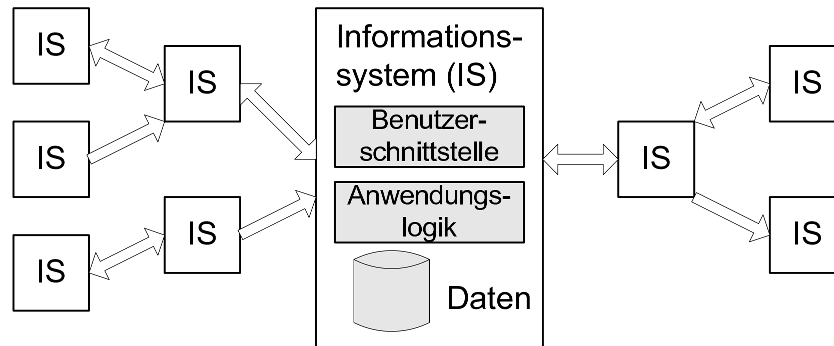


Abbildung 1.3: Aufbau von Informationssystemen und ihre Integration

sollen, wie die Leistungseinheiten, teilautonom sein. Die Informationssysteme sollen dabei keine Kenntnis über andere Informationssysteme haben, mit denen sie einen Informationsaustausch haben. Des Weiteren sollen die Informationssysteme aus Gründen des Informationsaustauschs mit anderen Informationssystemen möglichst nicht angepasst werden müssen. So ist es möglich, leicht Informationssysteme hinzuzunehmen, zu entfernen oder auszutauschen. Die Informationssysteme reichen nun ihre geänderten Informationen bzw. Daten an ein Änderungspropagationssystem weiter, dass dann die entsprechenden Informationssysteme über die Änderung benachrichtigt.

1.3 Gliederung

Kapitel 2 gibt einen Überblick über Technologien, die in Konkurrenz zur Änderungspropagation stehen oder als Grundlage für diese verwendet werden können.

In **Kapitel 3** wird das zur Grunde liegende Konzept des Änderungspropagationssystems beschrieben. Dafür werden Änderungsbeschreibungen und eine Sprache zur Definition von Propagationsprozessen definiert, die Änderungen verarbeiten. Außerdem wird eine weitere Sprache für die Definition von Bedingungen zwischen Änderungsbeschreibungen definiert. Des Weiteren werden Konzepte zur Erkennung von Konflikten und zur Einhaltung der Reihenfolge behandelt.

Mögliche Erweiterungen des Änderungspropagationssystem werden schließlich in **Kapitel 4** behandelt. Dazu gehören die Einbindung von Drittsystemen, Propagationen, die mehrere Änderungen verarbeiten können und die Verteilung des Propagationssystems.

Die Evaluierung des Ansatzes anhand eines Integrationsszenarios und eine Performanceevaluierung erfolgt in **Kapitel 5**. Zusätzlich wird der Ansatz noch mit verwandten EAI-Produkten verglichen.

Schlussfolgerungen und ein Ausblick in **Kapitel 6** bilden den Abschluss dieser Arbeit.

KAPITEL 2

Grundlagen

In diesem Kapitel wird ein Überblick über Technologien gegeben, die im Zusammenhang mit der Integration von Informationssystemen mittels Änderungspropagation stehen. Als Erstes wird der Begriff „Informationssystem“ geklärt. Danach werden Technologien untersucht, mit denen Integrationen innerhalb eines Unternehmens durchgeführt werden können. Diese sind *Enterprise-Resource-Planning-Systeme* und *Enterprise Application Integration*. Datenintegration fokussiert sich auf die Integration von Daten innerhalb eines Unternehmens. Um dies in großen und flexiblen Netzen zu ermöglichen, können *Peer-Data-Management-Systeme* eingesetzt werden. Eine weitere Technologie, durch die Informationssysteme integriert werden, stellen Workflows dar. Als Basistechnologien für die Integration von Informationssystemen sind XML und Message-oriented-Middleware (MOM) weit verbreitet. Eine weitere Basistechnologie, die verwandt zu MOM ist, sind Ereignissysteme. Da Integrationsaufgaben metadatenintensiv sind, werden noch Model-Managementsysteme untersucht. Zum Schluss wird begründet, warum die vorgestellten Technologien für eine Änderungspropagation zwischen heterogenen sowie autonomen Informationssystemen nicht ausreichend sind.

2.1 Informationssysteme

In diesem Abschnitt geht es um die Klärung des Begriffes „Informationssystem“, mögliche Architekturen und Modelle der Informationssysteme.

2.1.1 Definition

In der Literatur wird der Begriff „Informationssystem“ häufig verwendet [Vet90, BF97, Ste02, Kur02], ohne dass dabei der Begriff näher erläutert oder definiert wird. Daraus könnte man schließen, dass der Begriff und dessen Bedeutung weitläufig bekannt sind.

Da Informationssysteme in dieser Arbeit die Systeme sind, die miteinander integriert werden sollen, wird dieser Begriff zunächst genauer untersucht.

Bevor wir eine Definition für den Begriff „Informationssystem“ finden, wollen wir die Unterbegriffe „Information“ und „System“ genauer betrachten. In [Krc03] werden Informationen, Daten und Zeichen definiert. Werden auf unterster Ebene Zeichen durch eine Syntax kombiniert, erhält man Daten (z.B. 2,3). Wird zu den Daten ein Kontext hinzugefügt, wie zum Beispiel 'Gewicht des Produktes in kg', so erhält man Informationen. Die Daten sollten mit dem dazu gehörigen Kontext interpretiert werden, um Informationen zu erhalten.

Ein System ist laut [Vet94] folgendermaßen definiert:

> *Ein* ***System*** *stellt eine abgeschlossene Gesamtheit von Elementen dar, die miteinander durch Beziehungen verbunden sind und gemeinsam einen bestimmten Zweck zu erfüllen haben.*

Bei technischen Systemen werden die Elemente, aus denen ein System besteht, als Komponenten bezeichnet.

In [CS99] wird der Begriff „System“ durch seine Eigenschaften beschrieben: neue Fähigkeiten, Hierarchie, Kommunikation und Steuerung. Ein System verfügt über neue Fähigkeiten, die die einzelnen Komponenten noch nicht haben. Zum Beispiel kann man ein Auto verwenden, um von A nach B zu kommen. Dies ist eine Fähigkeit, die keine der Komponenten des Autos hat. Ein System setzt sich aus Komponenten zusammen, die wiederum Systeme sein können. Folglich besteht ein System aus einer Hierarchie von Komponenten. Zwischen den Komponenten muss eine Kommunikation stattfinden, damit die neuen Fähigkeiten realisiert werden können. Um eine sinnvolle Kommunikation zu erreichen, muss diese gesteuert werden.

Der Begriff „Informationssystem“ ist laut [Krc03] folgendermaßen definiert:

> *Bei* ***Informationssystemen*** *handelt es sich um soziotechnische ("Mensch-Maschine") Systeme, die menschliche und maschinelle Komponenten (Teilsysteme) umfassen und zum Ziel der optimalen Bereitstellung von Information und Kommunikation nach wirtschaftlichen Kriterien eingesetzt werden.*

Wie an der Definition zu sehen ist, spielt der Mensch eine große Rolle im Informationssystem, da er die Daten interpretiert und dadurch aus den Daten Informationen macht. Aus Sicht der Änderungspropagation muss der Begriff „Informationssystem“ aber nicht so streng definiert werden, da es sich dabei um ein System handelt, dessen Daten integriert werden sollen. Es kann sich also dabei auch um ein Anwendungssystem handeln, was auch oft als Teil eines Informationssystems oder als Informationssystem selbst gesehen wird. Wichtig für uns aus Sicht der Integration von Daten ist nur, dass das Informationssystem über eine persistente Datenspeicherung verfügt. Deshalb genügt in diesem Zusammenhang die folgende abgeschwächte Aussage:

> *Ein* ***Informationssystem*** *ist ein System, das einen Teil seiner Daten durch eine persistente Speicherung dauerhaft verwaltet.*

Die persistente Speicherung kann in Datenbanken oder in Dateien erfolgen.

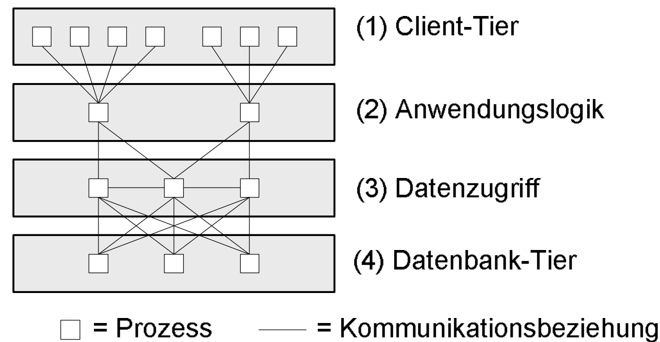


Abbildung 2.1: Beispiel eines 4-Tier-Informationssystem [Har01a]

2.1.2 Architektur

In diesem Abschnitt soll die Architektur eines Informationssystems untersucht werden, um festzustellen wo Datenänderungen eingespielt werden können. Um die Architektur von Informationssystemen zu beschreiben, kann man die Artefakte des Informationssystems Ebenen zuordnen. Verfügt ein Informationssystem über mehrere Ebenen so spricht man von einer Ebenenarchitektur. Diese können wie in [Har01b] dargestellt mehrere Schnittstellen zu unteren Ebenen haben, da sie zum Beispiel Funktionen von zwei Ebenen brauchen, um ihre Funktionalität zu implementieren. Weiterhin kann eine Ebene auch eine komplexe Unterstruktur haben.

Eine weitere Beschreibung von Informationssystem-Architekturen, ist die Unterteilung in so genannte Schichten (engl. Tiers, vgl. Abbildung 2.1). In [Har01b] ist eine Schicht(Tier) folgendermaßen definiert:

> *A tier is a layer that corresponds to a process or a collection of processes. A tier contains all artifacts of a software system that can be associated with the tiers processes.*

Wichtig bei der Definition ist, dass eine Schicht einer Ebene gleich gesetzt werden kann. Mehrere Ebenen können sich innerhalb eines Betriebssystemprozesses befinden, während eine Schicht einem oder mehreren Prozessen zugeordnet ist und damit zwischen Schichten Prozessgrenzen befinden.

Informationssysteme verfügen des Öfteren über Konsistenzregeln für ihre Daten, die oberhalb der Datenbankschicht angesiedelt sind, d.h. in den Schichten Anwendungslogik und Datenzugriff. Aus diesem Grund sollte eine Datenintegration von heterogenen und autonomen Informationssystemen auf der Ebene der Anwendungslogik erfolgen, da diese alle Konsistenzregeln beachten kann, auch die in der Datenbankschicht. Wird direkt die Datenbank für die Datenintegration angesprochen, so kann es durch die Nichtbeachtung von Konsistenzregeln der oberen Schichten zu Inkonsistenzen führen. Andererseits können alle Konsistenzregeln in der Datenbankschicht realisiert sein. Des Weiteren führt die Umgehung der Anwendungsschicht dazu, dass Anwendungslogiken teilweise im Integrationssystem nachprogrammiert werden müssen, da zum Beispiel bei Änderungen von A auch B geändert werden muss.

2.1.3 Modelle und Geschäftsobjekte

Auf einer sehr abstrahierten Ebene stellt das Modell eines Informationssystems eine Menge miteinander verbundener Geschäftsobjekte (engl. *Business Objects*) dar. Aus diesem Grund wollen wir die Geschäftsobjekte genauer untersuchen. Laut [JGJ97] ist ein Geschäftsobjekt wie folgt definiert:

> *We define a* ***business object*** *as representing something concrete and significant in the business – a representation of members of the business or "some thing" handled or used by people in the business.*

Diese Definition ist sehr weitläufig und beinhaltet dadurch eine recht große Anzahl an möglichen Geschäftsobjekten. Allerdings muss hier angemerkt werden, dass nicht in allen Quellen das Geschäftsobjekt so weitläufig definiert ist.

In [Fay02] werden dagegen drei Arten von Eigenschaften/Objekten definiert: *Enduring Business Themes*, *Business Objects* und *Industrial Objects*. *Enduring Business Themes* sind Eigenschaften des Modellierungsgegenstandes, die beständig sind. In einer Fabrik ist die Produktion ein *Enduring Business Theme*, während ein Produktionsprozess ein *Business Object* und eine Fertigungsmaschine ein *Industrial Object* ist. Um diese zu unterscheiden, stellt [Fay02] sieben Kriterien auf, die je nach Modellierungsgegenstand andere Werte haben: *Stability over time*, *Adaptibility*, *Essentiality*, *Intuition*, *Expliciteness*, *Commonitality to the Domain*, *Tangebility*. Betrachtet man zum Beispiel Zeitstabilität ("Stability over time") so sind *Enduring Business Themes* stabil, während Geschäftsobjekte nach außen stabil sind, sich aber intern ändern können. In unserem Fall wollen wir die Unterscheidung nicht so stark vertiefen, sondern Geschäftsobjekte als Modell realer (z.B. Resource) oder künstlicher Objekte (z.B. Auftrag) verstehen, die eine bestimmte Funktion im Unternehmen haben.

Gruppen von Geschäftsobjekten lassen sich zu Geschäftsobjekttypen zusammenfassen, die sich in einer Meta-Ebene weiter oben befinden. Geschäftsobjekttypen entsprechen Klassen. In diesen Geschäftsobjekttypen werden die gemeinsamen Eigenschaften der Geschäftsobjektgruppe festgelegt.

Weiterhin kapselt ein Geschäftsobjekt die Daten und Funktionalitäten desselben [MS00]. Dies ist natürlich ähnlich zu der Definition programmiersprachlicher Objekte, bei der die Daten durch Methoden gekapselt werden. Die Funktionalität ist durch die bereitgestellten Methoden realisiert und liefert dadurch eine kontrollierte Möglichkeit den Zustand zu ändern. Geschäftsobjekte haben laut [SE98] spezielle Implementierungsobjekte, die die Repräsentation desselben für Benutzer anderer Geschäftsobjekte oder anderer Systeme anbieten. Es muss angemerkt werden, dass ein Geschäftsobjekt auch Implementierungsobjekte zur persistenten Speicherung verwendet. Vergleicht man die Implementierungsklassen, aus denen ein Geschäftsobjekttyp besteht, mit dem Modell eines Informationssystems, so stellt laut [Sch98] der Geschäftsobjekttyp ein Teil des Fachkonzepts dar, während die Hilfsklassen Bestandteil des DV-Konzeptes sind. Dabei ist anzumerken, dass die Geschäftsobjekte Beziehungen untereinander haben. Eine Ressource hat zum Beispiel Beziehungen zu Produkten, die sie bearbeitet und zu Produktionsaufträgen, die sie fertigt. Dabei gibt es eine Vielzahl von Beziehungen, die ein solches Geschäftsobjekt mit anderen Geschäftsobjekten unterhalten kann. In

UML [BRJ99], eine Sprache mit der man beispielsweise Geschäftsobjekte modellieren kann, gibt es vier Arten von Beziehungen: Vererbung (*Is-A*), Assoziation, Aggregation und Komposition. Die Komposition stellt dabei eine Sonderrolle da, denn die interne Zusammensetzung eines Geschäftsobjektes durch Hilfsobjekte, wie zum Beispiel *Kundenauftrag* und *Auftragsposition*, werden üblicherweise durch Kompositionen modelliert und stellen dadurch eine enge Beziehung zwischen den Objekten dar. Je nach Betrachtungsebene können auch die Geschäftsobjekte selbst durch Kompositionen miteinander verbunden sein. Ein Beispiel hierfür ist ein Auto, das abstrakt gesehen ein Geschäftsobjekt darstellt. Dieses Geschäftsobjekt kann aber auch detaillierter betrachtet werden und damit in seine Komponenten unterteilt werden.

Die Integration von Daten kann auf der Ebene von Geschäftsobjekten oder persistenten Implementierungsobjekten erfolgen. Weil Implementierungsobjekte einen engen Zusammenhalt haben, eignen sich Geschäftsobjekte besser für die Integration. Ein Beispiel hierfür sind Auftragskopf und Auftragspositionen (Implementierungsobjekte), die einen Kundenauftrag bilden (Geschäftsobjekt). Des Weiteren stellen Geschäftsobjekte Objekte im Fachkonzept dar und ermöglichen dadurch eine bessere Diskussionsmöglichkeit in Integrationsprojekten.

Da Geschäftsobjekte intern Kompositionen verwenden, eignet sich besonders XML als Format für den Datenaustausch [Dau03]. Der Grund hierfür ist die ebenfalls hierarchische Datenstruktur von XML. In einem späteren Abschnitt (2.7) dieses Kapitels wird auf XML noch detaillierter eingegangen.

Betrachtet man nun ein produzierendes Unternehmen oder auch andere Unternehmen, so verfügen diese in den meisten Fällen über mehrere Informationssysteme, die im Laufe der Zeit selbst entwickelt oder hinzugekauft wurden. Unternehmen verfügen aus diesem Grund nicht über ein globales Unternehmensmodell. Selbst wenn das Unternehmen SAP R/3 einsetzt, ist dieses Modell nicht all umfassend. Beispiele hierfür sind die wichtigen Module auf der Ebene der taktischen Produktionsplanung, wie zum Beispiel das Fabriklayout oder auch das Produktdatenmanagement. In Abbildung 2.2 wird dargestellt, dass sich ein Unternehmensmodell in den meisten Fällen aus mehreren Partialmodellen zusammensetzt. Diese Partialmodelle sind nicht disjunkt, sondern überschneiden sich in manchen Bereichen. Dies bedeutet, dass Daten von beiden betroffenen Systemen, die die Überschneidung bilden, benötigt werden. Diese überschneidenden Bereiche sollten möglichst klein sein. Eine große Überschneidung kommt dann zustande, wenn mehrere Informationssysteme ähnliche Aufgaben erledigen. Diese Überschneidungen müssen mittels Integrationssystemen integriert werden, damit eine globale Datenkonsistenz gewährleistet ist. Bevor wir die *Enterprise Application Integration* (EAI) untersuchen, wollen wir noch einen Blick auf die *Enterprise-Resource-Planning-Systeme* werfen, zu denen auch das oben erwähnte SAP R/3 gehört.

2.2 Enterprise-Resource-Planning-Systeme

Enterprise-Resource-Planning-Systeme, abgekürzt ERP-Systeme, sind aus *Material-Requirements-Planning-* (MRP) und aus *Manufacturing-Resource-Planning*-Systemen (MRP II) entstanden [KvH00]. ERP-Systeme sind dabei konfigurierbare Informati-

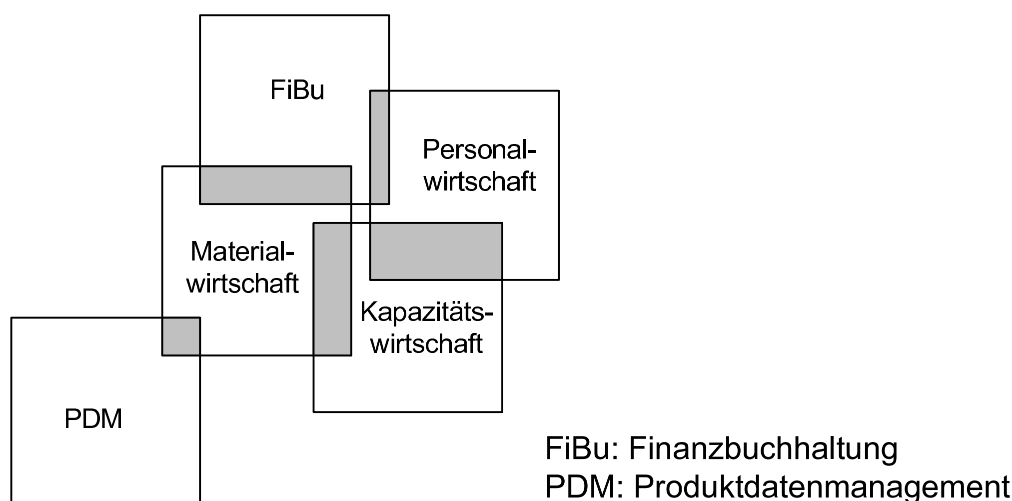


Abbildung 2.2: Das Unternehmensmodell und seine Partiallmodelle [Hei00]

onssystempakete, die Daten sowie Prozesse innerhalb von Unternehmensfunktionen übergreifend integrieren [KvH00]. In Abbildung 2.3 sind die wichtigsten Module eines ERP-Systems bzw. Unternehmensfunktionen und deren Bezüge dargestellt. Durch diese Integration wird es möglich, genauere und zeitnahe Informationen über Abläufe im Unternehmen zu bekommen [PG00]. Dies bedeutet zum Beispiel, dass für den Kunden genauere Informationen und auch Geschäftsregeln vorliegen, wie beispielsweise der maximale Kredit eines Kunden. Durch ein ERP-System werden außerdem die Kosten gesenkt, Reaktionszeiten verkürzt und der Kundenservice erhöht [STSB02].

Allerdings verlangt der Einsatz ein *Business Reengineering* [HC94], bei dem bestehende Geschäftsprozesse analysiert und dann für die Verwendung im ERP-System angepasst werden. Die Geschäftsprozesse müssen dabei so angepasst werden, dass sie durch das gewählte ERP-System realisierbar sind. Diese Aufgabe ist oft langwierig und teuer. Laut [Has00] sollte sich eine Anwendung an die Geschäftsorganisation anpassen und nicht umgekehrt. Als Argument für ein ERP-System spricht dabei, dass durch ein solches System die besten Geschäftspraktiken realisiert werden. Allerdings wie in [SKTY00] dargestellt, gibt es aber erhebliche Unterschiede zwischen den Anforderungen aufgrund unterschiedlicher Kulturen und Länder. Dies führt zu einem erheblichen Anpassungsbedarf. Außerdem führt die Verwendung der konfigurierbaren Geschäftsprozesse der ERP-Systeme zu einer Vereinheitlichung der Unternehmen, was zu einem Verlust von Wettbewerbsvorteilen führen kann. Aus diesen Gründen haben manche Unternehmen entschieden, kein Standard ERP-System einzusetzen und dafür eine Eigenentwicklung zu verwenden. ERP-Systeme können an spezielle Anforderungen angepasst werden, was aber dazu führt, dass neuere Versionen schwer einzupflegen sind, da der Code des ERP-Systems verändert wurde. Bei vielen Projekten ist die Einführung von ERP-Systemen gescheitert. Allerdings gibt es auch viele erfolgreiche Beispiele, wie zum Beispiel in [Bro04], bei dem ein ERP-System für die Verwaltung eines College eingesetzt wird.

Weiterhin stellt bei internationalen Unternehmen mit einer breiten Produktband-

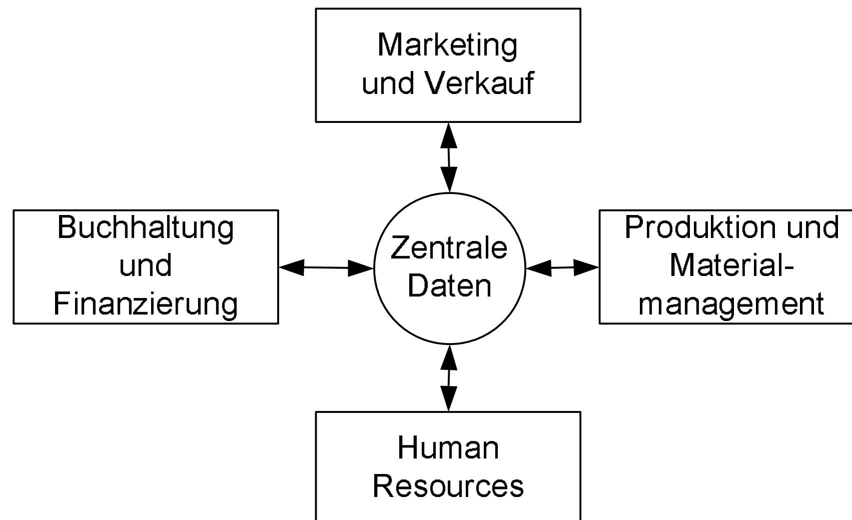


Abbildung 2.3: ERP-System und Integration [BMW01]

breite die ERP-Implementierung an mehreren Standorten eine Herausforderung dar [MTvF00], die abhängig von der gewählten Geschäftsstrategie (z.B. mehrere teilautonome Standorte) ist. Insbesondere ist hier das Ausmaß der Autonomie der Geschäftsbereiche ausschlaggebend. Für wandlungsfähige Unternehmen wird dabei von einem hohen Grad von Autonomie ausgegangen, um eine schnelle Reaktion auf Veränderungen zu ermöglichen [Son99].

2.3 Enterprise Application Integration

Enterprise Application Integration (EAI) ist ein Schlagwort, das in der heutigen Zeit häufig verwendet wird. Unter diesem Begriff versteht man die Anstrengung, die im Unternehmen vorhandenen Anwendungen und Informationssysteme miteinander zu integrieren. Die EAI-Technologie entstand dabei Mitte der 90er Jahre [LSH03]. Da mit EAI unterschiedlichste Anwendungen und Informationssysteme integriert werden und in dieser Arbeit Informationssysteme miteinander integriert werden sollen, wollen wir diese Technologie genauer untersuchen.

In [LSH03] wird ausgesagt, dass beim Einsatz von EAI eine geringere Anstrengung für die Umsetzung innerhalb eines Unternehmens notwendig ist als durch die Verwendung eines ERP-Systems. Durch diese Integration mit EAI wird erreicht, dass einheitliche Daten vorhanden und Geschäftsprozesse systemübergreifend realisiert sind. Die Kostenersparnis ist dadurch begründet, dass in existierenden Systemen Prozesse schon teilweise realisiert sind und die Daten bereits verwaltet werden. Mittels EAI können nun die vorhandenen Daten bzw. Prozesse integriert werden.

Abbildung 2.4: Ebenen in einem Unternehmen [Cum02]

2.3.1 Unternehmenssicht

Mit dem Schlagwort EAI ist sehr deutlich herausgehoben, dass es sich um Anwendungsintegration innerhalb eines Unternehmens handelt. Die Anwendungsintegration kann auf verschiedenen Ebenen innerhalb des Unternehmens stattfinden. Abbildung 2.4 stellt diese unterschiedlichen Ebenen dar. Auf unterster Ebene werden unterschiedliche Anwendungskomponenten zu einer Anwendung integriert. Diese können dann durch unterschiedliche Geschäftsprozesse integriert werden, welche selber in unterschiedlichen Geschäftsdomänen existieren. Aus diesen setzt sich schließlich ein Unternehmen zusammen, das wiederum Bestandteil einer oder mehrerer virtueller Unternehmen sein kann. Diese Art von Unternehmen sind Unternehmenszusammenschlüsse, die ein gemeinsames Ziel erreichen wollen [AFHS95].

Ein Unternehmen kann durch verschiedene Organisationsansätze strukturiert werden [Krü84]. Dies kann sich deutlich auf die Integration auswirken. Dadurch können weitgehend unabhängige Bereiche geschaffen werden, wie zum Beispiel Divisionen. Der Integrationsaufwand zwischen den Bereichen wird verringert, wenn die Bereiche unabhängiger sind. Im Sonderforschungsbereich 467 [Son99] wurden Leistungseinheiten eingeführt, die weitgehend autonom sind, aber dennoch über Schnittstellen zu anderen Leistungseinheiten verfügen. Der Grad der Autonomie hat einen großen Einfluss auf die Wahl der Integrationslösung, denn die Softwaresysteme sollten bei hoher Autonomie der Unternehmensbereiche auch selbst weitgehend autonom bleiben.

2.3.2 Klassifikationen

In der Literatur werden unterschiedliche Klassifikationen zum Thema EAI angegeben, die in der Tabelle 2.1 zusammengefasst sind. Dabei gibt es unterschiedliche Meinungen zu den Integrationsebenen, die sich teilweise überschneiden, bisweilen auch widerspre-

Klassifikation	Ausprägungen
Dimensionen [Has00]	• Autonomie • Heterogenität • Verteilung
Integrationsebenen I [RMB01]	• Präsentationsebene • Datenebene • Datenkonsistenzebene • Funktionsebene • Prozessebene • Komponentenebene
Integrationsebenen II [Lin00]	• Datenebene • Funktionsebene • Applikationsschnittstellenebene • Methodenebene
Grad der Kopplung [RMB01]	• Lose Kopplung • Enge Kopplung
Offenheit der zu integrierenden Anwendung [RMB01]	• White-Box • Black-Box
Bereich [LJdP97]	• Innerhalb eines Unternehmens • Zwischen Unternehmen
Art [LJdP97]	• Interfacing • Integration
Architekturen [Mül05]	• Point-to-Point • Hub & Spoke • Bus-orientiert • Verteile Objekte

Tabelle 2.1: Klassifikationen zum Thema EAI

chen. Wie man sieht, haben die Integrationsebenen (I und II) Überschneidungen, aber auch disjunkte Teile. Im folgenden Absatz wird die Integrationsebene I erklärt.

Die Präsentationsebene integriert verschiedene Anwendungen zu einer einheitlichen Benutzerschnittstelle, so dass es für den Benutzer aussieht, als ob es sich um eine einzige Anwendung handelt. Ein ERP-System verfügt über mehrere Anwendungen, hat aber wie am Beispiel von SAP R/3 ersichtlich auch eine einheitliche Benutzerschnittstelle. Im Umfeld von Web-Technologien eignen sich besonders Portale um unterschiedlichste Anwendungen in einer Benutzerschnittstelle zusammenzuführen. Ein weiteres wichtiges Schlagwort in diesem Zusammenhang ist das einheitliche und anwendungsübergreifende „Look-And-Feel". Dies ermöglicht dem Benutzer ein schnelles Zurechtfinden in fremden Anwendungen.

Unter der Datenebene wird hier hauptsächlich die Integration autonomer Daten-

quellen zu einer einheitlichen Zugriffsebene verstanden, wie zum Beispiel durch Föderierte Datenbanksysteme (Abschnitt 2.4.2). Bei dieser Art von Datenbanksystemen wird zusätzlich noch ein einheitliches Schema bereitgestellt, über das die einzelnen Datenquellen integriert sind.

Bei der Integration auf der Ebene der Datenkonsistenz werden im Vergleich zur Datenebene die autonomen Datenquellen so integriert, dass ihre Daten konsistent untereinander bleiben. Dies kann durch eine globale Anwendung erfolgen, die Datenänderungen an alle Anwendungen sendet oder auch durch den Austausch von Änderungsinformationen untereinander. Begriffe, die in diesem Zusammenhang oft auftauchen, sind Synchronisation und Propagation. *Update Propagation* wird zum Beispiel von replizierten Datenbanken verwendet, um die Daten zu synchronisieren. In [RMB01] gehört die konsistenzerhaltende Datenintegration hingegen zur Funktionsintegration, da im Beispiel des Buches [RMB01] eine globale Anwendung zwei Änderungsfunktionen aufruft. Wir dagegen sind der Meinung, dass die konsistenzerhaltende Datenintegration auch unabhängig von den Funktionen erledigt werden kann, indem zum Beispiel Veränderungen direkt in der Datenhaltungsschicht erkannt und verteilt werden. Dies ist aber nicht in allen Fällen zu empfehlen, da dadurch Konsistenzregeln umgangen werden, die in der Anwendungsschicht realisiert sind. Besser ist es, Änderungen im heterogenen Umfeld über die Anwendungsschichten einzuspielen, sofern eine Schnittstelle bereitgestellt wird. Sind keine Konsistenzregeln in der Anwendungsschicht realisiert oder handelt es sich um eine homogene Systemlandschaft, kann die Anwendungsschicht umgangen werden.

Die Funktionsebene integriert Anwendungen durch den gegenseitigen Aufruf von Funktionen, die zum Beispiel als RPC oder über verteilte Objekttechnologien bereitgestellt werden.

Auf der Prozessebene werden unterschiedliche Funktionen zu einem Geschäftsprozess integriert. Dies erfolgt häufig durch die Verwendung von so genannten Workflow-Management-Systemen (WFMS) und wird in Abschnitt 2.6 noch genauer untersucht.

Als letzte Ebene ist die Komponentenintegration zu sehen, bei der Anwendungen aus verschieden Komponenten zusammengesetzt werden. Dies verkürzt die Anwendungsimplementierung, da Komponenten wieder verwendet werden können.

Linthicum [Lin00] schlägt in seinem Buch noch die Methodenintegration vor. Diese soll wieder verwendete Methoden, die Geschäftslogiken realisieren, identifizieren und global bereitstellen. Im Gegensatz zur Funktionsintegration sollen hierbei die einzelnen Anwendungen angepasst werden, um Redundanzen zu vermeiden. Dies ist nicht immer machbar, da Legacy-Systeme oft schlecht veränderbar sind. Außerdem ist mit dieser Integrationsebene ein hoher oft nicht realisierbarer Aufwand verbunden.

Häufig wird die Art der Integration auch noch durch die Stärke der Kopplung unterschieden: enge und lose Kopplung. In der Literatur herrschen zu diesen Begriffen unterschiedliche Meinungen zu ihrer Definition. In [RMB01] wird zum Beispiel bei der losen Kopplung von der Abhängigkeit weniger Schnittstellen, während in [Cum02] über asynchrone Kommunikation im Zusammenhang mit loser Kopplung gesprochen wird.

Unterschieden werden muss noch zwischen der Integration innerhalb eines Unternehmens, also die klassische EAI, und der Integration zwischen Unternehmen (B2B),

die neue Anforderungen (z.B. kein zentrales Integrationssystem) mit sich bringt [SH01].

2.3.3 Technologien

Die Technologien, die zur Realisierung eines EAI-Projektes eingesetzt werden können, sind vielfältig. Sie unterscheiden sich hauptsächlich durch die Integrationsebenen (Tabelle 2.1). Für die Datenintegration können föderierte Datenbanksysteme oder Datenbankmiddleware (z.B. JDBC oder ODBC) verwendet werden. Für die Funktionsintegration können RPC-Systeme (z.B. DCE) oder verteilte Objektsysteme wie CORBA [Zah99] und COM+ zum Einsatz kommen. Zur Integration von Komponenten und ihr Deployment eignen sich J2EE und das *Corba Component Modell* CCM. Weiterhin können zur Entkopplung der Anwendungen Message-Systeme eingesetzt werden, die die Anwendung Point-to-Point mit einem Message-Broker oder einem Process-Broker [JWP00] integrieren.

Nach der allgemeinen Betrachtung von EAI sollen im nächsten Abschnitt die unterschiedlichen Arten von Datenintegrationen angeschaut werden.

2.4 Datenintegration im Unternehmen

In diesem Abschnitt werden die unterschiedlichen Arten der Datenintegration untersucht (vgl. Abbildung 2.5). Diese können unterteilt werden in das Vorhandensein eines globalen Schemas[1] mit homogener Infrastruktur (Zentrale Datenbank, Verteilte Datenbank und replizierte Datenbanken), föderierte Datenbanken (globales Schema sowie autonome, heterogene Komponentendatenbanken) und lokale Modelle (Schemata), die über Geschäftsprozesse integriert sind.

2.4.1 Globales Schema und homogene Systemlandschaft

2.4.1.1 Zentrale Datenbank

Ein globales Schema wird häufig mit einer zentralen Datenbank in Verbindung gebracht. Dabei werden alle Daten in einem DBMS und einer Datenbank gespeichert. Dieser Ansatz wird häufig angestrebt, da es keine Probleme mit Duplikaten gibt. Da alle Daten auf einem Rechner sind, kann dieser Rechner schnell zum Flaschenhals werden. Außerdem ist keine hohe Ausfallsicherheit gegeben. Dieser Ansatz wurde unternehmensweit in den 70er Jahre forciert, als die gesamten Unternehmensdaten auf Großrechnern verwaltet wurden. Wenn die Leistung nicht ausreicht oder eine Datenlokalität[2] gefordert ist, können Verteilte Datenbanksysteme in Betracht gezogen werden.

[1] Hier wird Schema verwendet, da Datenmodell in der Literatur anders belegt ist (z.B. Relationales Datenmodell)

[2] Datenlokalität steht hier für die Verwaltung von Daten, wo sie anfallen und benötigt werden. Das bedeutet beispielsweise, dass deutsche Kunden in der deutschen Filiale und nicht in der amerikanischen Zentrale verwaltet werden.

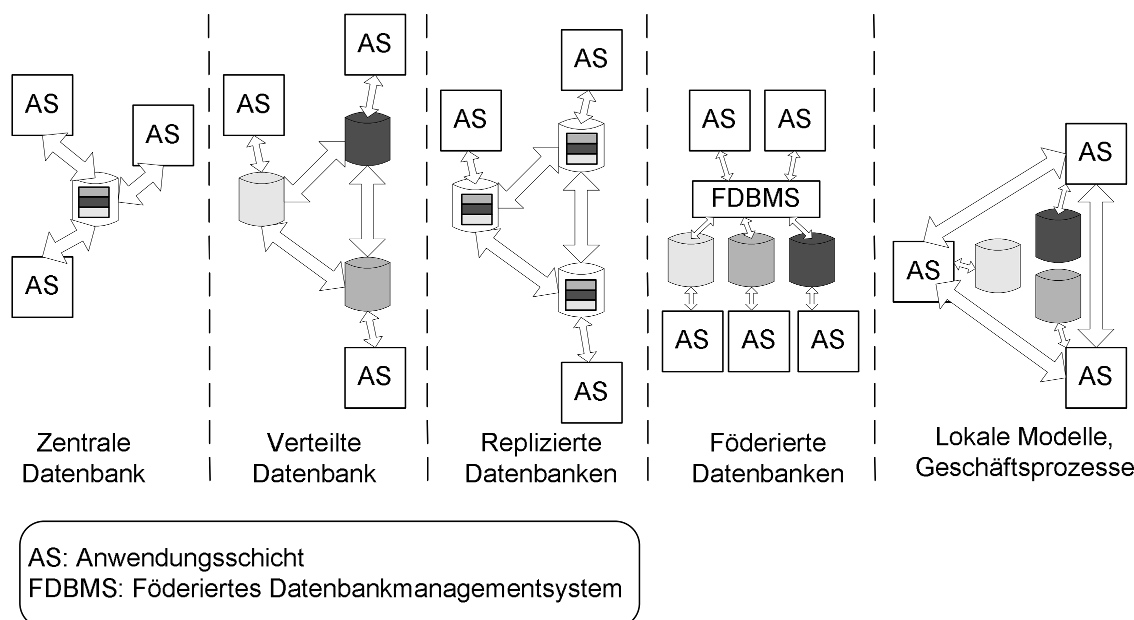


Abbildung 2.5: Arten von Datenintegrationen

2.4.1.2 Verteilte Datenbanken

Bei verteilten Datenbanken [Rah94, Dat00] werden die Daten einer Datenbank auf mehrere DBMS verteilt, um eine höhere Performanz und Datenlokalität zu erreichen. Dabei können die Daten repliziert oder nicht repliziert vorliegen. Da der erste Fall mit den replizierten Daten besonders wichtig für diese Arbeit ist, wird ihm ein extra Abschnitt gewidmet (Abschnitt 2.4.1.3).

Für den Datenbankanwender soll die verteilte Datenbank wie eine zentrale Datenbank erscheinen. Dies wird durch unterschiedliche Arten von Transparenz erreicht [Rah94]. Die Daten des globalen Schemas der verteilten Datenbank werden mittels Fragmentierung und Allokation auf die einzelnen physischen Datenbanken verteilt. Die Fragmentierung bildet Gruppen von Daten, die dann mit dem Allokationsschritt auf die Datenbanken verteilt werden. Relationen können als Ganzes auf die einzelnen Datenbanken verteilt werden. Die Fragmentierung kann ebenfalls innerhalb von Relationen erfolgen. Bei der horizontalen Partitionierung werden Zeilen auf verschiedene Datenbanken verteilt, während bei der vertikalen Partitionierung einzelne Spalten auf die Datenbanken verteilt werden.

Verteilte Datenbanken sind auf ein homogenes Umfeld beschränkt. Um dennoch Heterogenität zwischen den einzelnen Datenbanken zu ermöglichen, gibt es zwei Ansätze [dFRH98]: Database-Gateways und Database-Middleware[3]. Dadurch werden Heterogene Verteilte Datenbanken möglich [Dat00], d.h. ein anderes DBMS verwendet ein Gateway um eine Datenbank eines fremden DBMS einzubinden. Außerdem können Database-Gateways auch dafür eingesetzt werden, um Anwendungen die gewohnte

[3]Der Begriff Database-Middleware wird hier und in der angegeben Literatur anders verwendet als in Abschnitt 2.3.3

Sicht eines DBMS zu bieten, obwohl die Datenbank von einem anderen DBMS verwaltet wird [ACM00]. Dadurch kann auch Heterogenität bezüglich Datenmodellen beseitigt werden, wie am Beispiel eines SQL Gateway für IMS [Pau93]. Der zweite Ansatz, Database-Middleware, wird auch als „Föderierte Datenbanken“ bezeichnet. Föderierte Datenbanken sind auch eine Art von verteilten, heterogenen Datenbanken bei der die globale Sicht nur über die Föderation sichtbar ist (siehe Abschnitt 2.4.2).

2.4.1.3 Replikation

Nicht nur die Verteilung der Daten auf unterschiedliche Rechner bringt eine Datenlokalität und Performance, sondern auch die redundante Datenhaltung durch Replikation. Allerdings bringt die Replikation nicht nur Vorteile, sondern auch Nachteile, die je nach Methode der Replikation anders geartet sind. Diese Methoden lassen sich danach unterscheiden, wie Änderungen an die Kopien mitgeteilt werden [GHOS96]. Die Aktualisierung der Kopien erfolgt bei der „*Eager Replication*“ [KA00] noch bevor die Transaktion abgeschlossen wird. Bei *Lazy Replication* hingegen werden zuerst die Transaktionen *committed* und dann die Kopien über Änderungen informiert. Außerdem kann man unterscheiden, ob Änderungen in einer Gruppe von Kopien oder nur in einem ‘*Master*’ vorgenommen werden können. Der Master kann sich dabei von Objekt zu Objekt unterscheiden [GHOS96]. Die beiden Eigenschaften können in Lösungen beliebig kombiniert werden. Zum Beispiel kann es zu Änderungskonflikten kommen, wenn Lazy Replikation mit Änderungen in einer Gruppe von Kopien kombiniert wird. Diese Konflikte müssen schließlich erkannt und aufgelöst werden. Ein Nachteil von Eager Replikation ist die hohe Verfügbarkeitsanforderung – alle Systeme müssen bei einer Änderung verfügbar sein und es darf keine Netzwerk-Partition existieren [AT89]. Dieses Problem kann durch Voting-Verfahren [Rah94, JM90] verbessert werden. Dadurch kann der größere Teil der Netzwerk-Partition noch weiter arbeiten, da dieser die Mehrheit der Stimmen bekommen kann. Die Replikation kann durch das DBMS erfolgen, was eine homogene Systemlandschaft zur Folge hat, oder durch Datenbank-Middleware [PMJPKA05, LKPMJP05, MFJPPMK04]. Außerdem gibt es noch die Möglichkeit *Snapshots* [AL80, LHM+86] bereitzustellen, die nur lesbar sind und keine allzu großen Anforderungen an die Datenaktualität haben.

2.4.2 Föderierte Datenbanken

Mit einem Föderierten Datenbankmanagementsystem (FDBMS) werden einzelne DBMS und ihre Daten zu einer Einheit zusammengeschlossen. Die einzelnen DBMS können dabei untereinander heterogen sein und sollen so weit wie möglich ihre Autonomie behalten [Con97]. Dabei soll mit einem FDBMS vor allem semantische Heterogenität überwunden werden und eine Verteilungstransparenz geschaffen werden [Rah94]. Semantische Heterogenität tritt dann auf, wenn bei gleichen oder in Beziehung stehenden Daten unterschiedliche Bedeutungen, Interpretationen oder Verwendungszwecke existieren [SL90].

In Abbildung 2.6 ist die grundsätzliche Architektur eines föderierten Datenbanksystems illustriert. Für ein solches System existieren zwei Arten von Anwendungen: lokale

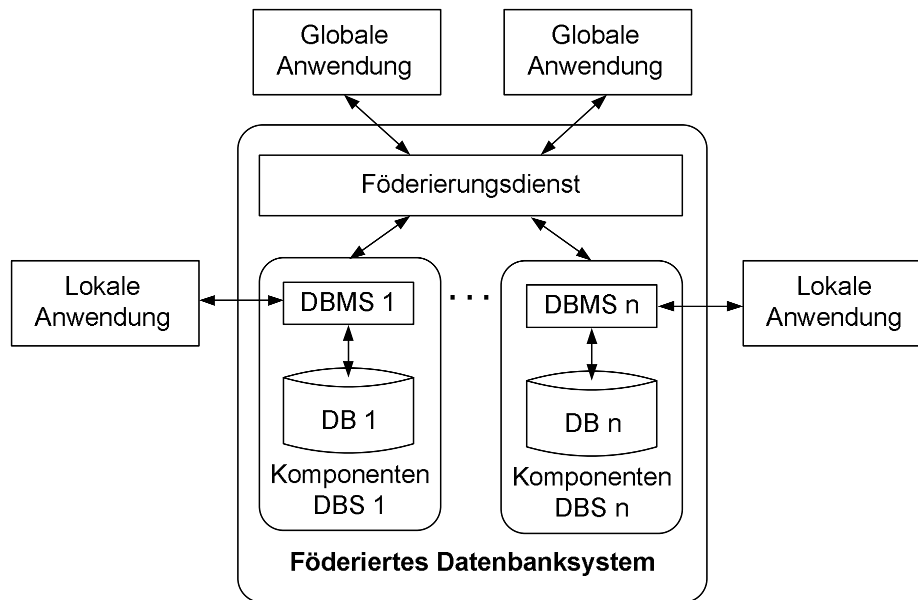


Abbildung 2.6: Architektur eines Föderierten Datenbanksystems [Con97]

und globale Anwendungen. Die lokalen Anwendungen verwenden weiterhin die Dienste des lokalen DBMS während die globalen die einheitliche Sicht verwenden, die durch das FDBMS bereitgestellt wird. Kern eines solchen FDBMS ist der Föderierungsdienst, der die einzelnen Komponenten-Datenbanksysteme integriert. Dafür verwendet er die Schnittstellen der Komponenten-DBMS, die über lokale Datenbanken verfügen.

Wichtig bei Föderierten Datenbanken ist die Schema-Architektur [SL90], die aus fünf Ebenen besteht. Das lokale Schema ist das Schema der lokalen Datenbank, des Komponenten-DBMS. Darauf aufbauend existieren Komponenten-Schemata, welche die lokalen Schemata in Schemata überführen, deren Datenmodell dem des FDBMSs entspricht [Con97]. Da nicht das gesamte Schema in einer Föderation teilnehmen soll, wird ein Export-Schema definiert, das einem Ausschnitt des Komponenten-Schema entspricht. Diese Export-Schemata werden zu einem föderierten Schema integriert. Aus dem föderierten Schema werden externe Schemata gebildet, die für die jeweiligen Anwendungen zugeschnitten sind.

Da bei vielen Informationssystemen der Zugriff nicht direkt über die darunter liegenden DBS erfolgen soll, wurde in der Dissertation von Klaudia Hergula [Her03] eine Möglichkeit geschaffen, Funktionen zu föderierten Funktionen zu integrieren. Dafür wurde ein Workflow Management System (WFMS) verwendet. Zusätzlich werden die Daten über ein FDBMS integriert, wobei beide Komponenten (WFMS und FDBMS) ebenfalls miteinander integriert wurden.

2.4.3 Lokale Modelle, Geschäftsprozesse und einheitliche Benutzerschnittstelle

Das ERP-System R/3 von SAP (vgl. Abschnitt 2.2) verfügt über eine anwendungsübergreifende einheitliche Benutzerschnittstelle. Dies ermöglicht ein leichtes Zurechtfinden in fremden Anwendungen. Bei einem SAP-System sind nur die benötigten Anwendungen zu installieren. Damit diese Anwendungen miteinander integriert werden können, tauschen diese Nachrichten aus. Dies basiert auf der Technologie *Application Link Enabling* (ALE) und *Intermediate Documents* (IDoc) [SAP07]. Das bedeutet, dass eine Anwendung nicht direkt in die Daten einer anderen Anwendung schreibt. Die Nachrichten (IDoc) stoßen stattdessen in der anderen Anwendung Geschäftsprozesse an, die dann die Daten entsprechend anpassen. Im nächsten Abschnitt untersuchen wir noch eine weitere Art von Datenintegration in großen verteilten Netzen: Peer-Data-Managementsysteme.

2.5 Peer-Data-Management

Peer-to-Peer-Systeme (P2P-Systeme) sind Rechnernetze, die es ermöglichen eine Vielzahl von Peers in einem Rechnernetz zu betreiben. Im Gegensatz zu Client-Server, kann ein Peer sowohl Dienste in Anspruch nehmen als auch anbieten, wobei die Peers in einem Rechnernetz gleichberechtigt sind. Eine Ausnahme sind die sogenannten *Superpeers*. Das Netz an sich ist selbst organisierend. Eines der Vorteile eines solchen Netzes ist die geringe Administration, da keine zentrale Infrastruktur benötigt wird. Für diese Arbeit sind die Peer-Data-Management-Systeme, ein Teilbereich der P2P-Systeme, wichtig, die sich auch mit Datenintegration beschäftigen. Diese Systeme werden nachfolgend genauer untersucht.

In einem Peer-Data-Management-System (PDM-System) [TIM+03, GHI+01] [BGK+02, HIM+04] fragen die Peers semantisch reiche Daten an oder stellen Daten bereit. Außerdem können die Daten der einzelnen Peers mittels *Update Propagation* angepasst werden [BGK+02]. Weitere Peers können Rechnerleistung für die Anfrageverarbeitung bereitstellen. Zwischen den einzelnen Peers bestehen semantische Beziehungen mit denen *Update Propagationen* oder Anfrage-Beziehungen definiert werden. Anfrage-Beziehungen erfolgen in Form von Sichten und definieren dessen Zusammensetzung aus weiteren Sichten bzw. Tabellen. Der letztere Teil ist verwandt mit der klassischen Datenintegration (FDBMS, Abschnitt 2.4.2). Im Piazza-Projekt [TIM+03, GHI+01, HIM+04] werden ebenfalls *Global-As-View*- und *Local-As-View*-Konzepte verwendet, um die semantischen Beziehungen zwischen den einzelnen Schemata zu beschreiben. Im Gegensatz zu einem FDBMS wird allerdings nicht ein globales Schema definiert, sondern jeder Peer definiert seine eigenen Beziehungen. Im Gegensatz zu reinen P2P-Systemen wird bei einem Peer-Data-Management davon ausgegangen, dass Peers das System nicht so häufig verlassen [HIM+04], d.h. das System is beständiger. Dennoch wird in einem Peer-Data-Management-System von einer Open-World-Assumption ausgegangen, bei der Ergebnisse von Anfragen unvollständig sein können [TIM+03].

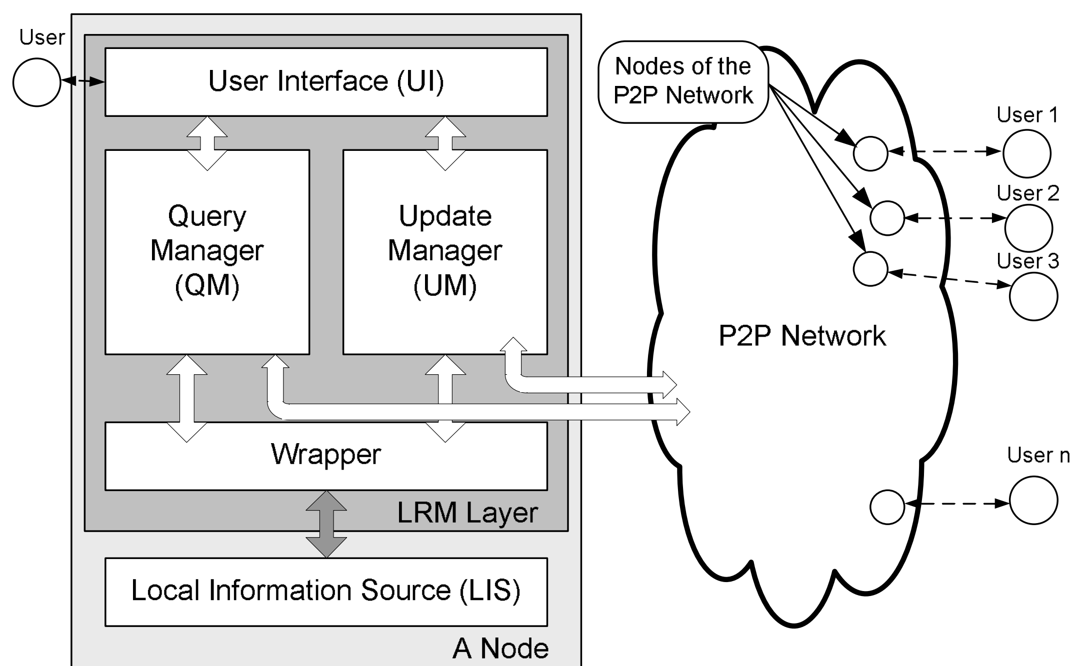


Abbildung 2.7: Architektur eines Peers in einem PDM-System [BGK+02]

Die Architektur eines Peers ist in Abbildung 2.7 dargestellt. Der Benutzer kann mittels der Benutzerschnittstelle (UI) Anfragen stellen, die dann an den *Query Manager* weitergeleitet werden. Dieser kommuniziert mit lokal bereitgestellten Daten (*Local Information Source*) oder mit anderen Peers im Netzwerk. Außerdem können Daten mittels *Update Propagationen* angepasst werden, was durch den *Update Manager* realisiert wird.

2.6 Workflows

2.6.1 Grundlagen

Workflow-Managementsysteme (WFMS) dienen zur Realisierung von Geschäftsprozessen. Ein *Geschäftsprozess* stellt dabei die betriebswirtschaftliche Sicht eines Unternehmensprozesses dar [JBS97]. Unter einem Unternehmensprozess wird ein Bündel von Aktivitäten verstanden, welcher einen oder mehrere Inputs hat und für den Kunden ein Output erzeugt, der für den Kunden einen Wert hat [HC94]. Die für die Informatik wichtige Implementierungssicht eines Geschäftsprozesses wird dabei als Workflow bezeichnet. Dabei muss man noch zwischen Modellen von Geschäftsprozessen sowie Workflows und deren Instanzen, die die eigentliche Ausführung eines Modells darstellen [LR00], unterscheiden. In diesem Zusammenhang wird auch zwischen *Design-Time* und *Runtime* oder auch Beschreibung und Ausführung [BW95] unterschieden. In der Design-Time bzw. Beschreibung wird der Workflow definiert, während in der Runtime bzw. Ausführung Instanzen der Beschreibungen erstellt werden, die dann ausgeführt

werden. Ein Workflow-Modell besteht aus einer Menge von *Tasks* die über einen Kontrollfluss miteinander verbunden sind. Ein Task ist eine logische Einheit, die als Ganzes von einer *Ressource* ausgeführt wird. Eine Ressource kann entweder eine Person, ein Computersystem oder eine sonstige Maschine sein [vdAvH02]. Um eine Aktivität auszuführen muss das Workflow-Managementsystem eine Ressource zuordnen. Für die Zuordnung von menschlichen Ressourcen wird eine sogenannte Organisationsstruktur verwendet, dies wird in der Literatur als *staff resolution* bezeichnet [LR00].

2.6.2 Workflow-Managementsysteme (WFMS)

Um die Workflow-Managementsysteme besser zu verstehen, wollen wir uns die Architektur eines solchen anschauen. Recht gut hierfür geeignet ist das Referenzmodell der Workflow Management Coalition (Abbildung 2.8), die sich als Ziel gesetzt hat, die Interoperabilität der Workflow-Produkte zu erhöhen [JBS97]. Aus diesem Grund handelt es sich beim Referenzmodell auch um eine Schnittstellendarstellung, bei der die benötigten Schnittstellen zu anderen Komponenten definiert werden. Kern des Referenzmodells ist der *Workflow Enacting Service*, der wiederum aus einer oder mehrerer *Workflow-Engines* besteht. Die Workflow-Engine ist dabei für die Ausführung eines Workflow zuständig. Dessen Beschreibung wird über eine Design-Time-Komponente (*Process Definition Tools*) erstellt und dann über eine Schnittstelle dem Workflow Enacting Service übergeben. Dafür ist eine standardisierte Sprache zur Beschreibung des Prozesses notwendig [Hol04]. Deshalb wurde eine Prozessbeschreibungssprache auf Basis von XML eingeführt mit dem Namen *XML Process Definition Language* (XPDL) [Wor05b] bzw. Business Process Execution Language (BPEL) [ACD+03, Oas07]. Um das WFMS zu verwalten und zu kontrollieren existieren *Administration und Monitoring Tools*, die dann über eine weitere Schnittstelle an den Enacting Service angeschlossen sind. Weiterhin existieren spezielle Anwendungen, die Benutzern ihre Arbeitslisten präsentieren [LR00]. Diese Anwendungen sind die sogenannten Workflow-Clients. Weiterhin existieren in Workflows Aktivitäten, die ohne Hilfe von Benutzern ausgeführt werden können. Dafür werden Anwendungen direkt (*Invoked Applications*) aufgerufen. Für die verteilte Ausführung von Workflows ist es nötig, dass *Workflow Enacting Services* miteinander kommunizieren.

Workflows bzw. Geschäftsprozesse basieren häufig auf der Verarbeitung von Dokumenten. Aus diesem Grund ist es von Vorteil, dass das WFMS eng mit Dokumentenmanagementsystemen (DMS) integriert ist [MR95]. Ein DMS verwaltet die Dokumente (z.B. Auftragsbestätigung), die an oder vom Unternehmen gesendet werden, sowie Dokumente für die interne Kommunikation.

2.6.3 Workflow-Beschreibungen

Der Ablauf eines Workflows wird hauptsächlich durch seinen Kontroll- und Datenfluss beschrieben, auf den in diesem Abschnitt eingegangen werden soll. Um den Kontroll- und Datenfluss zu beschreiben verwenden die WFMS Workflow-Beschreibungssprachen, die meist vom jeweiligen Produkt abhängig sind. Darauf soll hier nicht weiter einge-

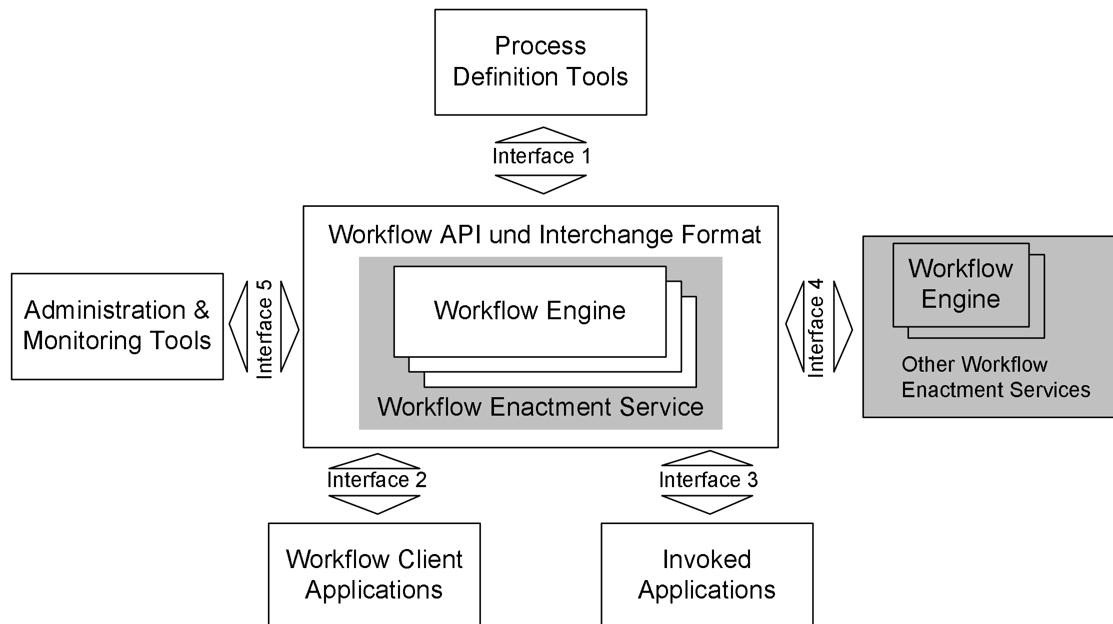


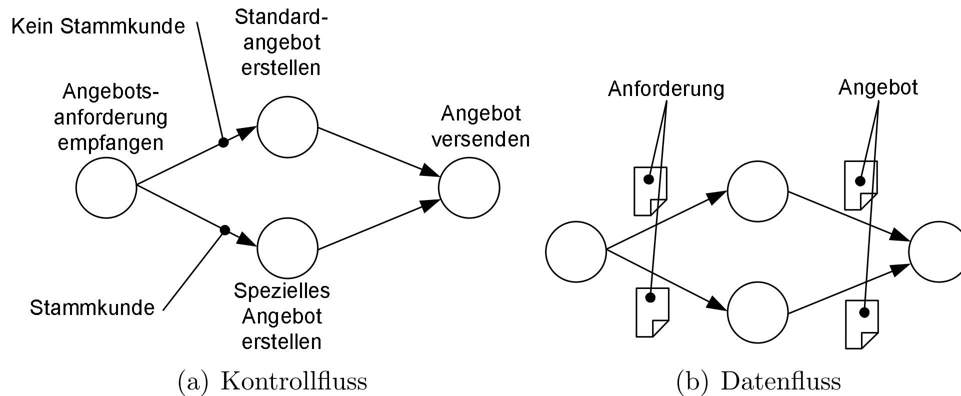
Abbildung 2.8: Referenzarchitektur nach WFMC [Wor05a]

gangen werden.

Der Kontrollfluss (Abbildung 2.9(a)) beschreibt den Ablauf der Tasks (Kreise) und damit ihre zeitliche Reihenfolge. Der Pfeil gibt dabei die zeitliche Ordnung an und bedeutet, dass der Task an der Spitze nach der Beendigung des Tasks am Start des Pfeiles ausgeführt wird. Es besteht weiterhin die Möglichkeit bestimmte Aktivitäten parallel auszuführen und damit die Workflow-Ausführung zu beschleunigen. Für eine Parallelisierung gehen dabei von einer Aktivität mehrere Pfeile aus, was als *Fork* bezeichnet wird. Treffen die parallelen Zweige in einer Aktivität zusammen, so wird das als *Join* bezeichnet. Die Zweige können *selektiv* ausgeführt werden, so wie in der Abbildung 2.9(a), bei der die Zweige nach dem Auswerten des Stammkunden-Attributs ausgeführt werden.

Der Datenfluss beschreibt dagegen nicht den Ablauf von Task-Ausführungen, sondern den Fluss der Daten zwischen den Tasks, so wie in Abbildung 2.9 (b) dargestellt. Die Pfeile beschreiben, dass die Daten, die durch einen Task erzeugt werden (am Start des Pfeiles) von der an der Pfeilspitze liegenden Task benötigt werden. Im Beispiel der Abbildung existieren zweierlei Dokumente, welche ein Anforderungs- und ein Angebotsdokument enthalten. Allerdings muss vom WFMS sichergestellt werden, dass die Daten vor der Ausführung der Aktivitäten vorhanden sind.

Mehr zur Kontroll- und Datenflussmodellierung steht in [LR00]. Es gibt auch noch andere Ansätze zur Modellierung, die dann Aktivitäten-, Kommunikations-, Zustands- oder Artifact-Modellierung heißen [CHR98].



(a) Kontrollfluss (b) Datenfluss

Abbildung 2.9: Kontroll- und Datenfluss eines Workflows[LR00]

2.6.4 Datenintegration mit Workflows

Die Workflow-Technologie wird hauptsächlich für die Integration von Funktionen verschiedenster Anwendungssysteme eingesetzt. In der letzten Zeit realisiert diese Technologie auch die Integration zwischen Prozessen, d.h. Prozesse kommunizieren untereinander. In diesem Zusammenhang wird der Begriff *Orchestration* verwendet. Um diese Anforderung zu realisieren, werden die Workflow-Systeme stark mit Messaging-Systemen und anderen Kommunikationsarten gekoppelt.

In Abschnitt 2.3 wurde die konsistenzerhaltende Datenintegration eingeführt, mit der erreicht werden soll, dass sämtliche Unternehmensdaten auf dem gleichen Stand sind. In diesem Abschnitt soll nun untersucht werden, welche Ansätze es zur konsistenzerhaltenden Datenintegration mit der Workflow-Technologie gibt.

Ein WFMS bietet viele Konzepte zur Implementierung von Geschäftsprozessen. Viele davon werden für die konsistenzerhaltende Datenintegration nicht benötigt, denn es handelt sich bei den Prozessen um kurzlebige Prozesse, die keine Benutzerinteraktionen haben. Diese werden auch als *Microflows* bezeichnet [LR00, LR02, KKL+04]. Microflows sind Workflows, die nicht unterbrechbar sind, d.h. sie unterstützen kein Vorwärts-Recovery [Ley96]. Vorwärts-Recovery sagt aus, dass nach einem Absturz der Workflow am letzten Task vorgesetzt wird. Microflows hingegen arbeiten innerhalb einer Transaktion und bei einem Absturz wird der gesamte Flow zurückgesetzt und dann neu gestartet. Die nicht unterbrechbaren Workflows eignen sich auch deshalb für die konsistenzerhaltende Datenintegration, da in dieser Art von Prozessen keine menschliche Interaktion benötigt wird und die einzige Aufgabe darin besteht bei einer Änderung eines Objektes alle Systeme, die ebenfalls Daten des Objektes speichern, über diese Änderung zu informieren. In [LR02] wurde untersucht wie Workflows für die Informationsintegration verwendet werden können. Diese Art von Integration fasst die Funktions- und Datenintegration zusammen.

Ein weiteres Beispiel der Datenintegration mittels Workflow-Technologie findet man bei der Integration mehrerer Informationssysteme im Bereich von Kliniken [JLM+05]. Dafür wurden die Prozesse der Klinik in Datenlogistik-Prozesse überführt. Bei den letzteren steht der Transport und die Transformation von Daten im Vordergrund. Sie ist

ein Ansatz zur Integration einer heterogenen Datenlandschaft, die ebenfalls Workflow-Technologien verwendet und spezialisiert ist auf die Integration von Klein- und mittelständischen Unternehmen (KMU) mit einem Großunternehmen [SGB02]. Hier wird ein Internet-Dateisystem verwendet, um Dateien mit dem KMU auszutauschen, die dann Workflows beim Großunternehmen anstoßen. Diese Workflows übernehmen die weitere Verarbeitung der Daten. Die gesamte Infrastruktur befindet sich auf den Rechnern des Großunternehmens, so dass die KMU finanziell nicht belastet werden.

2.7 XML Technologien

Die *eXtensible Markup Language* (XML) [BPSM+06] ist eine Metasprache, mit der andere Sprachen definiert werden können. Diese Sprachen können aktiv sein, wie zum Beispiel XSLT oder eine Sprache zum Speichern bzw. Austausch von Daten. XML ist sowohl von Computersystemen als auch vom Menschen lesbar. Dadurch können Daten, die zwischen einzelnen Informationssystemen ausgetauscht werden, leicht analysiert werden, mit der Möglichkeit der menschlichen Intervention. Außerdem enthält ein XML-Dokument nicht nur die benötigten Daten, sondern auch Struktur- und Elementinformationen. Aus diesem Grund kommen immer wieder die Begriffe selbstbeschreibend und semi-strukturiert im Zusammenhang mit XML zur Sprache. Einer der Nachteile von XML ist, dass außer dem String-Datentyp keinerlei Datentypen unterstützt werden. Um dennoch andere Datentypen zu verwenden, müssen diese serialisiert werden und können mittels XML Schema überprüft werden.

Weitere XML-Technologien:

XML Schema [FW04, TBMM04, BM04] ist eine Sprache mit der Schemabeschreibungen definiert werden können. Damit kann die Struktur eines XML Dokumentes festgelegt und den Elementen und Attributen Datentypen zugeordnet werden.

XPath [BBC+07] wird verwendet um einzelne Elemente, Attribute oder Inhalte aus einem XML Dokument zu extrahieren. Außerdem können mit XPath Berechnungen und Bool'sche Bedingungen definiert werden.

XSLT (*eXtensible Stylesheet Language Transformation*) [Kay07] ist eine Sprache mit der XML Dokumente transformiert werden können. Als Output der Transformation können wiederum XML Dokumente entstehen, aber auch HTML-Seiten bzw. Textdokumente. Das Prinzip eines solchen Stylesheets ist die Definition einer Menge von Transformationsregeln.

XQuery [BCF+07] ist eine Anfragesprache für XML, mit der Daten aus einer XML-Datenbank oder XML-Dokumenten gefiltert und transformiert werden können. Aus diesem Grund eignet sich XQuery wie XSLT für die Transformation von XML Dokumenten. Wie in [Kep04, Kep02] gezeigt wird, sind XQuery als auch XSLT Turing vollständig und damit auch gleich mächtig.

SOAP [Mit03, GHM+03a, GHM+03b], früher bekannt als *Simple Object Access Protocol*, ist ein Protokoll, mit dem Dokumente ausgetauscht werden können oder es

kann ein *Remote Procedure Call* realisiert werden. SOAP ist Grundlage für die Realisierung von *Web Services* [ACKM04].

2.8 Message Oriented Middleware

Die *Message Oriented Middleware* (MOM) ist eine Infrastruktur, die den asynchronen Austausch von Nachrichten ermöglicht [RMB01, Kel02]. Der Austausch kann dabei persistent oder transient durchgeführt werden und ggf. auch transaktional. Außerdem ermöglicht MOM eine lose Kopplung der Systeme. Ein weiterer Vorteil der MOM ist, dass der Empfänger beim Senden und der Sender beim Empfang nicht verfügbar sein müssen. Des Weiteren wird durch die Entkopplung und den expliziten Empfang von Nachrichten garantiert, dass der Empfänger nicht überlastet wird, da dieser die Nachrichten nur verarbeiten muss, wenn er derzeit dazu in der Lage ist. Allerdings kann bei einer dauerhaften Überlast die Warteschlange (engl. *Queue*) volllaufen. Die Nachrichten können frei definiert werden. Besonders interessant ist die Verwendung von XML als Nachrichtenformat [Kel02]. Dies ist begründet durch die Flexibilität von XML und dem selbstbeschreibenden Charakter. Die Message Oriented Middleware hat folgende Entwurfsziele [Cum02]:

Store and Forward. Die Nachricht wird von der Message Oriented Middleware entgegen genommen und solange vorgehalten, bis der Empfänger sie entgegen nehmen kann. Außerdem wird der Sender bis zum Empfang der Nachricht nicht blockiert (asynchron).

Message Broker. Der *Message Broker* ermöglicht flexiblere Kommunikationsmodelle wie zum Beispiel *Publish-Subscribe*.

Garantiertes Versenden. Es soll sichergestellt werden, dass jede Nachricht genau einmal verarbeitet wird. Dies wird unter anderem durch Transaktionen unterstützt.

Nachrichtenreihenfolge. Die Reihenfolge der Nachrichten von einer Quelle soll erhalten bleiben, wenn diese von einem Zielsystem gelesen werden.

Symbolisches Routen. Die Transportmedien sollen anhand von symbolischen Namen identifiziert werden.

Request-Response. Es soll möglich sein, dass in einer Anfrage (*Request*) der Empfänger der Antwort (*Response*) angegeben werden kann. Dadurch soll erreicht werden, dass ein Server die Antwort an die richtige Adresse senden kann.

Nachrichtentransformation. Da bei mehreren Empfängern einer Nachricht dessen Nachrichtenformatbedürfnisse auseinander gehen können, soll es die Möglichkeit geben, diese Nachrichten zu transformieren.

Adhoc Empfänger. Die meisten Empfänger von Nachrichten sind eher statisch, da sie immer die gleichen bestimmten Geschäftsfunktionen erfüllen müssen. Manche

Empfänger brauchen eine flexiblere Gestaltung von Nachrichtenabos. Sie brauchen so genannte Adhoc-Abos. Solche Adhoc-Abos werden nur für bestimmte Zeit benötigt, wie zum Beispiel ein Performanzmonitor, der aktuelle Aktivitäten darstellt.

Ausnahmeauflösung. Die Message Oriented Middleware soll dafür sorgen, dass möglichst viele Ausnahmen aufgelöst werden, so dass der Anwendungsentwickler möglichst wenige davon behandeln muss, ohne dabei die Anwendungsintegrität zu verletzten.

Standards. Die Verwendung von Standards, wie zum Beispiel JMS (Java Message Service), ermöglicht die leichte Austauschbarkeit von MOM-Produkten.

Dateitransfer. Manchmal ist es notwendig nicht nur die relativ kleinen Nachrichten zu transportieren, sondern auch relativ große Dateien, wie zum Beispiel Grafiken oder CAD-Zeichnungen. Dies könnte zum Beispiel über FTP erfolgen, wobei die URL mit einer Nachricht verschickt wird.

Man kann das Versenden von Nachrichten mit einer MOM noch dahin unterscheiden, ob das Kommunikationsparadigma eine Punkt-zu-Punkt-Kommunikation oder *Publish-Subscribe* ist.

Die Punkt-zu-Punkt-Kommunikation [Ley99] hat immer einen Sender und einen Empfänger pro Nachricht. Das Medium mit dem die Nachrichten ausgetauscht werden, wird wie bereits erwähnt Warteschlangen genannt. Bei unterschiedlichen Nachrichten können allerdings unterschiedliche Sender und Empfänger bei derselben Warteschlange existieren.

Das Publish-Subscribe-Paradigma [Ley99] verwendet *Message Broker* um seine Funktionalität zu realisieren. Dieser Ansatz ist in Abbildung 2.10 dargestellt. Im Gegensatz zur Punkt-zu-Punkt-Kommunikation können hier mehrere Empfänger pro Nachricht existieren, aber auch kein Empfänger. Ein Sender einer Nachricht wird Publisher genannt und dieser hat kein Wissen über mögliche Empfänger. Ein Empfänger wird Subscriber genannt. Dieser bekundet sein Interesse an bestimmten Nachrichten in dem er Subscriptions anmeldet. Subscriptions sind bestimmt durch das Transportmedium (Topic) und eventuellen Filterregeln.

Der *Java Message Service* (JMS) [HBS+02a] bietet eine einheitliche Schnittstelle zwischen Java und MOM-Produkten. Eine JMS-Nachricht besteht aus einem Nachrichtenkopf (Header), Nachrichteneigenschaften (Properties) und dem Nachrichtenkörper (Body). In JMS wird ebenfalls zwischen Punkt-zu-Punkt und Publish-Subscribe unterschieden. Sowohl für Punkt-zu-Punkt als auch Publish-Subscribe können Filter anhand von Header-Elementen und Properties erfolgen. JMS bietet allerdings keine Möglichkeit um die Transportmedien (Queues und Topics) zu erzeugen. Diese müssen über das MOM-Produkt angelegt werden. Um an die Basisobjekte von JMS zu kommen (Queues, Topic und die Connection Factories) wird der Java-Namensdienst JNDI verwendet. Dadurch wird eine Entkopplung von Implementierungsobjekten und Schnittstellen erreicht.

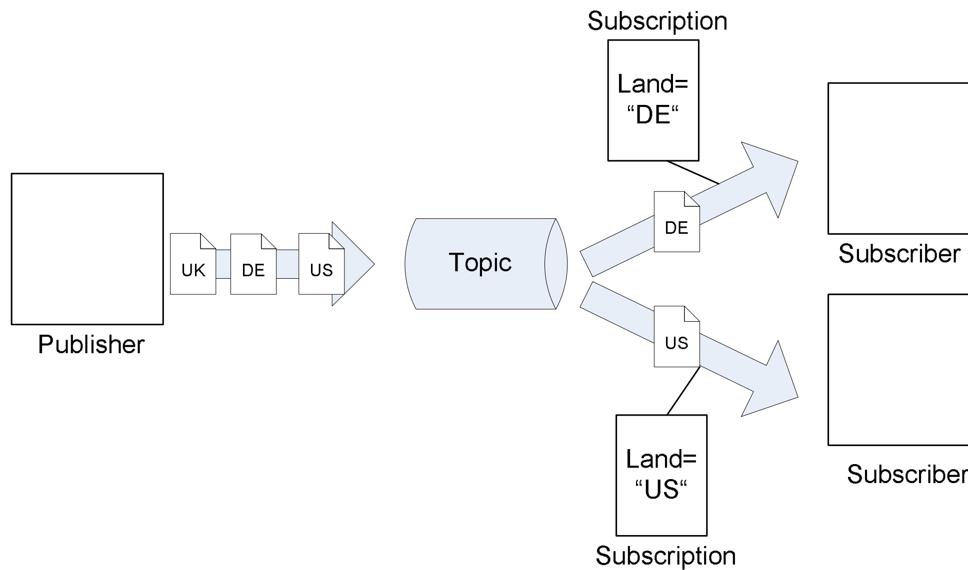


Abbildung 2.10: Publish-Subscribe

2.9 Ereignissysteme

Die Idee hinter Ereignissystemen (*Event Systems*) ist, dass ein Objekt auf die Zustandsänderungen eines anderen Objekts reagieren kann [CDK01]. Objekte die Ereignisse repräsentieren werden üblicherweise *Notifications* genannt. Ereignisse können mittels *Push* und *Pull* übertragen werden. Beim Push implementiert der Empfänger ein Interface mit dem er die Nachrichten empfängt, die das sendende Objekt verschickt. Bei Pull fragt der Empfänger Ereignisse ab, die nach einem bestimmten Zeitpunkt auftraten. Schwachstellen von Ereignissystemen sind, dass der Zeitraum vom Senden bis zum Empfang lang dauern kann und die Nachrichten in einer unterschiedlichen Reihenfolge beim Empfänger ankommen können [Mic01].

Verteilte Ereignissysteme könnte man nun einsetzen, um ein Propagationssystem zu realisieren, das Änderungen in der Anwendungsschicht mitteilt. Dies begründet sich unter anderem darin, dass Anwendungsschichten oft objektorientiert realisiert sind und diese Objekte Änderungen an denselben über ein Ereignissystem an andere Informationssysteme mitteilen könnten. Dieser Ansatz erfordert eine starke Anpassung der Anwendungsschicht. Diese Anpassung ist nicht immer möglich und gewollt.

Es besteht eine gewisse Verwandtschaft zwischen Ereignissystemen und dem Publish-Subscribe-Paradigma. Publish-Subscribe-Nachrichten können als Ereignisse angesehen werden [Cum02]. Daher verwenden Ereignissysteme das Publish-Subscribe-Paradigma [CDK01].

JINI ist ein *Framework* um verteilte Anwendungen zu erstellen. JINI definiert in seinem Standard [Mic01, CDK01] die Möglichkeit, verteilte Ereignisse auszutauschen. Dafür stellt der Standard eine Reihe von Schnittstellen und Klassen bereit. Die Kommunikation erfolgt über *Remote Method Invocation* (RMI). Allerdings definiert JINI keine Infrastruktur, die für die Verteilung der Ereignisse eingesetzt wird. Um dies zu ermöglichen, können sogenannte *Third Party Objects* eingesetzt werden.

Außerdem gibt es Ereignissysteme für die CORBA-Umgebung [Zah99]: Der *Event Service* [Gro04a] und der *Notification Service* [Gro04b]. Zuerst wurde der Event Service entwickelt und darauf aufbauend der Notification Service. Der Event Service hat zwei Modi für das Verbreiten von Ereignissen: Push und Pull. Beim Pull implementiert das bereitstellende Objekt eine Schnittstelle, mit dem die Ereignisse angefordert werden können. Der Konsument implementiert beim Push eine Schnittstelle mit dem er gesendete Ereignisse empfangen kann. Es kann zwischen bereitstellendem Objekt und Konsument ein Ereigniskanal zwischengeschaltet werden, der mehrere Ereignisbereitsteller und Konsumenten erlaubt. Außerdem kann mit einem Ereigniskanal Push und Pull kombiniert werden [Gro04a]. Der *Notification Service* ist eine Erweiterung des *Event Service*. Es ermöglicht typisierte Ereignisse. Konsumenten können Ereignisse filtern und nach Ereignistypen fragen. Ereignisquellen können sich nach gewünschten Ereignissen erkundigen. Außerdem ist es möglich, die Eigenschaften von Kanälen festzulegen, wie zum Beispiel FIFO.

2.10 Model-Management

2.10.1 Übersicht

Insbesondere Integrationsaufgaben sind metadaten-intensiv. Es müssen Modelle der einzelnen Informationssysteme verwaltet werden. Darauf aufbauend müssen Änderungsnachrichten definiert und Transformationen zwischen diesen Nachrichten erstellt werden. Um diese Aufgaben zu vereinfachen, können *Model-Managementsysteme* eingesetzt werden.

Model-Managementsysteme werden entwickelt, um Aufgaben, die im Zusammenhang mit der Verwaltung von Metadaten stehen zu vereinfachen [BHP00a, BHP00b]. Unter Metadaten wird beispielsweise ein Datenbankschema, ein ER-Modell, XML Schema oder ein UML-Modell verstanden. Zur vereinfachten Handhabung werden diese Metadaten durch einen Import vereinheitlicht, was generalisierte Operatoren ermöglicht. Veränderte Modelle können dann wieder in eine spezielle Darstellungsform exportiert werden. Modelle werden in Form von Graphen abgelegt, wobei die Knoten einzelne Modellelemente (z.B. Klassen, Attribute) und Kanten Beziehungen darstellen. Es gibt noch eine spezielle Art von Beziehungen, die eine Zuordnungsbeziehung darstellt, d.h. eine Klasse ist beispielsweise in einem Modell enthalten oder ein Attribut ist einer Klasse zugeordnet. Diese Art von Beziehung bildet einen azyklischen gerichteten Graphen. Zwei weitere wichtige Konstrukte in einem Model-Managementsystem sind *Mapping* und *Morphismen* [MRB03]. Mapping und Morphismen sind selbst auch Modelle [BHP00a, BHP00b], die Verbindungen zwischen zwei Modellen darstellen und damit Korrespondenzen zwischen den Modellen beschreiben. Ein Mapping enthält dabei auch Funktionen zum Übergang und ein Morphism nur die Relation zwischen mehreren Modellelementen.

Um einen Mehrwert zur Verwaltung der Metadaten zu bieten, stellen Model-Managementsysteme generische Operatoren bereit. Diese Operatoren haben als Eingabeparameter und als Ausgabeparameter Modelle, Mappings bzw. Morphismen. Die Operato-

ren lassen sich in zwei Gruppen aufteilen: Die einfachen und die komplexen Operatoren. Zu den einfachen gehören zum Beispiel das Erzeugen und Verändern eines Modells. Zu den komplexen gehören die folgenden:

Match Automatic Schema Matching (nächster Abschnitt).

Diff Das Finden der Unterschiede zwischen zwei Modellen anhand eines Morphismus [MRB03].

Merge Die Erstellung eines integrierten Modells aus zwei Modellen [PB03].

Mapping Composition Das Integrieren zweier Mappings.

2.10.2 Automatic Schema Matching

Mit *Automatic Schema Matching* werden Korrespondenzen zwischen Schemata gefunden [RB01]. Diese Korrespondenzen können als Grundlage für die Erstellung von Transformationen verwendet werden. Dies kann als unabhängiges System erfolgen oder innerhalb eines Model-Managementsystems als Match-Operator. Um Korrespondenzen zwischen den einzelnen Schemata zu finden, existieren unterschiedliche Methoden, die unterschiedliche Kriterien für das Erkennen von Korrespondenzen verwenden. Diese teilen sich auf in *Schema-Matchers*, *Instance-Matcher*, *Hybrid-Matcher* oder *Composite-Matchers*. Schema-Matchers nehmen als Grundlage die Schema-Definition. Beim Instance-Matching werden dagegen die vorhandenen Daten als Grundlage verwendet. Der Hybrid-Matcher verwendet mehrere Ansätze innerhalb eines einzigen Matchers. Im Gegensatz dazu ruft der Composite-Matcher mehrere unabhängige Matcher auf, deren Ergebnisse er dann kombiniert. Beim Schema-Matcher kann noch unterschieden werden, ob einzelne Elemente gematched werden (auf Grundlage von Namen, Beschreibungen oder Datentypen) oder die Struktur zu Rate gezogen wird.

Als Systeme (Forschungsprototypen) lassen sich hier Cupid [MBR01], Coma [DR02] und Protoplasm [BMPQ04] aufführen. Mit Coma lassen sich durch einen festgeschriebenen Prozess mehrere Matcher kombinieren. Dieser Prozess besteht aus Benutzerinteraktion (Beurteilung des Ergebnis und Auswahl von Matcher), Ausführen der Matcher und Gesamtergebnis berechnen. Gegebenenfalls kann der Prozess wiederholt werden. Der Protoplasm Prototyp verwendet dagegen frei definierbare Prozesse, so dass der Benutzer sich nicht mehr um die Auswahl und Reihenfolge der Matcher kümmern muss.

2.11 Schlussfolgerungen

Nachdem einige Technologien betrachtet wurden, die für eine Änderungspropagation verwendet werden bzw. als Grundlage dienen können, sollen diese bewertet werden.

Die ERP-Systeme lösen eine Vielzahl von Integrationsproblemen in einem Unternehmen. Allerdings decken sie nur einen Teilbereich der Softwareanwendungen ab, die in einem Unternehmen benötigt werden. Des Weiteren sind auf dem Markt vorhandene ERP-Systeme nicht für jedes Unternehmen geeignet oder es gibt Probleme mit

der Einführung. Aus diesen Gründen wird im Unternehmen weiterhin Integration von Daten benötigt.

Die EAI stellt eine umfassende Technologie dar, die eine Vielzahl von Produkten zum Vorschein gebracht hat. Die wichtigsten dieser Produkte sollen im Abschnitt 5.3 (Vergleich mit verwandten Ansätzen) untersucht werden. Wenn wir noch einmal einen Blick auf die EAI-Architekturen von Tabelle 2.1 werfen, so lassen sich Point-to-Point und Verteilte Objekte als Grundlage ausschließen. Point-to-Point hat zu viele Verbindungen zwischen den einzelnen Informationssystemen und damit auch einen zu hohen Erstellungs- und Wartungsaufwand. Verteilte Objekte dienen eher dazu, um Methoden, die von Objekten bereitgestellt werden, für andere Prozesse anzubieten. Die Bus-Architektur bietet eine gute Grundlage für ein Propagationssystem, ermöglicht allerdings kein inhaltsbasiertes Verteilen von Nachrichten [Pap06] und hat außerdem ein Problem beim Anpassen von Änderungsnachrichten nach den Anforderungen eines Zielsystems. Aus diesen Gründen wird als Basis eine Hub-and-Spoke-Architektur gewählt, die diese Nachteile beseitigt. Allerdings muss angemerkt werden, dass der Hub zu einem Flaschenhals werden kann.

Eine unternehmensweite zentrale Datenbank kann die Integrationsprobleme auch nicht lösen. Um eine solche zentrale Datenbank zu entwerfen, ist ein immenser Entwicklungsaufwand notwendig. Außerdem wird keine Datenlokalität und Autonomie realisiert. Des Weiteren stellt auch dieser Ansatz einen Flaschenhals dar. Diese Probleme können mittels Verteilter Datenbanken und Replikation gelöst werden. Allerdings muss weiterhin ein kostenintensives zentrales Modell entwickelt werden. Des Weiteren befindet sich die Integration bei der datenbankbasierten Replikation in der Datenschicht, d.h. evtl. vorhandene Konsistenzregeln in der Anwendungsschicht werden umgangen. Föderierte Datenbanken bringen unterschiedlichste Datenquellen auf ein gemeinsames Modell, beschäftigen sich aber weniger mit der konsistenzerhaltenden Datenintegration.

Peer-Data-Management-Systeme ermöglichen flexible Netze, mit denen auch *Update Propagation* möglich ist. Auch diese Systeme basieren auf der Datenschicht und Konsistenzregeln werden in der Anwendungsschicht umgangen. Ein weiterer Nachteil in der Unternehmensumgebung ist der meist dynamische Charakter eines solchen Systems. Im Speziellen muss hier auf die *Open World Assumption* hingewiesen werden.

Als Grundlage für ein Änderungspropagationssystem sind Prozesse geeignet, da sie eine flexible Gestaltung von Propagationsaufgaben ermöglichen. Diese könnten zum Beispiel mit Microflows realisiert werden, aber nicht mit Rückwärts-Recovery-basierten Workflows. Allerdings hat die Workflow-Technologie noch keine standardisierten Tasks, die für die Änderungspropagation benötigt werden. Des Weiteren werden mit Workflowsystemen die Reihenfolgeeinhaltung von Änderungen nicht garantiert, da die Prozesszeiten stark voneinander abweichen können. Bei reinem Einsatz von Microflows wäre es möglich, müsste aber zusätzlich realisiert werden. Ein weiterer Schwachpunkt ist die Erkennung und Auflösung von Änderungskonflikten.

Eine Änderungspropagation sollte Änderungen von Geschäftsobjekten (z.B. Kundenauftrag) propagieren und nicht die von Implementierungsobjekten (z.B. Auftragskopf und Auftragspositionen), da durch Geschäftsobjekte ein hoher Zusammenhang zwischen den Implementierungsobjekten besteht. Außerdem soll das Einpflegen der

Daten wenn möglich auf der Ebene der Anwendungsschicht liegen, damit evtl. vorhandene Konsistenzregeln in dieser Schicht nicht umgangen werden.

XML bietet durch seine flexible Gestaltung und der Möglichkeit von menschlicher Ausnahmebehandlung bei Fehlern eine gute Grundlage um Änderungen zu beschreiben. Des Weiteren bietet XML eine Vielzahl von weiteren Technologien und darauf aufbauenden Produkten, die in einem Propagationssystem verwendet werden können. Durch seine hierarchische Struktur lassen sich die Daten von Geschäftsobjekten darstellen, da Geschäftsobjekte intern ebenfalls eine hierarchische Struktur haben.

Eine weitere viel versprechende Technologie ist die *Message-oriented Middleware* (MOM). Sie ermöglicht das sichere Übertragen von Änderungsnachrichten unter der Einhaltung der Reihenfolge.

Verwandt damit sind Ereignissysteme, die ebenfalls eingesetzt werden könnten (z.B. CORBA Notification Service). Zum Aufbau einer Hub-and-Spoke-Architektur eignen sich aber MOM-Systeme besser, da sie Point-to-Point-Kommunikationen ermöglichen, mit deren Hilfe die „Speichen“ (engl. Spokes) realisiert werden können. Außerdem erfordert die Verwendung von Ereignissystemen eine Anpassung der Anwendungsschicht des Informationssystems.

Das Model-Management ist eine Technologie, mit deren Hilfe die Modelle der einzelnen Informationssysteme verwaltet werden können und Transformationen zwischen einzelnen Änderungsformaten semi-automatisch entwickelt werden könnten. Diese Systeme werden in der vorliegenden Arbeit jedoch nicht genauer betrachtet.

KAPITEL 3

Grundlegende Konzeption

Nachdem Technologien und Konzepte zur Integration von Informationssystemen diskutiert und bewertet wurden, soll in diesem Kapitel das Konzept für das Änderungspropagationssystem im heterogenen Umfeld entwickelt werden. Die Integration soll auf Basis von Propagationen von Geschäftsobjektänderungen erfolgen, da dies ein geringeres Austauschvolumen als der vollständige Datenaustausch hat. Diese Art von Propagation wird auch bei der Replikation von Datenbanken eingesetzt. Aus diesem Grund werden zuerst die verschiedenen Replikationsvarianten untersucht. Dann werden die Basiskonzepte für die heterogene Propagation und je eine Sprache zur Definition von komplexen Abhängigkeiten sowie eine zur Definition von Bedingungen für Bool'sche Ausdrücke auf Änderungen vorgestellt. Anschließend werden die Komponenten des Propagationssystems beschrieben. Weitere wichtige Eigenschaften eines solchen Propagationssystems sind die Erkennung und Behandlung von Änderungskonflikten und die Einhaltung der Änderungsreihenfolge. Die Anbindung der Informationssysteme erfolgt mit Adaptern, was eine flexible Kapselung aus Sicht des Propagationssystems ermöglicht. Der Einsatz von Adaptern ermöglicht eine geringe bis keine Anpassung im zu integrierenden Informationssystem und dient als Bindeglied zwischen dem Informationssystem und dem Propagationssystem. Diese Eigenschaften und der Adapter des Propagationssystems werden in den Unterkapiteln 3.8 - 3.11 diskutiert.

3.1 Lösung für Replikation der Informationssystemdaten

Um ein Propagationssystem für heterogene und autonome Informationssysteme zu entwerfen, werden zuerst Lösungen für homogene Systeme nach passenden Konzepten untersucht, die als Grundlage für den heterogenen Fall geeignet sind. Informationssysteme haben im Gegensatz zur herkömmlichen Replikation (vgl. Abschnitt 2.4.1.3)

	Lazy Replication	Eager Replication
Master	1 Objektbesitzer N Transaktionen	1 Objektbesitzer 1 Transaktion
Group	N Objektbesitzer N Transaktionen	N Objektbesitzer 1 Transaktion

Abbildung 3.1: Replikationsstrategien nach [GHOS96] und ihre Verwendung für die Integration von Informationssystemen

keine explizit replizierten Daten. Das bedeutet, dass die Replikation nicht durch einen Administrator definiert wurde und die Daten auf einzelne Rechner verteilt wurden. Die Replikation zwischen den Informationssystemen entsteht über die Zeit und die einzelnen Systeme sind unabhängig voneinander. Die Gründe hierfür liegen in dem Datenbedarf einzelner Unternehmensbereiche, welche die Informationssysteme verwenden, und den Kauf und Entwicklung neuer Informationssysteme, die dann Daten redundant zu anderen Systemen verwalten. Die Datenbedürfnisse der einzelnen Informationssysteme überschneiden sich teilweise und daher entsteht eine Art von Replikation. Durch die unterschiedlichen Aufgaben und Abteilungen, die für die Informationssysteme verantwortlich sind, entsteht eine heterogene Systemlandschaft. Außerdem wachsen Informationssysteme mit der Zeit, d.h. neue Daten werden benötigt und neue Funktionalitäten müssen implementiert werden. Um eine Lösung für das Replikationsproblem zwischen den Informationssystemen zu konzipieren, muss sowohl die Heterogenität als auch die Autonomie der Informationssysteme berücksichtigt werden. Die Autonomie begründet sich hauptsächlich durch die Eigenverantwortung der einzelnen Abteilungen. Diese sollen weitestgehend unabhängig operieren können.

Als Erstes werden Lösungen im Bereich der replizierten Datenbanken genauer untersucht. Laut [GHOS96] lassen sich die Replikationslösungen in Gruppen unterteilen, die in Abbildung 3.1 dargestellt sind. Diese unterscheiden sich darin, ob alle Änderungen innerhalb einer Transaktion durchgeführt werden (*Eager Replication*) oder ob sie in unabhängigen Transaktionen für jedes System durchgeführt werden (*Lazy Replication*). Des Weiteren kann man unterscheiden, ob es einen Objektbesitzer gibt, d.h. ein änderbares Objekt (Master) oder ob es mehrere Objektbesitzer gibt, d.h. eine von Objektgruppe (Group) von der jedes beliebige Objekt geändert werden kann. *Eager Replikation* garantiert die *1-Kopien-Serialisierbarkeit* (*One Copy Serializability*) [BG82, BG83]. Allerdings müssen bei einer Änderung alle Knoten verfügbar sein. Außerdem kann man noch unterscheiden, ob Änderungen nur an einer Stelle durchgeführt werden können (*Update-Master*) oder in jedem Knoten (*Update-Anywhere*). *Update-Master* in Zusammenhang mit *Eager Replication* verhindert Deadlocks und mit *Lazy Replication* Änderungskonflikte.

Informationssysteme sollen unabhängig von anderen Informationssystemen sein (autonom) und sie sollen ebenfalls kein Wissen über andere Informationssysteme haben, mit denen sie integriert sind. Aus diesem Grund können Änderungen, die mehrere In-

formationssysteme betreffen, nicht innerhalb einer Transaktion durchgeführt werden, d.h. es wird hier eine Variante der Lazy Replikation verwendet. Daraus folgt aber auch, dass so genannte veraltete Informationen (*Stale Information*) in Kauf genommen werden müssen, was aber durch eine zeitnahe Weiterleitung von Änderungen verringert werden kann. Werden Informationssysteme zwischen unterschiedlichen Geschäftsbereichen integriert, so handelt es sich hauptsächlich um eine *Update-Master* Variante, da jedes der Geschäftsbereiche einen bestimmten Aufgabenbereich hat (vgl. z.B. Taylor). Allerdings sollte nicht ausgeschlossen werden, dass das Replikationssystem auch innerhalb eines Geschäftsbereichs eingesetzt wird oder die klassische Aufgabenverteilung nicht angewendet wird, wie in neueren Unternehmensansätzen. Ein Beispiel hierfür ist die Anpassung von Kundendaten durch einen Produktionsleiter, da der Kunde aus anderen Gründen gerade mit ihm telefoniert. Weiterhin kann man bei Replikationssystemen noch unterscheiden, ob Änderungen bzw. geänderte Daten mittels *Push* oder *Pull* [SS05] übertragen werden. Bei Pull kann noch unterschieden werden, ob die Aktion benutzergesteuert (manuell), periodisch oder On-Demand (sobald auf ein Objekt zugegriffen wird) ausgeführt wird. Benutzergesteuertes und periodisches Pull führt dazu, dass die Informationen stärker veraltet sind und die Konfliktwahrscheinlichkeit steigt. Aber auch die On-Demand-Pull-Variante ist nicht besser als die Push-Variante, wenn die Daten hauptsächlich gelesen werden, da in diesem Fall häufig die Aktualität überprüft werden muss. Außerdem muss bei der On-Demand-Pull-Variante ein Mechanismus implementiert werden, welcher die Erkennung des aktuellen Geschäftsobjektes ermöglicht. Dies kann zum Beispiel durch einen Zeitstempel erreicht werden, was allerdings zu einer Erweiterung der Informationssysteme führt, die aber bei nicht allen Informationssystemen möglich ist. Außerdem müssen interne Zugriffe der lesenden Art erkannt werden und an das Integrationssystem weitergeleitet werden. Dies führt zu einer weiteren Anpassung der Informationssysteme. Aus diesem Grund wird in dieser Arbeit die Push-Variante verwendet.

Bei Push kann noch unterschieden werden, ob Änderungen nach einer Transaktion (*deferred*) oder sofort (*immediate*) weitergeleitet werden [PS00]. In der *eager*-Variante kann es nur im Zusammenhang mit *immediate* verwendet werden, während *lazy* mit beiden verwendet werden kann. Die *Immediate*-Variante erhöht die Datenaktualität. Allerdings werden in den anderen Informationssystemen Daten sichtbar, die noch nicht *committed* wurden, sofern diese Änderungen nicht in den anderen Informationssystemen ebenfalls in einer Transaktion ausgeführt werden. Es wurde für die hier entwickelte Propagationslösung die *Deferred*-Variante gewählt, da Adapter[1] die Veränderungen im Informationssystem erkennen, darauf angewiesen sind, dass die Änderungen sichtbar sind, d.h. die Transaktion erfolgreich abgeschlossen wurde. Außerdem müssen keine Transaktionen unterstützt werden, die identisch im Quell- und Zielsystem sind.

Des Weiteren kann man unterscheiden, ob Zustände oder Operationen weitergeleitet werden [SS05]. In einer Lösung für homogene Systeme kann man durch Übermittlung von Operationen das Traffic-Aufkommen reduzieren, da eine Operation viele Datensätze betreffen kann. Beispiel hierfür ist ein `INSERT INTO`, bei der ein `SELECT`

[1]Ein Adapter ist eine Komponente, die als Bindeglied zwischen einem Informationssystem und dem hier konzipierten Propagationssystem dient.

als Input verwendet wird. Problematisch wird dies aber im heterogenen Fall, denn die Input-Tabelle muss im Zielsystem nicht unbedingt vorhanden sein. Aus diesem Grund eignen sich hier keine Mengenoperationen sondern nur Satzoperationen. Allerdings ist auch hier die Transformation schwieriger als bei der Transformation von Zuständen. Als Beispiel hierfür kann die Transformation von Vorname und Nachname zu Name genannt werden; wenn eine Änderungsoperation nur den Nachnamen ändert, d.h. der Vorname ist in der Änderungsoperation nicht vorhanden, wird aber für die Transformation benötigt. Das Erkennen von Operationen auf der hier angedachten Geschäftsobjektebene ist ebenfalls sehr schwierig und erfordert eine Anpassung der beteiligten Informationssysteme. Eine solche Anpassung sollte allerdings bei diesem Ansatz vermieden werden.

Nachdem die erste Auswahl an Konzepten statt gefunden hat, müssen weitere Basiskonzepte entwickelt werden, die spezifisch für das hier angestrebte System sind, aber durchaus auch in homogenen Fällen zum Einsatz kommen könnten.

3.2 Basiskonzepte

Um die Daten der Informationssysteme konsistent zu halten, sind die Konzepte der Abhängigkeit und ihre Ausführungsinstanzen, die Propagationsprozesse sowie die Beschreibung einer Änderung wichtig. Diese werden in den folgenden Unterabschnitten genauer untersucht.

3.2.1 Abhängigkeiten und Propagationsprozesse

Bevor Änderungen eines Geschäftsobjektes in einem System an die davon betroffenen Geschäftsobjekte in anderen Systemen weitergeleitet werden können, müssen Beziehungen zwischen den jeweiligen Geschäftsobjekttypen definiert werden. Für die Beziehungen wurde der Begriff und das Konzept *Abhängigkeit* eingeführt. Eine Abhängigkeit ist eine gerichtete Beziehung zwischen einem Quellobjekt in einem Quellsystem zu einem oder mehreren Zielobjekten in Zielsystemen. Um eine Konsistenz der Informationssysteme zu gewährleisten, müssen bei einer Änderung des Quellobjektes die Daten des entsprechenden Zielobjektes ebenfalls angepasst werden. Anzumerken ist, dass die Daten und die Datenstruktur eines Quellobjektes nicht notwendigerweise eins zu eins mit denen des Zielobjektes übereinstimmen müssen. Dies ist in der Heterogenität der Informationssysteme begründet. Änderungsbeschreibungen, die die geänderten Daten eines Geschäftsobjektes beschreiben, müssen deshalb an die Anforderungen des Zielobjektes angepasst werden, d.h. die Änderungsbeschreibung muss unter Umständen transformiert werden.

In Abbildung 3.2 sind Abhängigkeiten zwischen drei Informationssystemen (Quadrate) und deren Geschäftsobjekttypen (Kreise, siehe Abschnitt 2.1.3) dargestellt. Wie in der Abbildung zu sehen ist, müssen nicht alle Geschäftsobjekttypen Quelle oder Ziel einer Abhängigkeit sein. Eine Schwierigkeit entsteht, wenn ein Geschäftsobjekttyp Ziel von zwei Abhängigkeiten ist. In diesem Fall muss sichergestellt werden, dass die von den

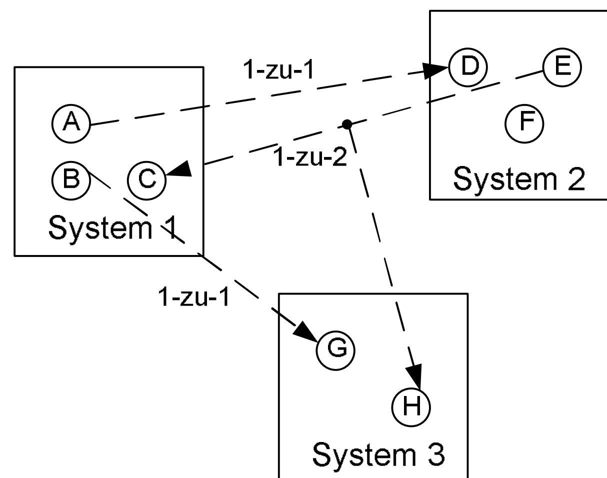


Abbildung 3.2: Beispiel von Abhängigkeiten zwischen Informationssystemen und Geschäftsobjekttypen

Abhängigkeiten betroffenen Teilmengen der Geschäftsobjekte disjunkt sind. Andernfalls ist es schwierig zu entscheiden, welche Änderungen den Vorrang bekommen soll. Dadurch kann es zu nicht deterministischen Verhalten kommen, denn die Quelle der Änderung ist nicht mehr eindeutig. Außerdem ist aus der Abbildung ersichtlich, dass eine Abhängigkeit mehrere Zielgeschäftsobjekte haben kann, die in unterschiedlichen Informationssystemen liegen können. Diese werden 1-zu-N-Abhängigkeiten genannt, wobei die Eins für die Anzahl von Quellen und N für die Anzahl von Zielen steht. In diesem Kapitel ist die Anzahl der Quellen auf eins beschränkt. Sollen mehrere Quellen verarbeitet werden, ist das grundsätzlich möglich und wird im nächsten Kapitel behandelt.

Um eine Integration der Informationssysteme zu ermöglichen, werden Änderungen entlang der definierten Abhängigkeiten, wie in Abbildung 3.2 dargestellt, propagiert. Die Abhängigkeiten können deshalb als Pfade der Änderungen angesehen werden, wobei die Abhängigkeit ein Design-Time-Charakter und der Pfad einen Runtime-Charakter hat.

Die Realisierung eines solchen Pfades erfolgt durch ein weiteres Konzept, den so genannten Propagationsprozessen. Dieser stellt eine logische Einheit für die Propagation von Änderungen dar. Wie bereits diskutiert, muss es sich bei einer Abhängigkeit nicht um eine 1-zu-1-Beziehung handeln, sondern sie kann mehrere Ziele haben. Dies wirkt sich ebenfalls auf den Propagationsprozess aus. Eine 1-zu-N-Abhängigkeit kann auch als n 1-zu-1-Abhängigkeiten realisiert werden, hat dann aber eine andere Semantik im Fehlerfall. Dadurch geht jedoch der logische Zusammenhalt verloren, denn es werden in diesem Fall n Propagationsprozesse ausgeführt. Außerdem sollte ein Propagationsprozess als Microflow (Abschnitt 2.6.4 und 2.11) ausgeführt werden. Dies sollte innerhalb

einer Transaktionssphäre[2] geschehen, so dass fehlerhafte Prozesse wiederholt werden können.

3.2.2 Änderungsbeschreibung

Damit Änderungen, die in einem Geschäftsobjekt aufgetreten sind, an andere Systeme weitergeleitet werden können, müssen sie in einer definierten Art und Weise beschrieben werden. In diesem Abschnitt wird deshalb die Beschreibung von Änderungen genauer untersucht.

Ändern sich ein oder mehrere Attribute eines Geschäftsobjektes, so ändert sich der Zustand des Geschäftsobjektes. Ein Zustand ist die Menge aller Attributwerte eines Objektes. Diese können selbst komplexe Objekte sein, wie z.B. die Adresse eines Kunden. Des Weiteren kann man die Art der Zustandsänderung unterscheiden. Es gibt grundsätzlich drei Arten wie sich der Zustand eines Objektes ändern kann: ein neues Objekt wird erzeugt (*create*), ein bestehendes Objekt wird geändert (*update*), d.h. Attributwerte ändern sich, oder ein bestehendes Objekt wird gelöscht (*delete*).

Nachdem die Grundlagen einer Änderung diskutiert wurden, kann eine *Änderungsbeschreibung* (AB) genauer untersucht werden. Eine Änderungsbeschreibung ist selbst auch ein Objekt und beschreibt die Änderung, die in einem System aufgetreten ist und vom Propagationssystem verarbeitet wird. Ein solches Objekt lässt sich durch ein Tupel beschreiben:

$$AB = (S, GT, A, B, D, TS) \tag{3.1}$$

S: *System* in dem die Änderung auftrat oder in dem sie angewendet werden soll.

GT: Typ des geänderten (zu ändernden) Geschäftsobjektes.

A: Änderungsart $A \in \{create, update, delete\}$.

B: Zustandsbeschreibung (kurz: Zustand) des Geschäftsobjektes im System S, bevor die Änderung auftrat. Dieser Zustand wird Davor-Zustand bezeichnet.

D: Zustandsbeschreibung des Geschäftsobjektes im System S nach der Änderung. Dieser Zustand wird Danach-Zustand bezeichnet.

TS: Zeitstempel der Änderung.

Während der Verarbeitung im Propagationssystem können sich alle Elemente des Tupels ändern. Das System S ändert sich vom Quellsystem auf das Zielsystem. Der Typ des Geschäftsobjektes (GT) ändert sich aufgrund von unterschiedlichen Bezeichnern im Quellsystem und Zielsystem. Der Übergang von einer Änderungsart (A) zu einer anderen ist komplexer als die des Namens und wird deshalb anhand eines Beispiels erklärt:

[2] Eine Transaktionssphäre ist ein Teilprozess, der nicht durch das übliche Store-and-Forward abgehandelt wird, sondern innerhalb einer Transaktion und damit auch ggf. als Ganzes zurückgesetzt wird.

Zustand	*create*	*update*	*delete*
Davor (B)	Null	X	X
Danach (D)	X	X	Null

Tabelle 3.1: Änderungsarten und Zustände in Änderungsbeschreibungen

Ein Produkt ist im PDM-System[3] sichtbar, da es mit dessen Hilfe entwickelt wird. Das ERP-System (vgl. Abschnitt 2.2) hat aber noch keine Informationen über das Produkt, da die Produktion noch nicht angelaufen ist. Der Zustand des Objektes ändert sich nun auf einen produktionsbereiten Zustand. Dies ist eine Änderung im PDM und wird deshalb auch als *update* an das Propagationssystem gesendet. Da das Objekt noch nicht im ERP-System existiert, muss es angelegt werden und A muss sich auf *create* ändern.

Die Verwendung der Zustandsbeschreibungen B und D sind abhängig von der Änderungsart. Dies ist in Tabelle 3.1 dargestellt.

Die beiden Zustände (D u. B) ändern sich anhand von Transformationen, wobei eine Transformation immer auf beide Zustände angewandt wird. Transformationen werden benötigt, da die Daten der Systeme semantisch oder strukturell heterogen sind [Her03].

Um eine Änderungsbeschreibung vollständig zu definieren, sollte sie zwei Zustände haben, da bei einem Update eines Geschäftsobjektes mit mehreren Implementierungsobjekten (z.B. Kundenauftrag, der aus einem Auftragskopf und mehreren Auftragspositionen besteht), müssen die Unteränderungsarten (*create, update, delete*) erkannt werden, um die entsprechenden Änderungen der Implementierungsobjekte durchzuführen. In dem Beispiel der Kundenauftragsänderung können Auftragspositionen geändert werden, wegfallen oder hinzukommen. Außerdem können durch die Verwendung von zwei Zuständen Änderungsdeltas im Propagationssystem berechnet werden.

Die Änderung kann anstatt mit zwei Zuständen auch mit einem Zustand und einem Änderungsdelta Δ beschrieben werden. Allerdings ist die Verarbeitung im Propagationssystem schwierig, da für das Änderungsdelta ein anderes Schema als für die Zustände benötigt wird. Zum Beispiel für $B = \{betrag = 10\}$ und $D = \{betrag = 30\}$ ergibt sich ein Änderungsdelta $\Delta = \{betrag {+}{=} 20\}$. Wird die gleiche Beschreibungsart (nur Zustände) verwendet, können auch einheitliche Schemas (Zustandsvalidierung) und Transformationen eingesetzt werden. Außerdem können Zustände leichter beschrieben werden. Für Änderungsdeltas werden für Zahlen Addition und Subtraktion sowie für Strings Anfüge- und Ausschneide-Operationen benötigt. Dies ist deutlich komplexer als die Beschreibung durch Zustände.

3.3 Transaktionen

Nachdem die ersten Grundlagen eines Propagationssystems im heterogen Umfeld besprochen wurden, soll in diesem Abschnitt noch die Rolle von Transaktionen in diesem

[3]Product Data Management dient zur Verwaltung von Produktdaten während der Entwicklungsphase.

Umfeld besprochen werden. In [GHOS96] wird beschrieben, dass Transaktionen im homogenen Fall, die im Quellsystem ausgeführt werden, auch so in den anderen Replikaten durchgeführt werden müssen. Die Frage stellt sich, ob das auf den heterogenem Fall übertragen werden kann.

Werden im heterogenen Fall zwei Informationssysteme miteinander integriert, erledigen diese in den meisten Fällen unterschiedliche Aufgaben. Daraus lässt sich schließen, dass die geänderten Geschäftsobjekte in unterschiedlichen Transaktionen verarbeitet werden. Ein Beispiel ist ein BDE-System (Betriebsdatenerfassung), bei dem ein Fertigungsauftrag durch das Eintreffen einer Meldung auf fertig markiert wird und dieser mit der Endzeit versehen wird. Dagegen müssen im angebundenen Finanzbuchhaltungssystem auf dieses Ereignis hin Buchungen im Halbfertigprodukt-Konto und Fertigprodukt-Konto ausgeführt werden.

In vielen Fällen müssen deshalb keine Transaktionsinformationen mit versendet werden. Da wir hier nur den heterogenen Fall betrachten wollen, werden keine Transaktionsinformationen propagiert.

3.4 XML als Basis für Änderungspropagation

In diesem Abschnitt wird untersucht, welche Rolle XML für den Einsatz im Propagationssystem spielen kann. XML wird dabei verwendet um flexible Zustandsbeschreibungen der Geschäftsobjekte zu ermöglichen, die in einer Änderungsbeschreibung verpackt sind. Darauf aufbauend können Technologien eingesetzt werden, die die Verarbeitung dieser Zustände ermöglichen.

3.4.1 XML zur Definition von Zustandsbeschreibungen

Die *eXtensible Markup Language*, kurz XML, eignet sich besonders gut für die Definition der Zustände aus den folgenden Gründen. XML stellt eine Sprache bereit, mit der gleichzeitig Daten sowie deren Bedeutung beschrieben werden. Deshalb werden XML-Dokumente auch als selbstbeschreibend bezeichnet. Dies führt dazu, dass Daten vom Menschen lesbar sind und Fehler schneller entdeckt werden können. Ein weiterer wichtiger Vorteil ist, dass durch die Standardisierung und weltweite Akzeptanz von XML eine Reihe weiterer Standards und Technologien entwickelt wurden. So kann die Struktur eines XML-Dokumentes durch XML Schema festgelegt und mit den entsprechenden Parsern kontrolliert werden. Bedingungen können durch XPath-Ausdrücke definiert werden und für Transformationen können Transformationssprachen wie XQuery[4] und XSLT verwendet werden.

XML wird dabei für die Beschreibung der Zustände B und D einer Änderungsbeschreibung verwendet, denn durch die Verwendung von XML als Sprache für die Zustände, ist die Bedeutung der geänderten Daten definiert. Durch die Überprüfung der Struktur kann festgestellt werden, ob es sich um ein gewisses Geschäftsobjekt handelt. Wie schon in Abschnitt 2.1.3 erwähnt, kann ein Geschäftsobjekt aus weiteren

[4] XQuery wird als Anfragesprache für XML-Daten verwendet

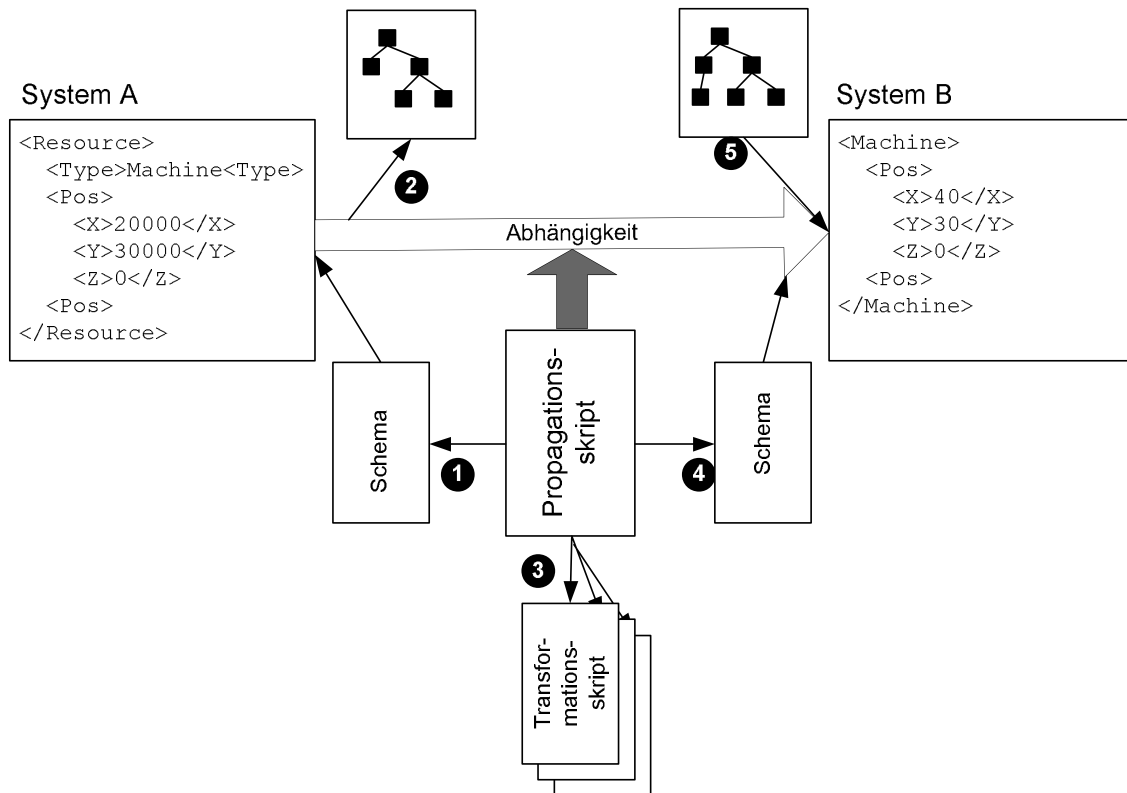


Abbildung 3.3: Das Zusammenspiel von Technologien zur Änderungspropagation

Objekten bestehen, die durch eine Komposition verbunden sind. Eine Komposition ist eine strenge hierarchische Beziehung zwischen Objekten. Dies bedeutet aber auch, dass ein Geschäftsobjekt durch die hierarchische Struktur von XML dargestellt werden kann (vgl. [Dau03]).

3.4.2 Technologie für eine XML-basierte Änderungspropagation

Im vorigen Abschnitt wurde hervorgehoben, dass die Vielzahl von verfügbaren XML-Standards und XML-Werkzeugen, einen großen Vorteil für den Einsatz von XML darstellt. In diesem Abschnitt werden deshalb weitere Standards für den Einsatz der Änderungspropagation untersucht. Als Beispiel soll hier eine neue Maschine (Ressource) von System A zu System B propagiert werden (vgl. Abbildung 3.3). Die Schwierigkeit liegt darin, dass die Koordinatensysteme einen unterschiedlichen Ursprung (x ist um 20m verschoben) und unterschiedliche Einheiten (mm und m) haben. Außerdem muss noch die Struktur der Daten angepasst werden.

In Abbildung 3.3 sind zunächst die grundsätzlich benötigten Technologien für eine Änderungspropagation dargestellt. Die Abhängigkeit ist durch ein Propagationsskript beschrieben, das weitere Technologien verwendet, um seine Aufgabe zu erledigen. Als Erstes muss überprüft werden (1), ob die eingegangene Änderung überhaupt dem ent-

spricht, was man erwartet. Dafür wird ein Schema verwendet, welches Strukturinformationen enthält. Im XML-Fall kann man dafür XML Schema verwenden, das zum einen mächtiger als DTD (z.B. Datentypen) ist und zum anderen in XML beschrieben ist. Kritiker von XML Schema merken oft an, dass XML Schema zu kompliziert ist (z.B. Vorwort von [Dau03]). Es existieren aber eine Reihe von Editoren (z.B. XML Spy von Altova), mit deren Hilfe man einfach XML Schemas erstellen kann.

Wie schon oben erwähnt, muss bei der Beispielspropagation eine Koordinatentranslation und eine Strukturänderung vorgenommen werden. In (3) werden deswegen Transformationsskripte aufgerufen. Dabei kann es sich um ein Transformationsskript handeln oder um mehrere Skripte, die eine schrittweise Transformation ermöglichen. Im XML-Fall hat man grundsätzlich die Wahl zwischen XSLT oder XQuery. In unserem System wurden beide Möglichkeiten durch die Verwendung der Saxon-Engine realisiert. Eine Diskussion über die Mächtigkeit von XSLT und XQuery ist in [BMN02, Kep02] zu finden und soll hier nicht weiter vertieft werden.

Gegebenenfalls kann am Ende noch die Zieländerungsbeschreibung nach ihrer Struktur überprüft werden (4). Dies ist aber nicht immer notwendig, denn wenn die Eingangsänderungsbeschreibung und die Transformationen korrekt sind, muss auch die Zieländerungsbeschreibung korrekt sein. Die Überprüfung ist eine Zeitfrage und ein Weglassen kann somit die Performanz steigern. Solange man aber in einer Entwicklungsphase ist, ist die Überprüfung für den Integrationsprogrammierer eine Hilfe zum Feststellen der Korrektheit von Transformationen und wird deshalb optional unterstützt.

Was aus der Abbildung nicht ersichtlich ist, ist die Definition von Filterregeln und bedingten Ausführungen. Für diesen Zweck wird eine Sprache benötigt, die es ermöglicht, Bedingungen auf den Änderungsanforderungen zu definieren. In XML gibt es dafür eine mächtige Sprache, die auch von XQuery und XSLT verwendet wird: XPath. Diese ist aber für die zustandsübergreifenden Regeln nicht ausreichend. Deswegen wird im Abschnitt 3.6.1 eine Sprache dafür eingeführt.

Um die Performanz der Ausführung eines Propagationsskriptes zu steigern, besonders wenn viele Zwischenschritte, wie Transformationen ausführen oder Bedingungen überprüfen, enthalten sind, empfiehlt es sich, besonders bei einem serialisierten Format wie XML, ein internes Format (2) zu verwenden. Dieses wird dann bis zum Schluss verwendet, bis die Änderungsanforderung wieder ins XML-Format umgewandelt wird (5). Wird wie hier XML als externes Format verwendet, eignet sich DOM gut für die interne Repräsentation, da dieses direkt von entsprechenden Werkzeugen unterstützt wird. DOM [HHW+04] ist die objekt-orientierte Darstellung eines XML-Dokumentes.

3.5 Sprache für die Definition von Abhängigkeiten

Wie in Abschnitt 3.2 beschrieben, ist die Abhängigkeit das Basiskonzept einer Änderungspropagation und beschreibt durch Pfeile die Integrationsbeziehungen zwischen den Geschäftsobjekten. Allerdings ist diese Beschreibung durch die Verwendung von Pfeilen und möglichen Attributen, die diesen zugeordnet sind, nicht mächtig genug, da die Gestaltung durch diese Art eingeschränkt ist, z.B. ist es nicht möglich mehrere Transformationsskripte zu verwenden, oder paralleler und sequentieller Verarbeitung

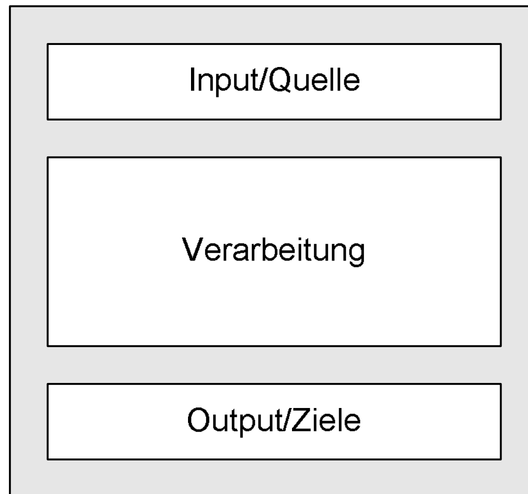


Abbildung 3.4: Aufbau eines Propagationsskriptes

zu definieren. Aus diesem Grund führen wir ein weiteres Konzept ein: Propagationsskripte (vgl. auch Abbildung 3.3). Diese beschreiben im Detail, wie eine Abhängigkeit definiert ist.

Durch die Einführung von Propagationsskripten wird auch die Beziehung zwischen Abhängigkeiten und Propagationsprozessen klarer. Denn ein Propagationsprozess ist die Ausführung eines Propagationsskriptes und damit auch einer Abhängigkeit. Im Rahmen dieser Arbeit wurde die Sprache XML Propagation Definition Language (XPDL) zur Definition der Propagationsskripte entwickelt. XPDL ist eine Sprache, die als Basis eine für Propagationsprozesse angepasste und erweiterte Version der Workflow-Sprache XRL [vdAVK01, VHvdA02, vdAK03] hat. In früheren Veröffentlichungen wurde XPDL selbst XRL [Ker01, CHRM02, CHRM03] und XRL+ [RCHM02] genannt.

In Abbildung 3.4 ist der grundsätzliche Aufbau eines Propagationsskriptes dargestellt. Wie in Abschnitt 3.2 beschrieben, gibt es ein Quellgeschäftsobjekt, in dem die Datenänderung auftrat. Aus diesem Grund braucht man einen Teil im Propagationsskript, der definiert, in welchem System und in welchem Geschäftsobjekt eine Änderung auftrat. Angedeutet wurde auch schon, dass im heterogenen Fall oftmals Transformationen notwendig sein können. Diese und andere Verarbeitungsschritte können dann im Verarbeitungsteil definiert werden. Schließlich hat jede Abhängigkeit mindestens ein Zielsystem und Zielgeschäftsobjekt, das dann im Output-Teil definiert wird.

Ein Propagationsskript hat also Befehle um den Input zu deklarieren. Für den Verarbeitungsteil gibt es Befehle, die den Kontrollfluss steuern und Befehle, die die Änderungsbeschreibungen verarbeiten. Weiterhin braucht man spezielle Output-Befehle, welche die Änderungsbeschreibungen an die Systeme überreichen. Die Befehle werden in drei Gruppen unterteilt: Deklaration der Eingabe, Kontrollfluss und Verarbeitungs- sowie Ausgabebefehle.

3.5.1 Deklaration der Eingabe

In XPDL wird am Anfang eines Propagationsskriptes mit `input_declaration` die Spezifikation der zu empfangenden Änderungsbeschreibung eingeleitet. Dafür wird innerhalb der Input-Deklaration ein Element mit dem Namen `start_input` definiert. Dadurch wird sichergestellt, dass beim Auftreten einer definierten Änderung im Quellsystem und Weiterleitung an das Propagationssystem als Änderungsbeschreibung das entsprechende Propagationsskript gestartet wird.

```
start_input(system (S), GO_Typ (GT), out, expression?)
```

Die Änderungsbeschreibungen, die zu einem Start des Propagationsskriptes führen, können anhand des Quellsystems S und dem Geschäftsobjekttypen GT (`GO_Typ`) ausgewählt werden. Ausdrücke (`expression`) ermöglichen die weiteren Einschränkungen der qualifizierenden Änderungsbeschreibungen anhand der Inhalte der Zustände B und D und der Änderungsart A. Die Ausdrücke werden in einer speziellen, auf XPath-basierenden Sprache abgefasst (siehe Abschnitt 3.6). Dadurch können Änderungsbeschreibungen vor der Verarbeitung gefiltert werden. Beispielsweise können dadurch nur Kundenaufträge, die ein bestimmtes Volumen haben, an ein bestimmtes Zielsystem gesendet werden.

Wird eine Änderungsbeschreibung empfangen, so wird diese gegen die Struktur des Geschäftsobjektes anhand eines XML Schemas überprüft.

Um die empfangene Änderungsbeschreibung weiterverarbeiten zu können, wird ihr ein interner Name (`out`) gegeben, der eindeutig sein muss und der dann als Input für andere Befehle verwendet werden kann, um beispielsweise eine Transformation mit dieser Änderungsbeschreibung durchzuführen, die die Änderungsbeschreibung als Input nimmt und dabei den Namen als Referenz für die Änderungsbeschreibung verwendet.

3.5.2 Kontrollfluss

Ein Propagationsprozess kann durch seinen Kontroll- und Datenfluss beschrieben werden, was einem Workflow ähnlich ist. Der Datenfluss beschreibt, wie eine Änderungsbeschreibung verarbeitet wird, bis sie schließlich an ein Zielsystem propagiert wird. Gesteuert wird der Datenfluss dabei durch den Input und Output der einzelnen Befehle, die die Datenänderungen verarbeiten. Der Kontrollfluss steuert im Gegensatz dazu, welcher Befehl als nächstes ausgeführt werden soll, welche parallel ausgeführt werden können oder welche überhaupt ausgeführt werden sollen. Der Aufbau eines Propagationsskriptes ähnelt dem eines Workflow (siehe Abschnitt 2.6). Aus diesem Grund werden einige Workflow-Sprachen und deren Kontrollflusssteuerungen in diesem Abschnitt genauer untersucht. Um dann die Kontrollflusselemente für XPDL zu definieren.

Um eine Grundlage für die Kontrollflusssteuerung in XPDL zu haben, wird die Steuerung in bestehenden Workflow-Sprachen untersucht. Dabei wurden moderne Workflow-Sprachen ausgewählt. Diese sind entweder in der Forschung entstanden, wie die *eXchangeable Routing Language* XRL [vdAVK01, VHvdA02, vdAK03], oder sind Bestandteil von Industrie-Produkten, wie XLang von Microsoft, das von BizTalk [Tat01]

Gruppe	sequentielle Ausführung	parallele Ausführung	bedingte Ausführung
XRL [vdAVK01, VHvdA02, vdAK03]	• sequence • any_sequence • beschränkte Auswahl[1]	• parallel_sync • parallel_no_sync • parallel_part_sync	• condition • while_do
XLang[Tat01]	sequence	all	• switch • while • pick
BPEL[ACD+03, Oas07]	sequence	flow	• switch (1.1) • if (2.0) • while • repeatUntil • pick • forEach

Tabelle 3.2: Steuerung des Kontrollflusses in gängigen Workflow-Sprachen

verwendet wird. Weiterhin wurde ein Industriestandard untersucht, der zur Koordination von Web-Services dient, die sogenannte *Business Process Execution Language for Web Services* BPEL4WS [ACD+03].

Die Befehle für die Kontrollflusssteuerungen wurden in drei Gruppen unterteilt: die sequentielle, parallele und bedingte Ausführung, wie in Tabelle 3.2 zu sehen ist. Befehle, die in XPDL (*XML Propagation Definition Language*) den Kontrollfluss eines Propagationsprozesses steuern, werden entsprechend der Gruppe in den nachfolgenden Abschnitten untersucht.

3.5.2.1 Sequentielle Ausführung

Befehle, die sequentiell ausgeführt werden sollen, werden nacheinander und in der angegebenen Reihenfolge ausgeführt. Die untersuchten Workflow-Sprachen unterstützen alle diese einfache Ausführungsart. Allerdings unterstützt XRL auch noch eine Ausführungsart, bei der die Befehle in beliebiger Reihenfolge ausgeführt werden können, d.h. die Sequenz A, B, C kann zum Beispiel in der Reihenfolge B, C, A ausgeführt werden. Weiterhin wird eine sequentielle Ausführung vorgeschlagen, bei der nur ein Teil ausgeführt werden muss [KZ02]. Dies bedeutet bei einer 2-von-3-Ausführung, dass beim obigen Beispiel die Sequenz B, C ausreichend ist.

Eine Propagationssprache und damit auch XPDL muss auf jeden Fall die Standardsequenz unterstützen, da Transformationen und Propagationen in einer bestimmten Reihenfolge ausgeführt werden müssen. Eine wahlfreie Ausführung wird dagegen seltener gebraucht und nur, wenn es sich um unabhängige Propagationsteilprozesse handelt. Diese können allerdings aufgrund ihrer Unabhängigkeit und des Fehlens menschlicher

[1] Vorgeschlagen in [KZ02]

Bearbeiter parallelisiert werden. Aus diesem Grund ist es unnötig eine wahlfreie und beschränkte Ausführungsart für die Propagationssprache anzubieten.

Der `sequence`-Befehl hat keine Argumente und muss deshalb nicht gesondert erklärt werden. Die Unterelemente beschreiben die Zweige, die in der angegebenen Reihenfolge ausgeführt werden sollen.

3.5.2.2 Parallele Ausführung

Bei der parallelen Ausführungsart werden die Befehle nebenläufig ausgeführt. Dabei gibt es wieder die einfache Form, bei der die nachfolgenden Befehle erst ausgeführt werden, wenn alle parallelen Zweige beendet wurden. Dies kann aufgelockert werden, so dass die Ausführung der nachfolgenden Befehle sofort (`parallel_no_sync`) oder nach einer angegeben Anzahl beendeter Zweige (`parallel_ part_sync`) fortgesetzt werden.

Wie im sequentiellen Fall wird die einfache Art des parallelen Ausführens in der Propagationssprache XPDL unterstützt. Die anderen beiden von XRL angebotenen Modi sind dabei weniger nützlich, da die nachfolgenden Befehle im Regelfall auf den Ergebnissen der parallelen Zweige aufbauen. In einigen wenigen Fällen können die nachfolgenden Befehle unabhängig sein und eine sofortige Fortsetzung ermöglichen. In solch einem Fall können die nachfolgenden Befehle in einem weiteren parallelen Zweig ausgeführt werden und damit die obigen Ausführungsarten für die Propagationssprache XPDL überflüssig machen.

Der `parallel`-Befehl hat keine Argumente und muss deshalb nicht gesondert erklärt werden. Die Unterelemente beschreiben die Zweige, die parallel ausgeführt werden sollen.

3.5.2.3 Bedingte Ausführung

Die bedingte Ausführung von Befehlen lässt sich in zwei Gruppen unterteilen. Bei der ersten Gruppe werden die Zweigelemente nicht iterativ sondern nur einmal ausgeführt: `condition`, `if` und `switch`, während bei der zweiten Gruppe der Zweig iterativ, d.h. solange eine Bedingung gültig ist, ausgeführt wird (`while`, `repeatUntil`, `forEach`).

Der `condition`-Befehl steuert die Ausführung der untergeordneten Befehle abhängig von der Auswertung einer Bedingung. Wird die Bedingung als wahr ausgewertet, so wird der `true`-Zweig ausgeführt, anderenfalls der `false`-Zweig. Der `switch`-Befehl dagegen besteht aus mindestens einem `case`-Zweig, der eine Bedingung definiert, welche die Ausführung seiner Zweigbefehle steuert. Dabei wird nur der erste als wahr evaluierte Zweig ausgeführt [Tat01]. Dadurch können bestimmte Transformationen oder Propagationen in Abhängigkeit von Bedingungen auf bestimmten Feldern der Änderungsbeschreibung ausgeführt werden.

Für die Verwendung in einer Propagationsumgebung wird der `while`-Befehl nicht benötigt, da eine variable Anzahl von Propagationen an das gleiche System eher unwahrscheinlich ist. Das einzige Szenario, das vom `while`-Befehl bearbeitet werden kann, ist eine Zerstückelung eines Geschäftsobjektes in seine Bestandteile. Ein Beispiel hierfür wäre eine Aufteilung der einzelnen Positionen eines Kundenauftrages. Wie man aber erkennen muss, kann das Geschäftsobjekt in seine Implementierungsobjekte auch im

Zielsystem aufgeteilt werden. Im Zielsystem bzw. seinen Adapter wird die Aufgabe erledigt, die Geschäftsobjekte der logischen Ebene auf die Datenobjekte der Implementierungsebene zu transformieren. Dieses Verfahren ist deshalb so wichtig, da Geschäftsobjekte einen logischen Zusammenhang haben und deshalb auch atomar im Zielsystem geändert werden sollten.

Der `condition`-Befehl oder der `switch`-Befehl wird dagegen benötigt, um bedingte Propagationen bzw. Transformationen durchzuführen. Ein Beispiel hierfür wäre die Propagation von einer Fertigungsmaschine in Abhängigkeit der Veränderung des Verfügbarkeitsstatus. Dadurch kann erreicht werden, dass nicht verfügbare Maschinen an ein bestimmtes System gemeldet werden, beispielsweise an ein Fabrikcockpit.

```
condition(in, expression, truePart, falsePart)
```

Der `condition`-Befehl hat als Input eine Änderung, dessen Namen mit `in` angegeben wird. Der Ausdruck, der auf der Nachricht ausgewertet werden soll, wird mit `expression` angegeben. Der `truePart` wird ausgeführt, wenn der boolesche Ausdruck als wahr evaluiert wird und der `falsePart`, wenn nicht. Der boolesche Ausdruck soll wie beim Befehl `start_change` die Möglichkeit geben, Ausdrücke auf bestimmten Änderungszuständen (B und D) zu definieren.

```
switch(in, case+, default?)

case(expression)
```

Der Befehl `switch` besteht aus einer Inputdeklaration und mindestens einer `case`-Deklaration und einem optionalen `default`-Teil, der aufgerufen wird, sofern keine `case`-Deklaration zur Ausführung ausgeführt wird. Ähnlich wie zu XLang [Tat01] wird jeweils nur die erste als wahr validierte `case`-Deklaration ausgeführt.

3.5.3 Verarbeitungs- und Output-Befehle

In diesem Abschnitt werden nun die Befehle behandelt, mit denen Änderungsbeschreibungen transformiert, gefiltert und propagiert werden können.

3.5.3.1 Transform-Befehl

Wie schon mehrmals hervorgehoben, wurde die Propagationslösung für die Integration heterogener Informationssysteme konzipiert. Die Sprache XPDL soll es ermöglichen, Änderungsbeschreibungen, genauer gesagt deren Zustandsbeschreibungen (B und D), an die Geschäftsobjekte im Zielsystem anzupassen. Durch diese Anpassung benötigt das Zielsystem keine Kenntnisse über den strukturellen Aufbau bzw. die semantischen Unterschiede zu den Geschäftsobjekten des Quellsystems. Eine Propagationssprache soll ein Konstrukt bereitstellen, mit dem Transformationen der Zustandsbeschreibungen (B und D) möglich sind. Als Beispiel für eine solche Beschreibung sei hier eine Koordinatentranslation zwischen zwei Fabriklayoutwerkzeugen genannt, die einen unterschiedlichen Koordinatenursprung haben, d.h. bei der Propagation von Fertigungsressourcen

müssen die Koordinaten entsprechend dem neuen Koordinatensystem berechnet werden.

Da XML verwendet wird, um die Zustände zu beschreiben, bietet sich die Verwendung von XSLT oder XQuery für Transformationsskripte an. Das Bereitstellen beider Varianten ermöglicht ein Entwicklungsfreiraum und durch die Verwendung von Standard-Engines bzw. Prozessoren keinen erhöhten Entwicklungsaufwand. Da es für den Entwickler eines Propagationsskriptes nicht entscheidend ist, in welcher Sprache ein Transformationsskript implementiert ist, wird dies in XPDL transparent gehalten. Das Propagationssystem hat Kenntnis wie es die unterschiedlichen Transformationsskripte verwenden muss und deshalb kann es vom Entwickler transparent gehalten werden. Dadurch kann die Sprache der Transformationsskripte gewechselt werden, ohne dass das Propagationsskript verändert werden muss. Weiterhin soll für den Entwickler verborgen werden, welcher der Zustände in der Änderungsbeschreibung vorhanden ist (z.B. D für *create*) und transformiert werden muss. Des Weiteren sollen Übergänge zwischen Änderungsarten möglich sein, wie schon in Abschnitt 3.2.2 erwähnt wurde. Dies betrifft auch den Übergang von Änderungsbeschreibungen mit zwei Zuständen zu Änderungsbeschreibungen mit einem Zustand, wofür eine Reduktion der Änderungszustände möglich sein muss. Im Gegensatz dazu ist der umgekehrte Fall nicht möglich, da nicht mehr Informationen generiert werden können als vorhanden sind. Deshalb muss bei einer Änderungsbeschreibung mit einem Zustand, die Anzahl der Zustände konstant bleiben. Der Transform-Befehl hat folgendes Aussehen:

```
transform(in, out, script, reduceTo?, parameter*)
```

Dieser Befehl hat als Input (`in`) und Output (`out`) eine Änderungsbeschreibung. Ein weiteres obligatorisches Attribut ist der Name des Transformationsskriptes (`skript`), unter dem das eigentliche Script zu finden ist. Optional ist dagegen die Reduktionsspezifikation (`reduceTo`), mit der Änderungsbeschreibungen (*update*) mit zwei Zuständen zu einem reduziert werden können. Die Reduktion erfordert ggf. den Zugriff auf den anderen Zustand, was aus dem Transformationsskript möglich sein sollte. Dies kann für Transformationen ohne Reduktion ebenfalls notwendig sein. Deshalb sollte es möglich sein aus einem Transformationsskript auch auf den jeweils anderen Zustand zuzugreifen, was durch eine XPath-Bibliothek bereitgestellt wird, siehe Abschnitt 3.6.2. Zusätzlich kann noch eine beliebige Anzahl von Parametern definiert werden, die dann im Transformationsskript verwendet werden können. Diese Parameter bestehen aus Namen-Wert-Paaren. Es kann damit zum Beispiel der Translationsvektor (vgl. Abschnitt 3.4.2) für die oben erwähnte Koordinatentransformation übergeben werden und damit kann das Transformationsskript unabhängig vom Translationsvektor bleiben. Dies erhöht zu einem gewissen Grad die Wiederverwendbarkeit von Transformationsskripten.

3.5.3.2 Propagate-Befehl

Die transformierten Änderungsbeschreibungen müssen irgendwann an ein Zielsystem propagiert werden. Dafür ist der `propagate`-Befehl zuständig. Mit diesem Befehl wird

Quell-änderungsart	Zieländerungsart		
	create	update	delete
create	O	-	-
update	X	O	X
delete	X	-	O

Tabelle 3.3: Sinnvolle Übergänge zwischen Änderungsarten [(O) kein Übergang, (X) sinnvoller Übergang, (-) nicht sinnvoller Übergang

das interne Format der Zustände der Änderungsbeschreibung (DOM) wieder serialisiert, d.h. in die Textrepräsentation von XML gebracht und schließlich wird die Änderungsbeschreibung an das Zielsystem gesendet.

```
propagate(in, system, GO_Typ, chg_type?)
```

Dieser Befehl hat als Input die durch `in` definierte Änderungsbeschreibung. Der Parameter `system` definiert das Zielsystem. Der Name des Zielsystems wird dabei intern in seine physische Adresse aufgelöst, an der das Zielsystem seine Änderungsbeschreibungen abholt. Der GO_Typ wird angegeben, damit das Zielsystem weiß, um welches Geschäftsobjekt es sich handelt. Außerdem können im Debug-Modus die Änderungsbeschreibungen daraufhin überprüft werden, ob die Änderungszustände dem Schema entsprechen. Dies muss nur in einem Debug-Modus erfolgen, da bei korrektem Input und korrekter Verarbeitung auch ein korrektes Ergebnis entsteht. Deshalb kann aus Performancegründen die Überprüfung der Ausgabe ausgeschaltet werden. Der optionale Parameter `chg_type` wird zum Überschreiben von Änderungsarten verwendet, d.h. der Übergang von einer Änderungsart in eine andere.

Beim Einsatz des Attributes `chg_type` stellt sich die Frage, welche der Überschreibungen von Änderungsarten wirklich sinnvoll sind. Tabelle 3.3 gibt darüber Aufschluss, was im Folgenden diskutiert wird. Es ist nicht sinnvoll ein *create* in eine andere Änderungsart überzuführen, denn das Objekt kann noch nicht in dem anderen System vorhanden sein. Im Gegensatz dazu kann ein *update* zu einem *create* überführt werden. Als Beispiel sei hier als Quelle ein Produktdatenmanagement-System (PDM) und als Ziel ein ERP-System angegeben. Das PDM-System verwaltet Daten über Produkte in der Entwicklung, während das ERP-System Produkte verwaltet, die Produktionsreife haben. Wird im PDM ein Produkt zur Produktreife gebracht, was ein *update* darstellt, so kann im ERP das Produkt angelegt werden (*create*). Das Gleiche gilt, wenn das Produkt ausläuft. In diesem Fall wird das Produkt gelöscht (*delete*). Dafür wird zuerst mit einer Transformation (Abschnitt 3.5.3.1) eine Reduktion des *Updates* auf einen Zustand durchgeführt. Dies wird dann durch das Propagate in eine *create*-Änderung umgewandelt. Die Überführung der Änderungsart *delete* ist durch das Ziel einer Historie-Datenbank begründet, die die gesamte Historie der Änderungen verwaltet. In diesem Fall wird eine `create` Änderungsart benötigt, um einen Eintrag in die Datenbank einzufügen.

3.5.3.3 Der Filterbefehl

Es kann durchaus vorkommen, dass Filter-Befehle notwendig sind, um gezielt Änderungsbeschreibungen anhand von Bedingungen zu filtern. Allerdings ermöglicht die `start_input`-Deklaration die Definition von Bedingungen, um eine Selektion der Eingabe zu machen. Weiterhin wurde bei den Kontrollflussbefehlen schon eine bedingte Verarbeitung eingeführt, so dass der Filter-Befehl nicht notwendig ist und zu unsauberer Programmierung und damit zu schwer erkennbaren Programmabläufen führen würde. Denn der Filter-Befehl wirkt sich im Gegensatz zu den Kontrollflussbefehlen auf den Datenfluss aus, der aus den Propagationsskripten schwer ersichtlich ist. Dies könnte zwar durch eine geeignete Darstellung der Propagationsskripte vermindert werden, doch ist das Filtern im Datenfluss der Kontrollflussregulierung durch bedingte Ausführung unterlegen. Dies ist vor allem durch die bessere Lesbarkeit des Propagationsskriptes begründet.

3.6 Pfadausdrücke für Änderungsbeschreibungen

Um bedingte Ausführungen über alle Zustände einer Änderungsbeschreibung in XPDL zu ermöglichen, wird eine Sprache benötigt, die es ermöglicht Bedingungen zu formulieren. Diese Sprache ist die *Propagation Condition Language* (PCL). Außerdem müssen in bestimmten Fällen mit Pfadausdrücken aus einer der beiden Transformationssprachen (XSLT oder XQuery) Fragmente selektiert werden, die aus anderen Zuständen oder sogar anderen Änderungsbeschreibungen stammen. Da beide Transformationssprachen XPath hierfür verwenden, wurde eine Erweiterung von XPath in Form einer Bibliothek gewählt.

3.6.1 Propagation Condition Language (PCL)

In Abschnitt 3.5 wurde eine Sprache zur Steuerung von Informationsflüssen von Quellsystemen zu Zielsystemen eingeführt. Dabei wurde festgestellt, dass bestimmte empfangene Änderungsbeschreibungen gefiltert werden müssen (`start_input`) und dass der Kontrollfluss anhand von Bedingungen reguliert werden muss (`condition` oder `switch`).

Diese Bedingungen können sich auf einen Zustand beziehen, wie zum Beispiel `/Person/Age` > `18`. Diese Art von Bedingungen kann einfach durch die Verwendung der mächtigen Pfadausdruckssprache XPath definiert werden. Um die zwei Zustände der Änderungsbeschreibung zu realisieren, könnte man diese Zustände in einem XML-Dokument codieren, so dass zwei Elemente `before` und `after` unterhalb des Wurzelknotens wären und dann XPath für die Pfadausdrücke einsetzen. Dies führt aber dazu, dass Ausdrücke schwer realisierbar wären, die beispielsweise auf den Danach-Zustand und bei Nicht-Existenz (delete-Änderungsart) auf den Davor-Zustand Bezug nehmen.

Um diesen Anforderungen gerecht zu werden, wurde die Sprache XPath erweitert, um die oben genannte Art von Ausdrücken zu ermöglichen. Dafür wurde ein von XPath nicht verwendetes Zeichen '%' als Erkennungsmerkmal der Elemente der neuen Sprache

mit dem Namen *Propagation Condition Language* (PCL) eingeführt. Um jetzt die oben genannte Bedingung zu beschreiben, wurden noch mehrere Schlüsselworte eingeführt. Bevor die Schlüsselworte aufgelistet werden, soll die Sprache anhand eines Beispiels verdeutlicht werden, bei dem die Änderung eines Namens abgeprüft wird.

(a) `%after%/Person/Nachname != %before%/Person/Nachname`

Dieser Ausdruck gibt die Bedingung auf einen geänderten Namen wieder. Der Nachname einer Person nach einer Änderung wird mit dem Nachnamen derselben Person vor der Änderung verglichen. Nach dem Beispiel mit zwei Schlüsselworten, sollen nun alle eingeführt werden.

before Dieses Schlüsselwort gibt das Wurzelelement des Davor-Zustandes (B) der aktuellen Änderung zurück. Ist dieser Zustand nicht vorhanden, so wird ein Null zurückgegeben. Dies trifft auf *create* zu.

beforeOrAfter Das Schlüsselwort ist ähnlich zu before, aber im Fall eines nicht vorhandenen Davor-Zustandes wird der Danach-Zustand (D) verwendet. Hat beispielsweise ein Kundenauftrag ein bestimmtes Volumen, so wird es an ein bestimmtes System propagiert oder beim Löschen wieder von dem bestimmten Informationssystem entfernt.

after Dieses Schlüsselwort gibt den Danach-Zustand (D) zurück, der ggf. Null sein kann.

afterOrBefore Entspricht der bevorzugten Zustandsselektion für den Danach-Zustand.

chgType Dieses Schlüsselwort kann verwendet werden, um gezielt Bedingungen auf die Änderungsart zu stellen.

timestamp Hiermit können bedingte Ausführungen oder Filterausdrücke realisiert werden, die auf ein bestimmtes Zeitschema abzielen (z.B. erste Monatshälfte).

Die weiteren Elemente der Änderungsbeschreibung (System S, Geschäftsobjekttyp GT) werden in PCL nicht benötigt, da jedes Propagationsskript speziell für Änderungsbeschreibungen mit einem bestimmten System und Geschäftsobjekttyp geschrieben ist.

Die Implementierung der *Propagation Condition Language* erfolgt, indem die oben genannten Schlüsselworte komplett in XPath übersetzt werden. Dies erfolgt durch die in XPath ermöglichte Einbindung von selbst entwickelten Funktionen. Die Funktionen sind dabei durch einen bestimmten *Namespace* gekennzeichnet. Die Realisierung der Funktionen erfolgt durch ein Objekt, welches über die aktuelle Änderung verfügt. Das oben genannte Beispiel (a) sieht übersetzt folgendermaßen aus:

```
pcl:after($pclObj)/Person/Nachname !=
pcl:before($pclObj)/Person/Nachname
```

Wie unschwer zu erkennen ist, leidet die Lesbarkeit deutlich unter der Übersetzung, deshalb wurden die speziellen Sprachelemente von PCL eingeführt.

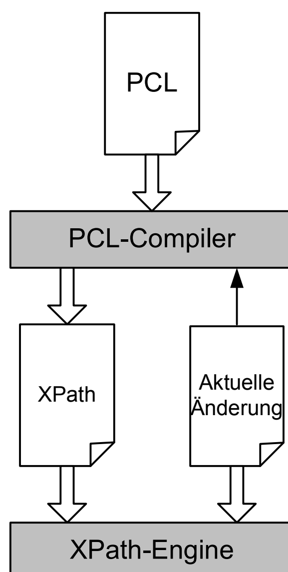


Abbildung 3.5: Die Architektur der Verarbeitung von PCL-Bedingungen

Der oben stehende XPath-Ausdruck entsteht durch Übersetzung des PCL-Ausdrucks mittels des PCL-Compilers, wie in Abbildung 3.5 dargestellt. Dieser muss Kenntnis über die aktuelle Änderung haben, muss aber nicht auf diese zu greifen. Der dabei entstehende XPath-Ausdruck kann dann von dem verwendeten XPath-Prozessor ausgeführt werden. Die dabei notwendige aktuelle Änderungsbeschreibung wird mittels des PCL-Objektes ($pclObj) übergeben.

Diese Sprache wird eingesetzt um Bedingungen innerhalb von XPDL zu definieren. Da die Sprache einen Compiler benötigt, ist dieser Ansatz nicht für den Einsatz innerhalb einer Transformationssprache geeignet. Hierfür sollten bestehende Erweiterungsmöglichkeiten verwendet werden. Dafür bietet sich eine XPath-Bibliothek an, die im folgenden Abschnitt behandelt wird.

3.6.2 XPath-Bibliothek

XPath wird ebenfalls von den Transformationsskripten verwendet, d.h. von XSLT und XQuery. Um nun erweiterte Möglichkeiten für die Transformationen bereitzustellen, wurde eine XPath-Bibliothek entwickelt, mit deren Hilfe auf Elemente des Propagationsprozesses zugegriffen werden kann. Als Beispiel sei hier die Berechnung einer relativen Verschiebung einer propagierten Fertigungsressource angegeben, die aus der alten und neuen Position in der Fabrikhalle berechnet wird. Dies kann durch eine Transformation realisiert werden, bei der auf den anderen Zustand zurückgegriffen wird, sofern es sich um einen Update handelt. Diese Art von Transformation kann mittels einer XPath-Bibliothek realisiert werden. Ähnlich wie bei der PCL-Realisierung verfügt die Bibliothek über ein Objekt, das Zugriff auf die prozessinternen Daten ermöglicht.

Es wurde hier die Form einer XPath-Erweiterung mit einer Bibliothek gewählt, anstatt diese durch XSLT-Erweiterungselemente bereitzustellen, weil XPath-Erweiterung-

en flexibler einsetzbar sind, z.B. in Pfadausdrücken oder Bedingungen. Eine spezielle Sprache wie im Falle von PCL, die XPath als Sprache erweitert, wurde deshalb nicht weiterverfolgt, da dies zu Änderungen im Kern des XSLT-Prozessors bzw. der XQuery-Engine geführt hätte und mit der XPath-Bibliothek nur der bestehende Erweiterungsmechanismus von XSLT bzw. XQuery verwendet werden muss.

Die Funktionen der Bibliothek lassen sich folgendermaßen gruppieren:

- **Informationen zum aktuellen Prozess und der aktuellen Transformation**
  Dieser Teil der Bibliothek stellt Informationen zum aktuellen Propagationsprozess bereit, wie die ID des Prozesses oder der Name des ausgeführten Propagationsskriptes. Weiterhin kann man Informationen über die aktuelle Transformation erfahren, wie z.B. der Name des Transformationsskriptes oder welcher Zustand aktuell transformiert wird (Davor- oder Danach-Zustand). Diese Art von XPath-Funktionen wird eher seltener als die nachfolgenden Funktionen gebraucht. Allerdings kann hiermit auf verarbeitungsinterne Daten der Prozesse zugegriffen werden, die in bestimmten Anwendungsfällen benötigt werden. Zum Beispiel kann hier ein Transformationsskript verwendet werden, das nur bei der Transformation des Danach-Zustandes die relative Verschiebung berechnet. Dies wird durch die Abfrage des aktuell transformierten Zustandes ermöglicht.

- **Zugriff auf Änderungen**
  Da jeder Prozess einen internen Speicher hat, in dem die Änderungsbeschreibungen (ABs) unter ihren Namen abgelegt sind, kann mittels einer Funktion auf diese ABs zugegriffen werden. Dafür wird der entsprechenden Funktion der Name der Änderungsbeschreibung übergeben. Weiterhin kann man die Existenz einer Änderungsbeschreibung abfragen. Diese Funktionalität ermöglicht die Integration von mehreren Änderungsbeschreibungen.

- **Zugriff auf Änderungselemente**
  Mit diesen Funktionen kann auf alle Elemente der Änderung zugegriffen werden, die durch das Tupel $AB = (S, GT, A, B, D, TS)$ definiert sind. Dadurch wird es zum Beispiel möglich, die relative Verschiebung zwischen den beiden Positionen zu berechnen, indem man bei der Danach-Transformation auf den Davor-Zustand zugreift.

3.7 Komponenten

In diesem Abschnitt werden die einzelnen Komponenten des Propagationssystems diskutiert. Als Erstes wird die Basisarchitektur vorgestellt. Danach wird auf jede einzelne Komponente dieser Architektur genauer eingegangen.

In Abbildung 3.6 ist die Architektur mit den Hauptkomponenten eines Propagationssystems dargestellt. Diese Basisarchitektur wurde in [CHRM01] vorgestellt und besteht aus drei Hauptkomponenten: dem Repository, dem Abhängigkeitsmanager und

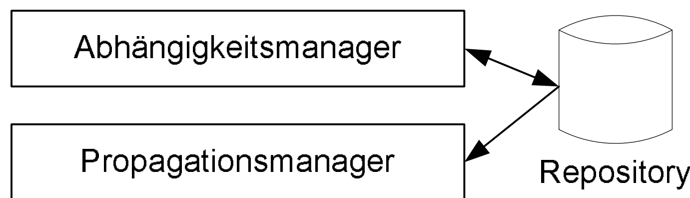


Abbildung 3.6: Basisarchitektur eines Propagationssystems

dem Propagationsmanager. Das Repository stellt einen zentralen Speicher für die Ablage von Abhängigkeiten und den dazugehörigen Daten bereit (vgl. Abschnitt 3.4.2). Zur Realisierung des Repository sollte ein DBMS verwendet werden, um die Konsistenz der darin enthaltenen Daten zu garantieren. Zur Definition oder Bearbeitung von den Daten im Repository wird eine Design-Time-Komponente verwendet, der Abhängigkeitsmanager. Die eigentliche Propagation der Änderungsbeschreibungen von Quellsystemen zu den Zielsystemen wird vom Propagationsmanager ausgeführt. Dazu verwendet er die Daten, die im Repository stehen und vom Abhängigkeitsmanager erstellt bzw. geändert wurden.

Im Nachfolgenden werden die Hauptkomponenten genauer untersucht. Die dort vorgestellten Komponenten sind schon auf eine Propagation von Änderungsbeschreibungen mit XML-Technologien (vgl. Abschnitt 3.4) ausgelegt.

3.7.1 Repository

In diesem Abschnitt wird der Server diskutiert, der die Metadaten für die Propagation bereitstellt, das Repository. Zuerst wird eine Übersicht über Aufgaben und Inhalte des Repository gegeben, bevor das darunter liegende Datenmodell diskutiert wird. Schließlich wird noch auf die Architektur des Repository eingegangen.

3.7.1.1 Übersicht

Das Repository dient als zentraler Speicher für Abhängigkeiten und alle Daten, die von einer Abhängigkeit benötigt werden. Wie schon erwähnt, wird hier eine Abhängigkeit durch ein Propagationsskript realisiert. Folgende Datenarten müssen im Repository gespeichert werden:

- Propagationsskripte (Beschreibung von Abhängigkeiten)
- Systeme (Metadaten über integrierte Informationssysteme)
- Schemas (Struktur der Zustandsbeschreibungen der geänderten Geschäftsobjekte)
- Transformationsskripte (Transformation der Zustandsbeschreibungen).

Im nächsten Abschnitt wird das Datenmodell vorgestellt, das als Grundlage für die Repository-Implementierung verwendet wird, um die oben beschriebenen Datenarten

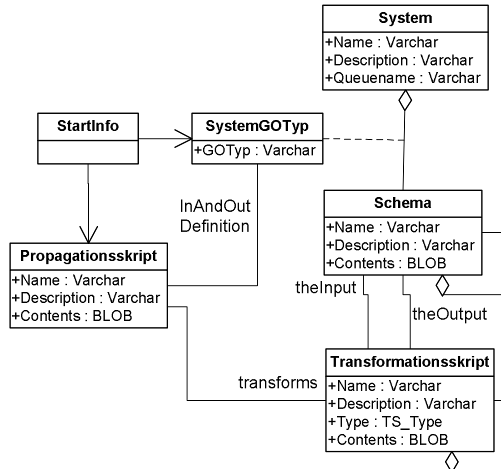


Abbildung 3.7: Datenmodell des Repositorys

zu speichern. Hier wird unter einem Repository kein Software-Entwicklungsrepository [McC93, BD94, Ber98, SBB+99] verstanden, sondern ein zentraler Speicher, der die Daten bereitstellt. Des Weiteren müssen die oben genannten Daten nicht fein granular gespeichert werden, da keine Anfragen auf z.B. Schemainhalte durchgeführt werden müssen. Daraus folgt, dass diese Objekte in sogenannten BLOBs gespeichert werden können und keine XML-Repository-Technologie [SRL00] notwendig ist.

3.7.1.2 Datenmodell

In Abbildung 3.7 ist das Repository-Datenmodell in der UML-Notation dargestellt. Die implementierte Lösung des Repository basiert auf einem relationalen Datenbankmanagementsystem (RDBMS). Kern dieses Modells ist das Propagationsskript bzw. die Abhängigkeit. Das Propagationsskript wird über einen Namen referenziert und enthält das Skript in serialisierter Form, gespeichert als *Binary Large Object* (BLOB). Da unterschiedliche XML-Dokumente auch unterschiedliche Zeichensätze haben können, ist das nicht-interpretierte Format (binär) besser geeignet. Das Propagationsskript enthält Informationen über die Änderungsbeschreibungen, die das Propagationsskript starten sollen (vgl. Abschnitt 3.5.1). Diese Art von Informationen werden redundant vorgehalten (Klasse StartInfo), um eine effizientere Ermittlung der zu startenden Propagationsskripte zu ermöglichen. Die Ermittlung erfolgt anhand von System und Geschäftsobjekttyp (GOTyp), die über SystemGOTyp definiert sind und mit StartInfo verbunden sind. Allerdings muss vom Repository garantiert werden, dass die StartInfo-Informationen mit dem Inhalt der Propagationsskripte konsistent sind.

Grundsätzlich müssen alle Beziehungen zwischen einzelnen Daten, die im Repository gespeichert werden, modelliert und gepflegt werden. Einen Überblick über diese Beziehungen gibt Tabelle 3.4. Aus diesem Grund müssen im Datenmodell auch der Input und Output eines Propagationsskriptes modelliert werden, um ein Löschen von Schemata bzw. System-GOTyp-Paaren zu verhindern. Transformationsskripte müssen ebenfalls vor dem Löschen geschützt werden, sofern diese von Propagationsskripten

verwendet werden. Im Datenmodell muss die Art des Transformationsskriptes (XSLT oder XQuery) abgelegt sein, damit die Laufzeitumgebung erkennt, welcher Prozessor initialisiert werden muss. Diese Unterscheidung erfolgt mit dem Attribut `Type` in der Klasse Transformationsskript (Abbildung 3.7). Das eigentliche Skript wird ebenfalls in der Klasse Transformationsskript vorgehalten. Eine weitere wichtige Beziehung eines Transformationsskriptes ist die Verwendung eines anderen Transformationsskriptes. Dadurch können Transformationsskriptteile ausgelagert und wiederverwendet werden. In diesem Fall besteht eine Aggregationsbeziehung vom Typ n-zu-m, d.h. ein Transformationsskript kann von mehreren Transformationsskripten eingebunden werden. Wiederverwendete Transformationsskripte müssen ebenfalls vor unbeabsichtigtem Löschen geschützt werden. Eine solche Wiederverwendung kann ebenfalls zwischen Schemas definiert werden.

Grundsätzlich werden die Klassen als Relationen im RDBMS abgebildet, bei dem die Attribute zur Spaltendefinition werden. Die Beziehungen zwischen den einzelnen Klassen werden über Fremdschlüsselbeziehungen abgebildet, wobei die Lösch-Semantik 'Restrict' verwendet wird. Dadurch wird ein unbeabsichtigtes Löschen von benötigten Informationen verhindert. Diese Beziehungsinformationen müssen beim Speichern (Anlegen oder Ändern) eines Dokumentes im Repository aus dem Dokument extrahiert und im Repository abgelegt werden. Nicht mehr gültige Beziehungen müssen gelöscht werden. Anzumerken ist, dass der Primärschlüssel (Name) nicht geändert werden kann, so dass die Fremdschlüssel-Beziehungen besser verwaltbar sind.

Betrachtet man das Repository auf einer Architektur-Ebene, so werden spezielle Parser benötigt, die die Beziehungen zwischen den einzelnen Objekten extrahieren. Weiterhin ist es die Aufgabe eines solchen Parsers sicherzustellen, dass das Dokument (z.B. Transformationsskript) schema-konform ist. Um diesen Teil zu realisieren, können die Parser auf einer XML-Parser Komponente aufbauen, welche einen DOM-Baum erzeugt. Mit den speziellen Zugriffsroutinen kann auf die gesuchten Elemente wie *import* und *include* beim XML Schema zugegriffen werden.

3.7.1.3 Architektur

Wie wir festgestellt haben, werden *Parser* für die Überprüfung der einzelnen Skripte sowie Schemas benötigt und für die Extraktion der Beziehungen zwischen diesen Dokumenten. Wie in Abbildung 3.8 dargestellt, existiert ein spezieller Parser für jedes unterschiedliche Dokument: XML Schemas (Schema-Parser), Propagationsskripte (PS-Parser) und Transformationsskripte (XSLT- u. XQuery-Parser). Bis auf den XQuery-Parser basieren alle Parser auf einem XML-Parser, der durch Angabe eines XML Schemas die Gültigkeit eines zu speichernden Dokumentes überprüft. Da der jeweilige Parser auch Kenntnis darüber hat, wie in der entsprechenden Sprache Beziehungen ausgedrückt werden, wird der Parser verwendet, um Beziehungen aus den Dokumenten zu extrahieren und als Fremdschlüsselbeziehungen in der Datenbank abzulegen. Die Datenbank wird durch ein RDBMS verwaltet. Dieses ermöglicht die Sicherstellung der geforderten Konsistenz durch die Definition von Fremdschlüsselbeziehungen.

Weitere wichtige Komponenten sind der *Verbindungsmanager* und der *Repositorymanager*, da sie direkt für den Client sichtbar sind. Repository-Clients sind, wie

Datentyp	Erzeugen	Ändern	Löschen
Propagations-skript (PS)	• Schema konform • TS vorhanden • System-GOTyp-Paar vorhanden	• Schema konform • TS vorhanden • System-GOTyp-Paar vorhanden	–
Transformations-skript (TS)	• Schema konform • ggf. TS vorhanden (Wiederverwendung)	• Schema konform • ggf. TS vorhanden (Wiederverwendung)	• von PS verwendet • von TS verwendet
Schema	• Schema konform • ggf. Schema vor-handen (Wiederver-wendung)	• Schema konform • ggf. Schema vor-handen (Wiederver-wendung)	• von Schema verwen-det • von TS verwendet • von System-GOTyp-Paar verwendet
SystemGOTyp-Paar	System und Schema vorhanden	System und Schema vorhanden	von PS verwendet
System	–	–	wird von SystemGO-Typ verwendet

Tabelle 3.4: Konsistenzregeln für das Propagationsrepository

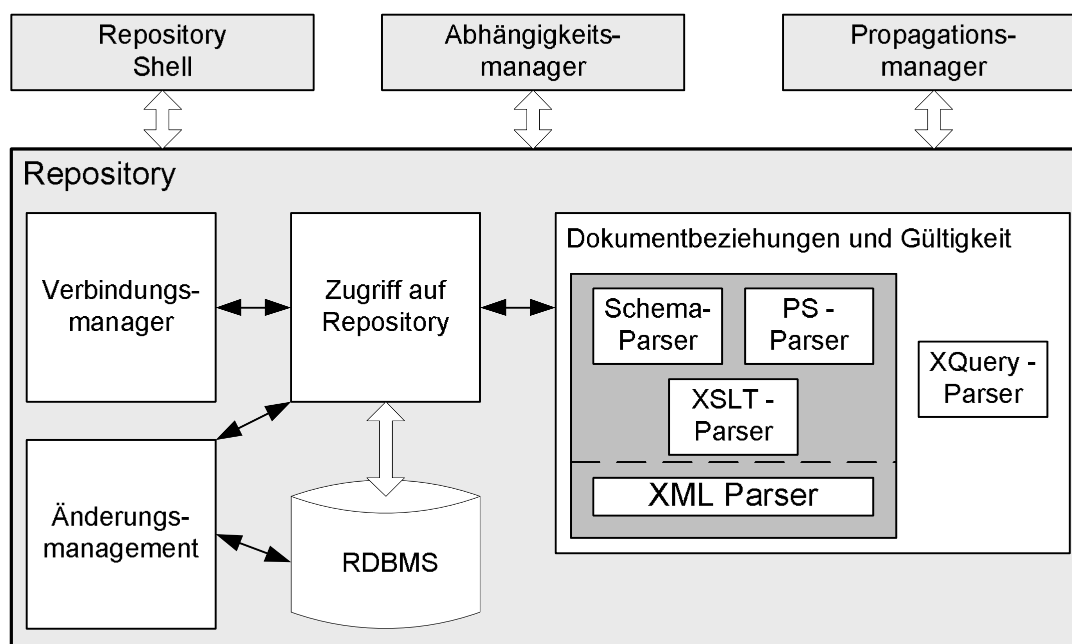


Abbildung 3.8: Architektur des Repositorys

in Abschnitt 3.7 erläutert, der Abhängigkeitsmanager und der Propagationsmanager. Zusätzlich verfügt das Repository über eine *Repository Shell*, mit der auf einfache Weise Daten im Repository eingesehen und verändert werden können. Es gibt zwei Komponenten, die von Clients zur Verwaltung der Daten verwendet werden. Die erste Komponente (Verbindungsmanager) stellt Funktionalität zur Anmeldung am Repository bereit. Diese Verbindung zum Repository kann für die gesamte Zeit, in der das Repository verwendet wird, gehalten werden. Allerdings sollte die Verbindung zum RDBMS nicht an die Verbindung zum Client gekoppelt sein, da diese mit Zeitbeschränkungen versehen sind und dann ihre Gültigkeit verlieren. Außerdem eignet sich in diesem Fall ein sogenannter *Connection-Pool* besser, da dadurch die langen Initialisierungszeiten von Datenbankverbindungen durch Wiederverwendung verhindert werden. Mit JDBC 2.0 [WH99] wird eine Schnittstelle zu relationalen Datenbanken bereitgestellt, die ein Connection-Pooling unterstützt.

Die zweite Komponente, der eigentliche Repositorymanager, stellt eine Reihe von Schnittstellen zum Zugriff auf die Repository-Daten bereit und implementiert diese. Bei Änderung der Repository-Daten muss sichergestellt werden, dass die in Tabelle 3.4 aufgelisteten Konsistenzregeln eingehalten werden. Diese Konsistenzregeln ergeben sich aus dem in Abbildung 3.7 dargestellten Datenmodell. Wie oben schon erwähnt, werden die Konsistenzregeln durch Pflege von Fremdschlüsselbeziehungen und deren Verwaltung im RDBMS garantiert. Dafür verwendet der Repositorymanager die Parser und speichert die extrahierten Fremdschlüsselbeziehungen zusammen mit dem eigentlichen Dokument in der Datenbank.

Eine weitere Komponente kann dazu verwendet werden, Veränderungen in Dokumenten sowie deren Auswirkungen auf andere Dokumente zu protokollieren und dem

Integrationsentwickler bereitzustellen. Wird zum Beispiel ein Schema verändert, so sind die Transformationsskripte, die dieses Schema als Input oder Output haben, davon betroffen und müssen ebenfalls angepasst werden. Diese Komponente liefert dem Entwickler nach der Schemaänderung alle Transformationsskripte, die von der Änderung betroffen sind.

Als nächste Hauptkomponente wird der Propagationsmanager betrachtet, der die Propagationen ausführt.

3.7.2 Propagationsmanager

In diesem Abschnitt wird die Komponente Propagationsmanager diskutiert. Zuerst wird eine Übersicht gegeben, bevor die Architektur diskutiert wird. Schließlich wird noch die Kommunikation zwischen dem Prozessmanager und den Prozessen diskutiert.

3.7.2.1 Übersicht

Der Propagationsmanager ist die Kernkomponente und die Laufzeitumgebung des Propagationssystems. Diese Kernkomponente verwendet die im Repository abgelegten und durch den Abhängigkeitsmanager definierten Abhängigkeiten, um die Änderungsanforderungen gezielt an die betroffenen Informationssysteme zu senden. Dafür werden, wie in Abschnitt 3.7.1 festgestellt wurde, folgende Daten benötigt: Propagationsskripte, Schemas, Systeminformationen und Transformationsskripte. Zusätzlich werden im Repository die im Propagationsskript enthaltene Start-Information gepflegt. Diese Information wird vom Propagationsmanager verwendet, um die entsprechenden Propagationsskripte zu starten. Eine Kernkomponente des Propagationsmanagers ist die XPDL-Engine, mit der die Propagationsskripte ausgeführt werden. Im nächsten Abschnitt wird die Architektur genauer betrachtet.

3.7.2.2 Architektur

In Abbildung 3.9 ist der Aufbau des Propagationsmanagers illustriert. Wie oben erwähnt wurde, ist eine der wichtigsten Komponenten die XPDL-Engine, die es ermöglicht, die Abhängigkeiten bzw. Propagationsskripte, die in XPDL programmiert wurden, auszuführen. Eine Ausführungsinstanz eines Propagationsskriptes ist der eingeführte Propagationsprozess (vgl. Abschnitt 3.2.1). Dieser führt die Beschreibung eines Propagationsskriptes bzw. Abhängigkeit aus. Dafür werden die Befehle der Sprache XPDL (*XML Propagation Definition Language*) in eine ausführbare Objektstruktur übersetzt, wobei jeder Befehl durch eine Klasse und jede Instanz eines Befehls durch ein Objekt repräsentiert werden [Ker01]. Diese Klassen verfügen über jeweils eine Initialisierungs- und eine Ausführungsmethode, die in der entsprechenden Phase ausgeführt werden. Durch die Realisierung von Klassen können neue Befehle hinzugefügt werden, indem sie implementiert und der Engine bekannt gemacht werden. Jede Instanz verfügt über einen Zustand, mit dem festgestellt werden kann, ob der Befehl initialisiert wurde, ausgeführt wird, erfolgreich beendet oder abgebrochen wurde. Diese Zustände können

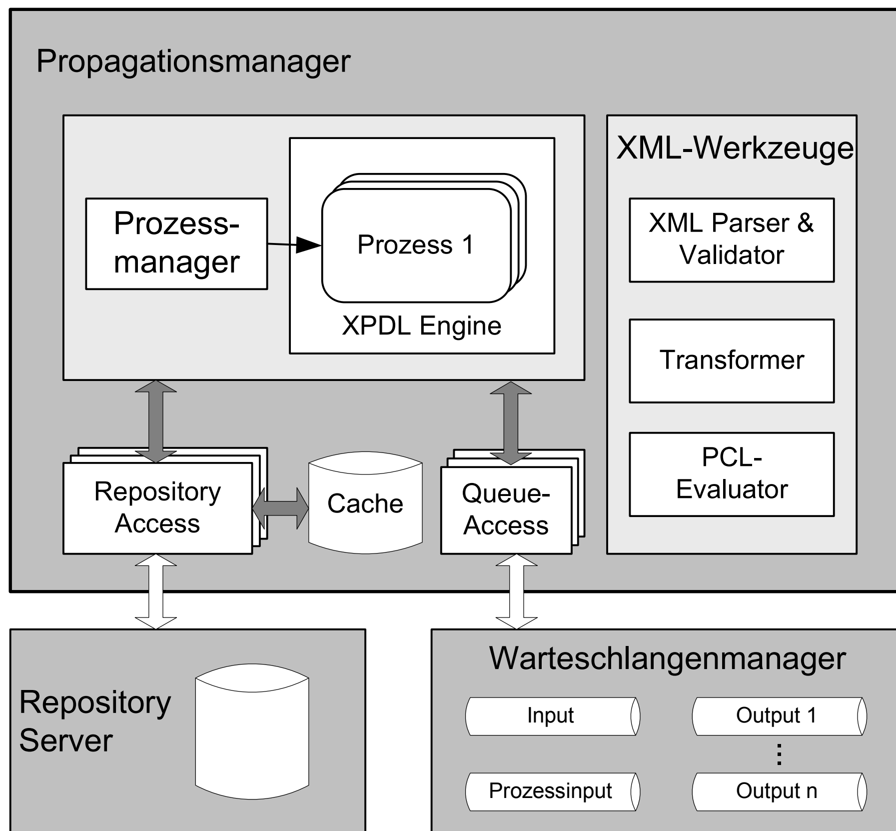


Abbildung 3.9: Architektur des Propagationsmanagers

vom übergeordneten Propagationsprozess beeinflusst werden und damit eine fehlerhafte Ausführung abgebrochen werden. Die Kontrolle über die einzelnen Prozesse, d.h. das Erzeugen und ggf. das Neustarten, übernimmt der Prozessmanager. Er kann auch Prozesse für das Herunterfahren des Propagationssystems abbrechen. Der Ablauf des Prozessmanagers sieht folgendermaßen aus:

1. Transaktionsbeginn
2. Warten auf Änderungsbeschreibungsnachricht von der Eingangswarteschlange (Input)
3. Hole Start-Informationen und die damit verbundenen Propagationsskripte vom Repository
4. Übergabe der Änderungsbeschreibungen an die Prozesse über die Prozesseingangswarteschlange (Prozessinput)
5. Transaktionsende
6. Starte Prozesse mit den jeweiligen Propagationsskripten
7. Gehe zu 1.

Damit es zu keinem Deadlock kommt, wenn die maximale Anzahl gleichzeitiger Prozesse kleiner ist als die Anzahl der Abhängigkeiten pro Änderung, müssen die Propagationsprozesse nach dem Transaktionsende gestartet und initialisiert werden. Die maximale Anzahl gleichzeitiger Prozesse ist eine Möglichkeit zur Regulierung der Performance.

Die Schritte 1-4 werden innerhalb einer Transaktion ausgeführt, sodass bei aufgetretenen Fehlern neu aufgesetzt werden kann. Der Prozessmanager greift dabei auf die Funktionalitäten einer *Queue*-Zugriffskomponente (*Queue-Access*) zurück und verfügt über eine Repository-Zugriffskomponente (*Repository Access*), um die Namen der Propagationsskripte zu bekommen (3.). Diese beiden Komponenten kapseln die Funktionalität der dahinterliegenden Systeme.

Ein Propagationsprozess verfügt ebenfalls über die beiden oben genannten Komponenten, die nachfolgend genauer erklärt werden.

Die Queue-Zugriffskomponente stellt ein vereinfachtes Interface zu den Warteschlangen (Queues) bereit, die in einem externen Warteschlangenmanager definiert sind. Dafür bedient er sich des *Java Message Services* (JMS) [HBS+02b], der eine mächtige Schnittstelle zu den Warteschlangen bereitstellt. Eine JMS-Nachricht besteht aus einem Meta-Teil und der eigentlichen Nachricht, wobei der Meta-Teil selbst aus zwei Teilen besteht, einem festen Teil mit Systemattributen und einem Benutzerteil mit beliebigen Attributen. In letzerem werden die Informationen einer Änderungsbeschreibung bis auf die Änderungszustände B und D ablegt, d.h. Informationen über die Herkunft (System S und Geschäftsobjekttyp GT) und die Änderungsart A. Im festen Teil wird schließlich noch der Änderungszeitpunkt TS oder genauer gesagt die Zeit, zu der die

Update-Änderung	Andere-Änderungen
1. Parsen und Überprüfen der Update-Struktur gegen internes Schema 2. Überprüfen Davor-Zustand (DOM) gegen Schema X 3. Überprüfen Danach-Zustand (DOM) gegen Schema X	Parsen und Überprüfen des Nachrichteninhalts gegen Schema X.

Tabelle 3.5: Schritte beim initialen Parsen der Änderungsanforderung

Änderungsbeschreibung an den Warteschlangenmanager übergeben wurde, festgehalten. Die Herkunftsinformationen (S u. GT) werden vom Prozessmanager extrahiert und dazu verwendet, die Propagationsskripte zu ermitteln, die gestartet werden sollen. Der Name des Propagationsskripts wird dabei dem Prozess übergeben, der seinerseits der XPDL-Engine die Aufforderung gibt, eine lauffähige Instanz des Propagationsskriptes zu erzeugen.

Wie in Abschnitt 3.4 erwähnt, eignet sich XML als Basis für die Implementierung eines Propagationssystems wegen der Selbstbeschreibung und der Vielzahl der Standards und Werkzeuge, die auf XML aufbauen. In diesem Propagationssystem wird deswegen ein XML-Parser verwendet, der zum Aufbau der internen Repräsentation (DOM) von Änderungsbeschreibungszuständen und zur Validierung der erwarteten Struktur eines solchen Zustandes dient.

In Tabelle 3.5 ist dargestellt, welche Schritte beim Parsen der Startnachricht notwendig sind. Diese sind abhängig davon, ob es nur einen Zustand oder zwei Zustände (*Update*) gibt. Im ersten Fall kann der Zustand direkt in der Nachricht kodiert werden. Im zweiten Fall braucht man eine Unterstützungsstruktur, die es ermöglicht, beide Zustände in einem Dokument zu verpacken. Dafür wird ein Dokument mit *Namespaces* verwendet und mit einem Wurzelelement *update*, welches zwei Unterelemente (*before* und *after*) hat. Der Inhalt dieser beiden Unterelemente wird dann noch als beliebig definiert (XML Schema Datentyp *any*). Dadurch können beliebige Zustände in den Unterelementen beschrieben werden, die, wie in Tabelle 3.5, getrennt und mit dem gleichen Schema für beide Zustände überprüft werden.

Für die Implementierung wäre aus Performanzgründen wünschenswert, dass nach dem Parsen der Update-Struktur deren Unterelemente nur noch validiert werden müssten und dafür kein erneutes Parsen notwendig wäre. Dies ist seit DOM-Level 3 möglich [HHW+04]. Allerdings wird dieser zum aktuellen Zeitpunkt noch nicht von allen eingesetzten Werkzeugen unterstützt. Deshalb wurde auf eine Behelfslösung zurückgegriffen, bei der der DOM-Baum zuerst serialisiert und danach erneut geparst sowie anhand des

entsprechenden Schemas validiert wird.

Einen weiteren Bestandteil der verwendeten XML-Werkzeuge stellen die Transformer dar, die in Form von XSLT-Prozessor und XQuery-Engine auftreten. Der Aufruf der richtigen Komponente erfolgt dabei transparent für den XPDL-Entwickler, da er nur den Namen der Transformationsskripte benötigt; die Art wird über das Repository ermittelt. Im Regelfall hat eine Transformation einen DOM-Zustand bzw. ein Dokument als Input und einen DOM-Zustand bzw. ein Dokument als Output. Bei einer *Update*-Änderungsbeschreibung werden beide Zustände nacheinander und transparent für den Programmierer transformiert. Über eine XPath-Bibliothek können noch weitere Zustände (z.B. der Davor-Zustand bei der Danach-Transformation) eingebunden werden und damit Inhalte verbunden werden. Dies ermöglicht zum Beispiel die Berechnung der relativen Verschiebung einer Maschine, wobei im Davor-Zustand B die alte Position und im Danach-Zustand D die aktuelle Position kodiert ist.

Die PCL-Komponente (PCL-Evaluator) ist im Gegensatz zu den anderen zwei nicht als Off-The-Shelve-Komponente zu haben, basiert aber auf einer XPath-Implementierung, die in den meisten Fällen Bestandteil eines XSLT-Prozessors oder einer XQuery-Engine ist.

3.7.2.3 Kommunikation zwischen Prozessmanager und Propagationsprozessen

Die Kommunikation zwischen Prozessmanager und Propagationsprozessen stellt eine Herausforderung dar, denn es können mehrere Propagationsprozesse durch eine eingegangene Änderungsbeschreibung gestartet werden, sofern mehrere Abhängigkeiten für ein System und Geschäftsobjekttyp definiert sind. Die hier gestellten Anforderungen treffen auch auf das Publish-Subscribe-Konzept [EFGK03] zu. Sendet ein Publisher eine Nachricht, so wird diese durch die Infrastruktur an alle interessierten Subscriber verteilt. Dies erfolgt über sogenannte Abonnements (Subscriptions), bei denen der Subscriber mitteilt, an welchen Nachrichten er interessiert ist. Das bedeutet, dass das Publish-Subscribe-Konzept vom Kommunikationssendepunkt gesteuert wird. Für die hier angestrebten Informationsaustausch wird aber eine Kommunikation benötigt, die vom Publisher gesteuert wird, d.h. vom Nachrichtensender bzw. Prozessmanager. Dieses Prinzip ist in Abbildung 3.10 dargestellt. Der Prozessmanager empfängt eine Änderungsbeschreibung über die Eingangswarteschlange (Input), startet die entsprechenden Prozesse (P2 u. P3) und reicht die Änderungsbeschreibung gezielt an die gestarteten Prozesse weiter. Es können aber schon ältere Prozesse (P1) existieren, die die Änderungsbeschreibung nicht bekommen sollen. Um eine Lösung für die Problematik zu finden, werden zunächst die Anforderungen beschrieben.

- Die Verteilung der Änderungsbeschreibungen soll sendergesteuert erfolgen: Beim Versenden der Änderungsbeschreibung an die Propagationsprozesse übergibt der Prozessmanager dem Kommunikationssystem zusammen mit der Änderungsbeschreibung eine Menge $P = \{pid_1, pid_2 \ldots pid_n\}$ von Prozesskennungen, um dem Kommunikationssystem mitzuteilen, für welche Propagationsprozesse die Änderungsbeschreibungen bestimmt sind (vgl. Abbildung 3.10).

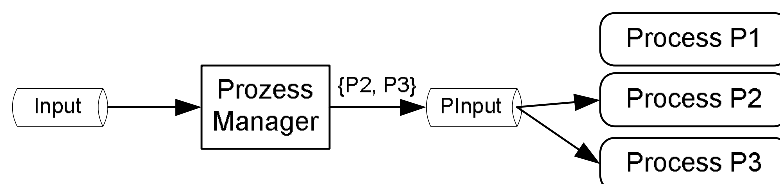


Abbildung 3.10: Problematik der Kommunikation zwischen Prozessmanager und den Propagationsprozessen

- Die Kommunikation soll asynchron erfolgen, d.h. der Propagationsprozess muss noch nicht zum Lesen bereit sein. Dadurch kann der Prozessmanager die Nachrichten versenden, ohne auf die Prozesse zu warten.

- Es soll garantiert werden, dass jede Änderungsbeschreibung von jedem Empfänger nur einmal gelesen wird.

- Es sollen Transaktionen unterstützt werden. Wird ein Propagationsprozess abgebrochen oder ein System-Crash tritt auf, dann müssen alle nicht vollständig verarbeiteten Änderungsbeschreibungen dem Propagationssystem wieder bereitgestellt werden, sodass die entsprechenden Prozesse wieder gestartet werden können. Die Transaktionsunterstützung wird auch beim Senden gefordert, d.h. Änderungsbeschreibungen werden für den Empfänger erst sichtbar, wenn der Sender (Prozessmanager) die Transaktion erfolgreich beendet hat.

- Nach einem Systemabsturz soll dem Prozessmanager alle nicht erfolgreich beendeten Propagationsprozesse mitgeteilt werden, sodass diese erneut gestartet werden können.

- Es sollen Änderungsbeschreibungen zuverlässig an die Propagationsprozesse übertragen werden. Jede Änderungsbeschreibung, die an das Kommunikationssystem gesendet wurde, muss von jedem Prozess vollständig verarbeitet werden. System-Crashs sollen dabei keinen Einfluss auf die sichere Übertragung haben.

Diese Anforderungen können zum Teil von einem zuverlässigen Multicast, wie zum Beispiel MBone [Hau99] erfüllt werden. Allerdings verfügt diese nicht über die geforderte Transaktionalität und die Kommunikation basiert außerdem auf IP-Adressen und nicht auf Prozesskennungen, wie sie hier gefordert werden.

Eine Lösung für das hier gestellte Problem sollte deshalb auf einem Warteschlangensystem basieren. Da die hier gestellten Anforderungen bis jetzt nicht in Produkten verfügbar sind, wurde eine Simulation dieses Prinzips gewählt, die darauf basiert, dass jedem Propagationsprozess über eine zuverlässige Warteschlange die Änderungsbeschreibungen übergeben werden. Jeder Prozess bekommt dabei eine dedizierte Nachricht. Die Prozess-ID wird dabei im Nachrichtenkopf codiert. Mittels JMS [HBS+02b] und dieser Kodierung wird ein selektives Lesen möglich, bei dem jeder Prozess nur seine Nachricht bekommt. Änderungsbeschreibungen, die nicht vollständig verarbeitet worden sind, können zusammen mit den dazu gehörigen Prozessen ermittelt werden, da in diesem Fall die jeweilige Nachricht noch im Warteschlangensystem ist und

die Prozesskennung im Nachrichtenkopf steht. Dies erfolgt indem die Prozesseingangswarteschlange (Prozessinput) beim Recovery untersucht wird und die entsprechenden Prozesse erneut gestartet werden.

Nachdem der Propagationsmanager betrachtet wurde, soll nun die letzte Hauptkomponente betrachtet werden: der Abhängigkeitsmanager.

3.7.3 Abhängigkeitsmanager

Der Abhängigkeitsmanager [CHM02] unterstützt den Entwickler bei der Erstellung der Abhängigkeiten (Propagationsskripte) und der Dokumente, die von diesen Propagationsskripten verwendet werden (vgl. Abbildung 3.3). Diese sind im Einzelnen: Systeminformationen, Schemas, Transformationsskripte und Propagationsskripte (Abhängigkeiten). Systeminformationen werden dabei durch einen einfachen Dialog eingegeben, denn die Informationen sind sehr einfach: Name und Beschreibung des Systems sowie der Name der Warteschlange, die für die Kommunikation mit dem System verwendet wird. Komplizierter ist die Eingabe von Schemas und Transformationsskripten, für deren Erstellung bzw. Modifizierung Standardwerkzeuge existieren, wie beispielsweise Altova Mapforce zur graphischen Erstellung von Transformationsskripten (XSLT u. XQuery). Diese Werkzeuge werden vom Abhängigkeitsmanager eingebunden und bei Bedarf aufgerufen. Die Kommunikation mit dem Werkzeug findet dabei über das Dateisystem statt, d.h. der Abhängigkeitsmanager muss vor dem Werkzeugaufruf die entsprechenden Dokumente aus dem Repository bereitstellen und nach der Bearbeitung im Falle einer Änderung diese wieder ins Repository übernehmen.

Im Gegensatz dazu, handelt es sich bei den Propagationsskripten um keine Standard XML-Technologie und aus diesem Grund erfolgt die Unterstützung der Erstellung bzw. Modifizierung ausschließlich im Abhängigkeitsmanager. Um den Entwickler bei der Erstellung der Abhängigkeiten zu unterstützen, werden drei Darstellungsarten bereitgestellt, die in Abbildung 3.11 illustriert werden. Da die Darstellungsarten nach rechts weiter eingeschränkt werden, sind die Übergänge nach rechts mit Bedingungen versehen. Diese Darstellungsarten und ihre Übergänge werden nachfolgend erklärt.

Zu einer linken Darstellungsart kann man ohne Bedingungen wechseln, will man jedoch in eine rechte Wechseln gelten folgende Bedingungen:

Von Textansicht zu Kontrollflussansicht: Das Dokument muss wohlgeformt und dem XPDL-Schema entsprechen.

Von Kontrollflussansicht zu Abhängigkeitsansicht: Der Kontrollfluss muss einer einfachen Abhängigkeit entsprechen.

3.7.3.1 Textansicht

Die Textansicht ist für den Entwickler die anspruchsvollste Darstellungsart, da der Entwickler direkt mit der internen XML-Repräsentation der Abhängigkeit konfrontiert wird. Dies bedeutet, dass der Abhängigkeitsmanager beim Speichern der Textansicht und beim Übergang zur Kontroll- und Datenflussansicht überprüfen muss, ob

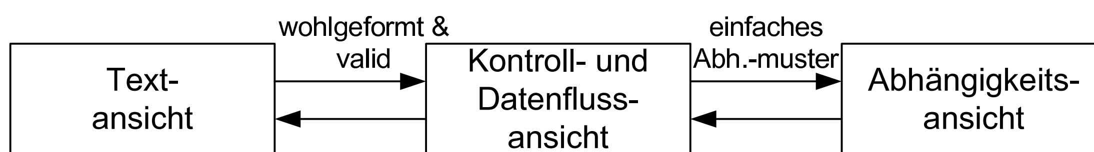


Abbildung 3.11: Die unterschiedlichen Sichten auf ein PS und ihre Übergänge

das Dokument wohlgeformt und gültig ist. Gültig ist das Dokument dann, wenn es schema-konform ist, d.h. nur die vorgesehenen Propagationsskriptbefehle verwendet (vgl. Abschnitt 3.5).

3.7.3.2 Die Kontroll- und Datenflussansicht

Diese Form der Darstellung ist vergleichbar mit der eines Workflows. Der Kontrollfluss wird dabei so dargestellt, dass die Abarbeitungssequenz von Oben nach Unten geht. In Abbildung 3.12 ist ein Beispiel für die Kontroll- und Datenflussansicht dargestellt. Ganz oben steht der Befehl, mit dem die erwartete Art von Änderungsbeschreibungen deklariert wird, die die entsprechenden Propagationsprozesse starten. Parallel ausgeführte Zweige werden dabei von rechts nach links dargestellt. Die einzelnen Befehle werden durch Farben und Formen unterschieden, dadurch kann schnell die Art des Befehles erkannt werden. Da diese Ansicht eine Untermenge der Textansicht ist, können alle Propagationsskripte aus dieser Ansicht in die Textansicht überführt werden. Wie oben schon erwähnt, ist das aber umgekehrt nicht möglich. Die Übergänge von und zur Abhängigkeitsansicht werden im nächsten Abschnitt erklärt. Details für eine mögliche Implementierung können in [Li03] gefunden werden.

Ebenfalls von oben nach unten geht der optional einblendbare Datenfluss. Dieser zeigt an, woher der Input eines Befehls kommt und wohin der Output geht.

3.7.3.3 Die Abhängigkeitsansicht

Deutlich verständlicher für den Entwickler als die Kontroll- und Datenflussansicht ist die Abhängigkeitsansicht (Abbildung 3.13). In dieser Ansicht wird der Datenfluss zwischen den einzelnen Systemen dargestellt und ggf. vorhandene Filter bzw. Transformationsbefehle als Icons (Filter F und Transformation T) dargestellt. Die Abhängigkeitsansicht ist abstrakter als die Kontroll- und Datenflussansicht. Aus diesem Grund können nicht alle gültigen Propagationsskripte in dieser Ansicht dargestellt werden. Diese müssen einem bestimmten Muster folgen:

- Eine `start_input`-Deklaration mit ggf. einer Filterbedingung (F), die im `start_input` codiert ist, da es in XPDL keine Filterbefehle (vgl. Abschnitt 3.5.3.3) gibt.

- Ggf. eine generelle Transformation (T).

- N-Zielsysteme (System 1 und 3), die parallel abgehandelt werden.

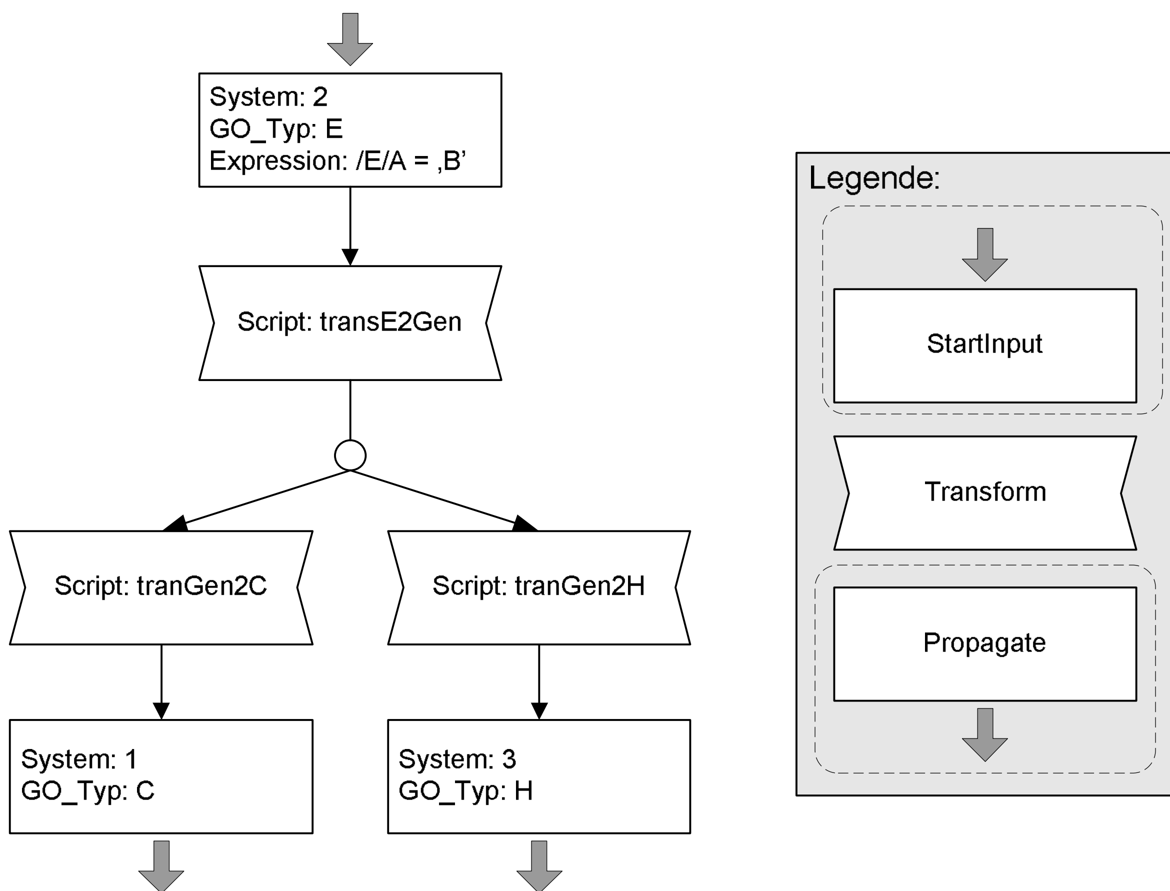


Abbildung 3.12: Die Kontrollflussansicht mit ausgeblendetem Datenfluss. Das Skript implementiert eine Abhängigkeit aus Abbildung 3.2

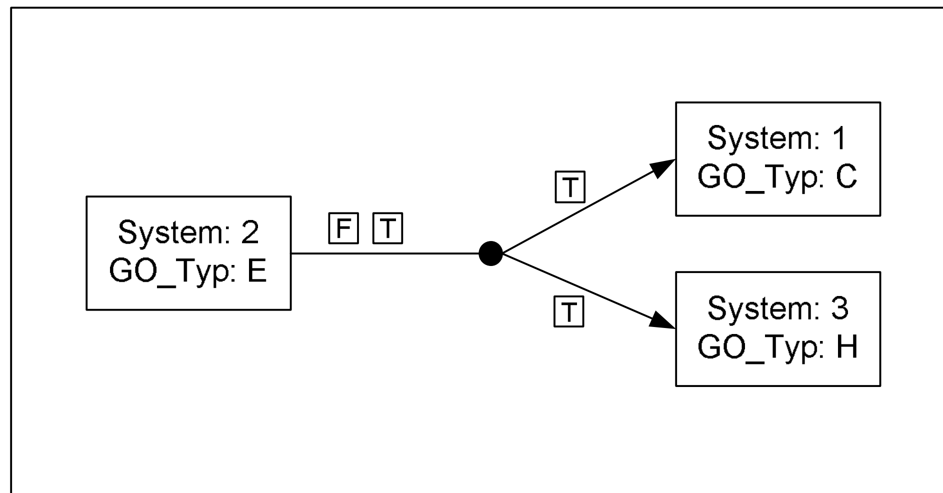


Abbildung 3.13: Die Abhängigkeitsansicht äquivalent zu Abb. 3.12

- Für jeden parallelen Zweig existiert ein `propagate`-Befehl (durch Pfeil dargestellt) und ggf. eine Filterbedingung (F), die über ein `condition`-Befehl realisiert werden. Vor dem `propagate`-Befehl kann noch eine Transformation (T) durchgeführt werden.

3.8 Konflikterkennung und Auflösung

In diesem Abschnitt wollen wir uns mit einem sehr wichtigen Aspekt eines Änderungspropagationssystems beschäftigen: Die Erkennung und Auflösung von Konflikten. Konflikte entstehen, wenn in mindestens zwei beteiligten Systemen "das gleiche" Objekt quasi gleichzeitig geändert wird. Das gleiche Objekt steht dabei für Objekte, die durch bidirektionale Integrationspfade zwischen den Systemen verbunden sind. Die Objekte müssen nicht identisch sein, sondern es reicht aus, wenn nur ein Teil ihrer Daten überlappend und durch Transformationen überführbar sind. Außerdem müssen die Änderungen der Objekte nicht zum exakt gleichen Zeitpunkt erfolgen, sondern es muss für einen Konflikt gelten, dass die Änderung des Objektes in System A zum Zeitpunkt der Änderung im System B noch nicht im System B sichtbar war, sofern die Änderung im System A vor der Änderung in System B erfolgte. In einer Formel ausgedrückt tritt ein Konflikt genau dann auf wenn gilt:

$$iPath(C_A, C_B) \wedge iPath(C_B, C_A) \wedge t(C_A) < t(C_B) \wedge$$
$$t(C_A) + t_{sync}(C_A, A, B) > t(C_B)$$

$\mathbf{iPath(C_x, C_y)}$: Die Funktion *iPath* beschreibt die Ausführung einer Abhängigkeit zwischen der Änderung C_x im Quellsystem x und der Änderung C_y im Zielsystem y.

$\mathbf{t(C_x)}$: Die Funktion t beschreibt den Zeitpunkt einer Änderung C im System X.

$\mathbf{t_{sync}(C_x, X, Y)}$: Die Funktion t_{sync} beschreibt die Zeitdauer einer Synchronisation einer Änderung C_x von System X zu System Y. Der Zeitpunkt $t(C_A)+t_{sync}(C_A, A, B)$ gibt an, wann die Änderung C_A im System B sichtbar war.

Alternativ gibt es noch die Konfliktvermeidung, die durch eine Master-Update-Situation oder durch Eager-Replikation sowie einigen Varianten davon erreicht werden kann. Bei der Master-Update-Konfiguration kann ein Objekt immer nur in einem System geändert werden und bei der Eager-Variante werden alle Systeme innerhalb einer Transaktion angepasst. Durch den Einsatz von Transaktionen für die Änderungspropagation werden die üblichen Änderungskonflikte verhindert. Die Eager-Replikation kann in einem solchen Propagationssystem nicht angewendet werden, da die beteiligten Informationssysteme weitestgehend autonom sein sollen. Oft sollte es aber gegeben sein, dass bestimmte Objekte nur in bestimmten Systemen geändert werden können und wir eine Master-Update-Situation vorliegen haben. Dennoch wird diese Situation nicht immer und für alle Geschäftsobjekttypen erreichbar sein und deshalb muss man Konflikte erkennen und auflösen.

Konflikte können durch logische Vektoruhren [Mat89] erkannt werden, die logische Zeit [Lam78] verwenden. Kann eine logische Uhr von einer einkommenden Synchronisationsnachricht nicht mit der lokalen Uhr in Reihenfolge gebracht werden, so liegt ein Konflikt vor [SS05]. Da logische Vektoruhren einige Nachteile haben [ABF02], wurden einige Varianten [TRA96, BA99, ABF02] entwickelt. Alternativ kann noch reale Zeit verwendet werden, die vom so genannten Two-timestamp-Verfahren [GHOS96] eingesetzt wird, bei der jedes Objekt über einen Zeitstempel (timestamp) der letzten Änderung verfügt. Wird ein Objekt geändert, wird sowohl der Zeitstempel der letzten Änderung als auch der aktuelle mit gesendet. Kommt jetzt eine Änderung in einem entfernten System an, wird der Zeitstempel der letzten Änderung mit dem lokalen verglichen, sind sie gleich, liegt kein Konflikt vor und die Änderung kann durchgeführt werden, andernfalls wurde ein Konflikt erkannt.

Die Konfliktauflösung erfolgt zumeist manuell, in dem ein Administrator die betroffenen Objekte betrachtet und versucht zu entscheiden, welche Daten gültig sind und diese neuen Daten an die beteiligten Systeme verteilt, was wieder über das Propagationssystem erfolgt.

Geschäftsobjekte können groß sein und deshalb kann es zu vielen erkannten Konflikten kommen. Kleinere Objekte schaffen hier Abhilfe [SS05]. Um trotz großer Geschäftsobjekte zu kleinen Objekten zu kommen, wird das Geschäftsobjekt in unabhängige Zonen aufgeteilt. Der Kunde könnte zum Beispiel zwei unabhängige Zonen haben: Adresse und Bankverbindung. Diese Zonen können durch entsprechende Annotationen (*group* und *id*) in den XML Schemas der Geschäftsobjekte definiert werden, wobei die id-Annotation ein eindeutiges Unterscheidungskriterium definiert und die group-Annotation die Zone definiert (vgl. Abbildung 3.14). Da für jede Zone ein Zeitstempel oder eine logische Uhr gespeichert werden muss und oft das Datenmodell eines Informationssystems nicht angepasst werden kann, schlagen wir ein zustandsbasiertes Verfahren vor, das eine Variante vom Two-timestamp-Verfahren darstellt. Statt Zeitstempel werden die Zustände der Objekte verwendet, welche ebenfalls einen bestimmten Zeitpunkt repräsentieren. Der Algorithmus für die Konflikterkennung ist der gleiche wie

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:cd="http://www.uni-stuttgart.de/sfb467/sies/conflict-detection"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

    <xs:element name="Kunde">
        <xs:complexType>
            <xs:sequence>
                <xs:element cd:id="true" name="ID"/>
                <xs:element cd:group="name" name="Vorname"/>
                <xs:element cd:group="name" name="Nachname"/>
                <xs:element cd:group="address" name="Adresse">
                </xs:element>
                   ...
                <xs:element cd:group="bank" name="Bankverbindung">
                   ...
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Abbildung 3.14: Beispiel eines XML Schemas für eine feingranulare Konflikterkennung

beim Two-timestamp-Verfahren. Wird in System A eine Änderung ausgeführt, wird diese mit dem Davor- und Danachzustand an das Zielsystem gesendet. Das Propagationssystem sendet schon aus anderen Gründen zwei Zustände (vgl. Abschnitt 3.2.2) und damit kann dieses Verfahren gut in einem solchen Propagationssystem eingesetzt werden. In System B wird der Davorzustand mit dem aktuellen Zustand verglichen, sind diese gleich liegt kein Konflikt vor und die Synchronisation kann durchgeführt werden. Bei diesem zustandsbasierten Verfahren handelt es sich um eine erste Lösung für eine Konflikterkennung in solch einem Propagationssystem. Die Betrachtung von Alternativverfahren würde den Rahmen dieser Arbeit sprengen.

Das Prinzip eines performanten Algorithmus für das zustandsbasierte Verfahren für Informationssysteme mit relationalen Datenbanken ist, dass die Änderung im relationalen System nur ausgeführt wird, wenn der Zustand mit dem Davorzustand übereinstimmt. Dies erfolgt durch entsprechende Selektion in der Änderungsoperation. Falls kein Objekt mit dem Zustand gefunden wurde, haben wir einen Konfliktkandidaten. Dieser Kandidat wird anhand seiner unabhängigen Zonen analysiert: erfolgte die Änderungen in unabhängigen Zonen, kann die Änderung doch noch durchgeführt werden, da kein Konflikt vorliegt. Die hohe Performanz wird dadurch erreicht, dass der Primärschlüssel sich ebenfalls im Selektionskriterium befindet. Dadurch wird das Objekt schnell gefunden und erst wenn es gefunden wurde, werden die anderen Selektionskriterien ausgewertet.

3.9 Reihenfolgeeinhaltung von propagierten Änderungsbeschreibungen

Die Änderungsbeschreibungen werden durch unabhängige Propagationsprozesse verarbeitet, die unterschiedliche Laufzeiten haben. Dies kann zur Vertauschung der Änderungsbeschreibungen für ein Zielsystem führen. Da Änderungsbeschreibungen grundsätzlich nicht als unabhängig angesehen werden können, muss die Reihenfolge der Änderungsbeschreibungen eingehalten werden. Beispiele für Beziehungen zwischen den einzelnen Änderungsbeschreibungen sind: Änderungen des gleichen Geschäftsobjektes, die zeitnah erfolgten oder es werden zwei unterschiedliche Geschäftsobjekte erzeugt, bei dem das zuletzt erzeugte Objekt eine Referenz auf das erste hat.

Um in replizierten Datenbanken überall die gleiche Reihenfolge zu garantieren, wurde die 1-Kopien-Serialisierbarkeit (One Copy Serializability) [BG82, BG83] eingeführt. Es sind dabei nur Schedules zulässig, die zu serialisierbaren Schedules auf einer nicht redundanten Datenbank äquivalent sind [Rah94]. Diese Eigenschaft wird in der Regel bei der Lazy-Replikation nicht erreicht [BKR+99, ATS+05]. Lösungen, die dieses Problem beseitigen, sind reine Eager-Replikationsansätze oder hybride Ansätze [BK97, BKR+99, ATS+05], die Eager- und Lazy-Replikation kombinieren. In Abschnitt 3.1 wurde argumentiert, dass innerhalb unabhängiger (autonomer) und heterogener Informationssysteme nur die Lazy Replikation in Frage kommt und dadurch die vorgeschlagenen Algorithmen nicht angewendet werden können, d.h. die 1-Kopien-Serialisierbarkeit kann in einem solchen Umfeld vermutlich nicht erreicht werden.

Aus diesem Grund greifen wir auf Reihenfolgeregeln zurück, die von Verteilten Systemen stammen und Ordnungen genannt werden. Bevor diese definiert werden können, müssen noch ein paar grundlegende Dinge eingeführt werden:

P: Menge aller Propagationsprozesse. Jeder Propagationsprozess hat einen Startknoten und mehrere Zielknoten, wobei der Startknoten über gerichtete Kanten mit den Zielknoten verbunden ist.

sn(p): Gibt den Startknoten eines Propagationsprozesses $p \in P$ zurück.

$\mathbf{Z_p}$**:** Menge aller Zielknoten eines Propagationsprozesses $p \in P$.

$\mathbf{kn_1 \sim kn_2}$**:** Definiert eine Relation, bei der die Knoten (kn_1, kn_2; Start oder Zielknoten) das gleiche Informationssystem als Quelle bzw. Ziel haben.

$\mathbf{kn_1 < kn_2}$**:** Die Relation $<$ definiert, dass kn_1 zeitlich vor kn_2 lag.

Nun können die Ordnungen definiert werden, wobei als Grundlage [CBMT96] diente:

1. **Totale Propagationsordnung:**

   Bei der totalen Propagationsordnung werden alle Änderungsbeschreibungen in Reihenfolge gebracht, sofern sie für das gleiche Zielsystem bestimmt sind.

   $\forall p_i, p_j \in P, \forall z_{p_i} \in Z_{p_i}, \forall z_{p_j} \in Z_{p_j}:$
   $p_i \neq p_j \wedge sn(p_i) < sn(p_j) \wedge z_{p_i} \sim z_{p_j} \Rightarrow z_{p_i} < z_{p_j}$

2. **FIFO-Propagationsordnung mit Berücksichtigung von Startsystem und Zielsystem:**

   Bei der FIFO-Ordnung werden Änderungsbeschreibungen in Reihenfolge gebracht, wenn sie vom gleichen Quellsystem und für das gleiche Zielsystem bestimmt sind.

   $\forall p_i, p_j \in P, \forall z_{p_i} \in Z_{p_i}, \forall z_{p_j} \in Z_{p_j}:$
   $p_i \neq p_j \wedge sn(p_i) \sim sn(p_j) \wedge sn(p_i) < sn(p_j) \wedge z_{p_i} \sim z_{p_j} \Rightarrow z_{p_i} < z_{p_j}$

3. **Ungeordnet:**

   Bei dieser Propagationsordnung wird keinerlei Einfluss auf die Reihenfolge der Änderungen genommen, d.h. die Änderungen werden so angewendet, wie sie durch die Propagationsprozesse verarbeitet wurden.

Diese Propagationsordnungen können mit der Konfiguration des Propagationssystems eingestellt werden. In unserem Prototypen ist die Verwendung der FIFO-Propagationsordnung als Standard eingestellt, da sie für die meisten Anwendungsfälle ausreichend ist. Dabei werden alle Änderungsbeschreibungen, die vom gleichen System stammen und für das gleiche System bestimmt sind, in Reihenfolge gebracht.

3.10 Fehlerbehandlung

In diesem Abschnitt werden die Fehler und deren Behandlung diskutiert, die im Prozessmanager oder den Propagationsprozessen auftreten können. Bevor auf diese eingegangen werden, sollen die möglichen Fehler klassifiziert werden.

3.10.1 Fehlerklassifikation

Die Fehler, die während einer Änderungspropagation auftreten, können in verschiedene Kategorien klassifiziert werden. Zuerst kann man unterscheiden, ob es sich um ein einmaliges Auftreten handelt oder ob sich der Fehler wiederholt. Zustandsbeschreibungen die nicht schema-konform sind, ist ein Beispiel für wiederholende Fehler. Ansonsten kann die Wiederholbarkeit schwer vorausgesagt werden und es wird einfach eine obere Grenze von Wiederholversuchen festgelegt. Weiterhin kann die Quelle des Fehlers unterschieden werden, ob der Fehler durch ein externes System oder intern auftritt (z.B. Skript oder Warteschlangenmanager). Ähnlich dazu lässt sich unterscheiden, ob es sich um einen Fehler der Abhängigkeitsdefinition, einen Kommunikationsfehler oder einen Lesefehler der Definitionen handelt. Außerdem kann man einen Fehler nach dem Ort des Auftretens unterscheiden: ob der Fehler im Prozessmanager oder in einem Prozess auftritt.

Wichtig ist vor allem die Unterscheidung, ob sich der Fehler voraussichtlich wiederholt oder ein einmaliges Auftreten hat. Im letzteren Fall kann versucht werden, den Prozessmanager oder den Prozess neu zu starten.

3.10.2 Fehlerbehandlung im Prozessmanager

Die Fehlerbehandlung im Prozessmanager erfolgt vom Lesen einer Änderungsanforderung, dem Starten der jeweiligen Prozesse, bis zum Schreiben in die Warteschlange zur Kommunikation mit den Prozessen. Dieser Ablauf erfolgt unter der Kontrolle einer Transaktion. Beim erfolgreichen Ausführen wird die Transaktion mit einem *Commit* abgeschlossen, so dass die Leseoperation zu einem Entfernen der Änderungsanforderungen aus der Eingangswarteschlange führt und die Schreiboperationen sichtbar werden. Handelt es sich um einen sich nicht wiederholenden Fehler, so wird ein Rollback durchgeführt. Komplizierter ist das Vorgehen bei einem sich wiederholenden Fehler, da in diesem Fall ein Rollback durchgeführt werden muss, damit alle Schreiboperationen zurückgesetzt werden. Dann wird die Nachricht nochmals gelesen und in einem Fehlerspeicher abgelegt. Der Fehlerspeicher kann die Form einer Tabelle in einer Datenbank oder die Form einer Warteschlange haben, wobei im letzteren Fall die Änderungsanforderung mit der Fehlerinformation angereichert werden muss, um einem Administrator zur Verfügung gestellt zu werden. Im Fall der Verwendung einer Tabelle kann die Fehlerinformation in einer Extra-Spalte gespeichert werden, unabhängig von der Änderungsanforderung. Der Ansatz mit der Warteschlange wird zum Beispiel in dem Produkt BizTalk verwendet [AHH+02, WML+05].

3.10.3 Fehlerbehandlung eines Propagationsprozesses

Wie schon erwähnt wurde, ist ein Propagationsprozess die ausführende Instanz eines Propagationsskriptes. Die Fehlerbehandlung innerhalb eines Propagationsskriptes erfolgt transparent für den Programmierer, wobei die Transaktionsgrenzen sich zwischen Start und Beenden eines Prozesses befinden. Einfacher gestaltet sich wieder die erfolgreiche Ausführung, bei der am Ende ein Commit ausgeführt wird. Bei einem sich nicht wiederholenden Fehler wird ein Rollback ausgeführt, der Prozess neu initialisiert und dann neu gestartet wird. Komplizierter ist die Reaktion bei einem sich wiederholenden Fehler. Dabei wird der Prozess zurückgesetzt, die Änderungsbeschreibung nochmals gelesen und mit Fehlerinformation angereichert und an die Fehlerwarteschlange gesendet. Dort kann sie dann von einem Administrator abgeholt, der Fehler behoben und wieder an das Propagationssystem übergeben werden.

Der Unterschied zwischen einer 1-zu-N-Abhängigkeit und n 1-zu-1-Abhängigkeiten ist bei der Fehlerbehandlung besonders groß. Tritt bei der Ausführung ein Fehler auf, so wird im 1-zu-N-Fall keine Änderungsbeschreibung propagiert, während beim 1-zu-1-Fall n Propagationsprozesse separat ausgeführt werden, von denen manche erfolgreich ausgeführt werden und manche durch einen Fehler zurückgesetzt werden.

3.11 Adapter

In diesem Abschnitt wird zuerst der generelle Aufbau eines Adapters und seine Funktionen untersucht. Weiterhin wird die Realisierung eines Adapters für relationale Datenbanken untersucht, wobei die Konzepte für Ziel- und Quelladapter getrennt betrachtet werden.

3.11.1 Genereller Adapter

Die Aufgaben eines Adapters unterteilen sich in zwei Gruppen: Aufgaben eines Quelladapters und Aufgaben eines Zieladapters. Zu den Aufgaben eines Quelladapters gehören das Erkennen von Veränderungen in den Daten des Informationssystems, die Erstellung der XML-Repräsentation der Änderungsinformation und das Übergeben der Änderungsinformation an das Propagationssystem. Die Aufgaben des Zieladapters sind das Warten auf Änderungsinformationen in der entsprechenden Warteschlange, das Parsen der Änderungsinformationen und das Anwenden dieser auf die Daten des Informationssystems. Da aufgrund von Konsistenzregeln, die im Informationssystem definiert sind, Änderungen abgelehnt werden können, müssen Fehlerinformationen abgespeichert werden können. Außerdem muss der Zieladapter über Mechanismen verfügen, die die Erkennung von Konflikten ermöglichen, so wie sie in Abschnitt 3.8 eingeführt wurden.

In Abbildung 3.15 ist der grundsätzliche Aufbau eines Quell- und Zieladapters dargestellt. In diesem Bild sind beide Arten vereint, da viele Systeme sowohl Quell- als auch Zielsystem sind. Der Adapter hat drei Schnittstellen nach außen: eine zum Propagationssystem, eine zum Informationssystem und eine zu Benutzern. Die Schnittstelle zum Propagationssystem führt über das Warteschlangensystem, wobei Änderungsbe-

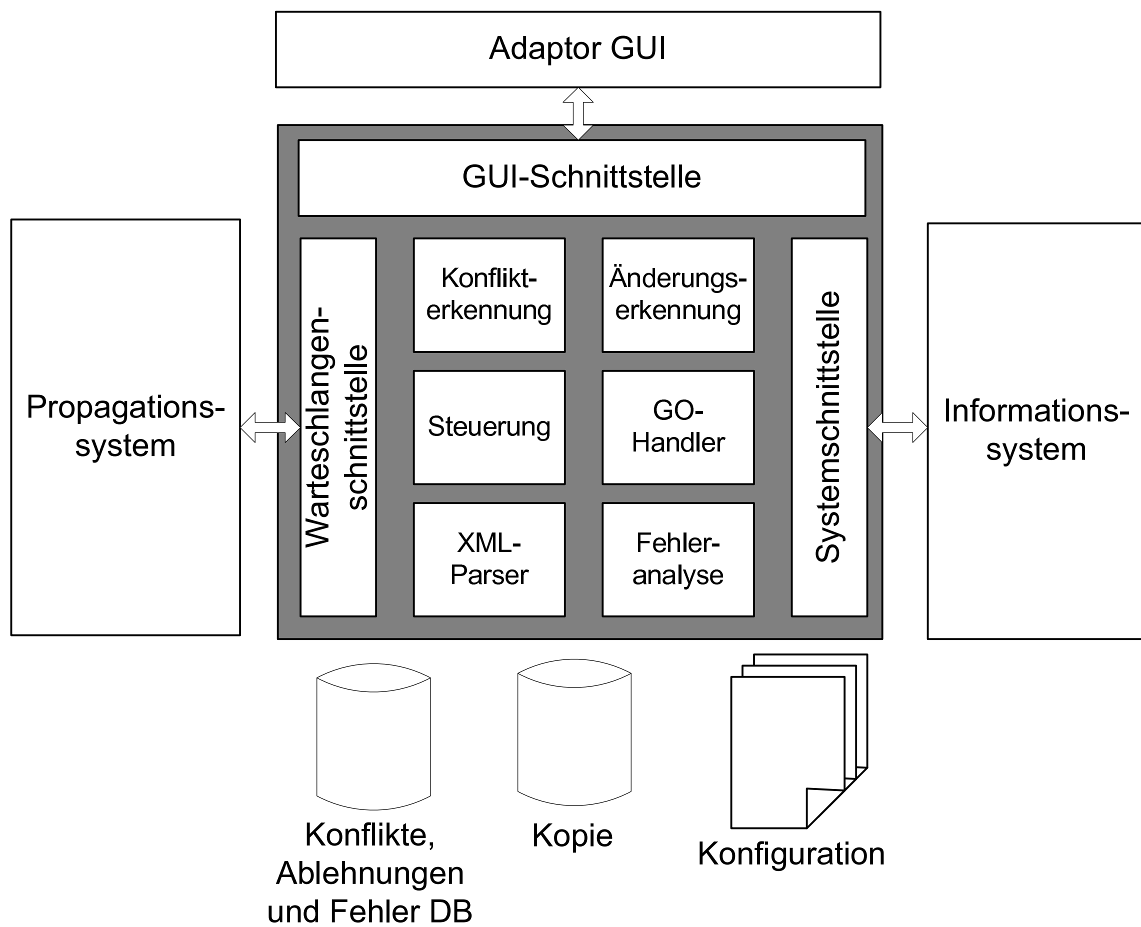


Abbildung 3.15: Die Architektur eines generellen Quell- und Zieladapters

schreibungen sowohl versendet als auch empfangen werden können. Die Schnittstelle zum Informationssystem führt über die Zugriffsmöglichkeiten des jeweiligen Systems. Dabei kann es sich beispielsweise um eine spezielle API handeln. Die dritte Schnittstelle, die Benutzerschnittstelle, ermöglicht die Auflösung von Konflikten, das Ansehen von nicht anwendbaren Änderungsbeschreibungen[5] und die Auflösung dieser.

Im Zentrum des Adapters steht die Steuerungskomponente, die die oben genannten Aufgaben koordiniert und dafür die anderen Komponenten verwendet. In der Konflikterkennung werden die Regeln aus Abschnitt 3.8 implementiert und dadurch eine Erkennung von Konflikten möglich. Die Änderungserkennung erfolgt durch die entsprechende Komponente. Dies kann beispielsweise durch den Vergleich mit Kopien erfolgen, die in der Kopie-DB gespeichert sind. Gegebenenfalls ist diese Komponente unnötig, da das Informationssystem Änderungen aktiv mitteilt. Erkannte Änderungen werden durch den Geschäftsobjekt-Handler (GO-Handler) als Änderungen von Geschäftsobjekten transformiert und schließlich von der Steuerungseinheit als Änderungsbeschreibungen verschickt. Der GO-Handler ist deshalb notwendig, da Änderungen feingranularer erkannt werden können, beispielsweise wird von einem Auftrag die Änderung einer Auftragsposition erkannt. Der GO-Handler dient aber auch zur Zerlegung der Geschäftsobjekte in ihre Speicherungseinheiten, sofern die Systemschnittstelle dies verlangt. Für die Analyse der Änderungszustände verwendet der GO-Handler einen XML-Parser. Sofern die Systemschnittstelle auf der Ebene von Geschäftsobjekten arbeitet und Änderungen auch auf dieser Ebene mitgeteilt werden, kann der GO-Handler weggelassen werden. Das jeweilige Informationssystem kann die Änderungen ablehnen. In diesem Fall muss der Adapter den gemeldeten Fehler analysieren und in die Fehler-DB mitsamt der Änderungsbeschreibung schreiben, so dass ein Administrator den Grund oder Fehler analysieren und Korrekturmaßnahmen ergreifen kann. Der gesamte Prozess vom Empfang einer Änderungsbeschreibung bis zur Anwendung der Änderungen muss steuerbar sein. Das gilt auch für den umgekehrten Fall, bei dem Änderungen erkannt werden und als Änderungsbeschreibungen an das Propagationssystem verschickt werden. Dafür dient die Konfiguration mittels der Konfigurationsdatei.

3.11.2 Adapter für relationale Datenbanken

Da die Informationssysteme oft Konsistenzregeln in der Anwendungsschicht realisiert haben und da auch aus anderen Gründen (z.B. implementierte Geschäftslogiken) nicht direkt auf die darunter liegende Datenbank zugegriffen werden soll, ist es oft unabdingbar, die API eines Informationssystems zu verwenden. Allerdings gibt es auch Systeme, die auf einer relationalen Datenbank beruhen und keine API bereitstellen. Aus diesem Grund muss in diesem Fall direkt auf die Datenbank zugegriffen werden. Daher wird in diesem Abschnitt die Lösung eines generellen Quell- bzw. Zieladapters für relationale Datenbanken genauer untersucht.

Grundsätzlich müssen Quell- und Zieladapter miteinander kommunizieren, so dass vom Zieladapter empfangene Änderungen nicht wieder vom Quelladapter erkannt werden und es dadurch zu sogenannten Propagationsschleifen [Spr05] kommt.

[5] Beispielsweise aufgrund von Konsistenzregeln

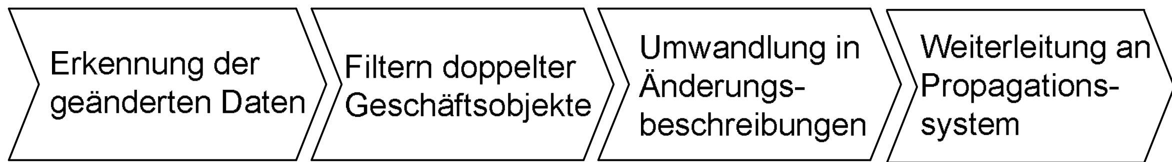


Abbildung 3.16: Der Prozess für die Erkennung von Änderungen und Weiterleitung an das Propagationssystem [Spr05]

3.11.2.1 Quelladapter

Der Quelladapter [Spr05] dient dazu, veränderte Geschäftsobjekte an das Propagationssystem weiterzuleiten. Problematisch dabei ist, dass Geschäftsobjekte nicht als solche in einer relationalen Datenbank vorliegen, sondern in ihren Bestandteilen. Beispielsweise setzt sich ein Kundenauftrag aus einem Auftragskopf und seinen Positionen zusammen. Um diesen Unterschied zu überbrücken, muss der Quelladapter den in Abbildung 3.16 dargestellten Prozess implementieren.

Als Erstes muss ein solcher Adapter Veränderungen in den Daten feststellen und das zugehörige Geschäftsobjekt ermitteln. Dieser Schritt kann zum Beispiel durch den Vergleich mit einer Kopie erfolgen. Dafür werden beide Tabellen, das Original und die Kopie, sortiert und schrittweise verglichen. Dadurch können sowohl neue Objekte (nur im Original vorhanden), geänderte Objekte (in beiden vorhanden, aber mit unterschiedlichen Werten) und gelöschte Objekte (nur in der Kopie vorhanden) erkannt werden. Außerdem ermöglicht dieser Ansatz eine vollständige Beschreibung der Änderung mit Davor- (B) und Danach-Zustand (D). Als Ergebnis des Erkennungsschrittes erhält man eine Liste von geänderten Geschäftsobjekten. Diese Liste enthält allerdings Duplikate, da beispielsweise Auftragskopf und Auftragspositionen des gleichen Auftrags geändert wurden. Deshalb müssen im nächsten Schritt mehrmals vorhandene Geschäftsobjekte gefiltert werden, sodass diese nur noch einmal vorhanden sind. Die resultierende Liste wird wieder als Input für den nächsten Schritt verwendet, welcher die Änderungsbeschreibungen erzeugt. Dafür muss eine Zusammensetzung des Geschäftsobjektzustandes aus den Daten in der relationalen Datenbank bekannt sein. Im letzten Schritt werden die Änderungsbeschreibungen an das Propagationssystem gesendet, welches diese dann verteilt.

3.11.2.2 Zieladapter

Der Zieladapter liest Änderungsbeschreibungen ABs aus der jeweiligen Warteschlange und transformiert diese in eine Menge von DML-Befehlen, die dann an die Datenbank gesendet werden. Diese Transformation ist der Kernpunkt eines Zieladapters für relationale Datenbanken. Dafür muss eine passende Beschreibung gefunden werden, sodass XML-Elemente der Zustandsbeschreibungen (B und D) in DML-Befehle transformiert werden können.

Es wurden für das Mapping vier verschiedene Varianten identifiziert:

Annotiertes XML Schema

Um das Mapping zu definieren, wird das XML Schema, das die Zustände definiert mit Mapping-Informationen annotiert. Der Vorteil hierbei ist, dass das XML Schema schon im Propagationssystem vorhanden ist und dann nur noch importiert und annotiert werden muss. Der Nachteil ist die Komplexität der Umsetzung und die Verständlichkeit des annotierten Schemas.

Zielstruktur-basierte Mapping-Definition[Vu05]

Hier wird nicht, wie bei der ersten Mapping-Definition, die Quellstruktur als Ausgangspunkt für die Mapping-Definition genommen, sondern die Zielstruktur. Dies bedeutet für den Zieladapter, das relationale Schema. Die Zuordnung von Tabellenzeilen und -spalten erfolgt mit XPath-Ausdrücken. Man erlangt durch die Verwendung von XPath eine hohe Mächtigkeit. Außerdem kann zur Erkennung von Konflikten (siehe Abschnitt 3.8) der Primärschlüssel einfach definiert werden oder direkt aus dem Katalog ausgelesen werden. Ein Nachteil ist, dass als Grundlage eine bisher nicht vorhandene Struktur verwendet wird, die sich aber aus dem DB-Katalog generieren und dann mit den XPath-Ausdrücken annotieren ließe, die die Daten aus der Zustandsbeschreibung extrahieren.

Aufteilung durch Abhängigkeit

Hier teilt ein Transformationsskript innerhalb einer Abhängigkeit das Geschäftsobjekt in seine relationalen Bestandteile auf. Dies muss natürlich sowohl für den Davor- als auch für den Danach-Zustand gemacht werden. Der Adapter müsste die Beschreibungen in Beziehung setzen (Zeilen vom Davor- und Danach-Zustand) und diese entsprechend in DML-Befehle umsetzen. Dies bedeutet eine einfache Umsetzung im Adapter. Allerdings kann das Geschäftsobjekt nicht mehr identifiziert werden, da es schon in seine Bestandteile zerlegt wurde. Was daraus folgt, ist außerdem die unterschiedliche Repräsentation der Geschäftsobjekte als Quell- oder Zielstruktur.

Aufteilung im Adapter

Mittels eines Transformationsskriptes im Adapter kann das Geschäftsobjekt in seine Bestandteile zerlegt werden. Dadurch wäre die Quell- und Zielstruktur gleich und die Zerlegung würde im Adapter erfolgen. Allerdings müssen noch die Primarschlüssel festgelegt werden, die zur Erstellung der DML-Statements benötigt werden. Dies könnte ebenfalls durch die Transformationsskripte erfolgen.

Am einfachsten und flexibelsten ist die Lösung mit der Aufteilung innerhalb des Adapters mit Transformationsskripten.

Nicht nur das Mapping von XML-Zuständen zu relationalen Zeilen und Spalten stellt ein Problem dar, sondern auch die Umsetzung der daraus generierten Zwischenrepräsentation in DML-Statements. Dafür sollte bekannt sein, welche Werte den Primärschlüssel bilden. Denn ohne diese Definition müsste der *where*-Teil der Anfrage alle Elemente des Davor-Zustandes enthalten und eventuell aufgetretene Konflikte (siehe Abschnitt 3.8) würden die Identifikation verhindern.

3.12 Zusammenfassung

In diesem Kapitel wurde die Basis für ein Propagationssystem zur Integration von autonomen und heterogenen Informationssystemen geschaffen. Zuerst wurden verschiedene Ansätze, die für die Replikation im homogenen Bereich existieren, untersucht. Als Grundlage für ein Propagationssystem wurde die Lazy-Replikation und das aktive Verbreiten von Änderungsinformationen (Push) verwendet, wobei die Propagationen nach den jeweiligen Quelltransaktionen ausgeführt werden (*deferred*). Außerdem wurde die Propagation von Zuständen ausgewählt, da Zustände sich leichter als Operationen transformieren lassen.

Grundbegriffe für eine solche Propagation sind Änderungsbeschreibungen ABs und Abhängigkeiten. Änderungsbeschreibungen sind Tupel, die alle wichtigen Informationen über eine Änderung eines Geschäftsobjektes enthalten. Die wichtigsten Elemente sind die zwei Zustände, die die Änderung des Geschäftsobjektes beschreiben. Zwei Zustände beschreiben die Änderung vollständig, auch wenn sie nicht immer notwendig wären. Außerdem ermöglichen die zwei Zustände Änderungen von Schlüsseln und die zeitstempellose Erkennung von Konflikten. Abhängigkeiten beschreiben Pfade, entlang derer Änderungsbeschreibungen von Quellsystemen zu Zielsystemen laufen sollen. Diese werden über sogenannte Propagationsskripte realisiert. Außerdem wurden Instanzen eingeführt, welche die Propagationsskripte ausführen. Diese Instanzen heißen Propagationsprozesse.

Des Weiteren wurde untersucht, ob Transaktionsinformationen im heterogenen Fall mit propagiert werden. Da der Transaktionskontext nicht unbedingt erhalten bleibt, ist es nicht immer sinnvoll, diese Informationen mitzusenden. In dieser Arbeit wurde davon ausgegangen, dass der Transaktionskontext zwischen den einzelnen Informationssystemen nicht erhalten bleibt und deshalb die Transaktionsinformationen nicht mit gesendet werden müssen.

Um Änderungsbeschreibungen im heterogenem Fall zu propagieren, wurde XML ausgewählt und eine spezielle Sprache auf Basis von XML entwickelt, mit der Propagationsskripte definiert werden können. Diese Sprache heißt XML Propagation Definition Language (XPDL). Außerdem wurde die Architektur des Propagationssystems vorgestellt. Um Bedingungen zwischen einzelnen Zuständen von Änderungsbeschreibungen zu ermöglichen, wurde die Propagation Condition Language (PCL) eingeführt, die auf XPath aufbaut.

Treten durch Abhängigkeiten Zyklen auf, so können Konflikte entstehen, die erkannt und aufgelöst werden müssen. Dafür wurden mehrere Varianten untersucht, wobei eine Variation des Two-timestamp-Verfahrens vorgeschlagen wurde. Um die auftretenden Konflikte zu minimieren, können Geschäftsobjekte in unabhängige Zonen aufgeteilt werden, innerhalb derer gleichzeitig in unterschiedlichen Systemen geändert werden können (beispielsweise Bankverbindung und Adresse).

Weiterhin sollte die Reihenfolge der Änderungen eingehalten werden. Aufbauend auf Reihenfolgeordnungen in Verteilten Systemen wurden drei Reihenfolgeordnungen für ein Propagationssystem eingeführt. Diese sind die totale, FIFO-Ordnung sowie keine Ordnung (ungeordnet).

Ebenfalls wurde die Fehlerbehandlung in einem Propagationssystem diskutiert und die Aufgaben sowie Aufbau von Adaptern vorgestellt.

KAPITEL 4

Komplexe Propagation

Nachdem das Propagationssystem grundlegend vorgestellt und erklärt wurde, sollen in diesem Kapitel mögliche Erweiterungen untersucht werden. Als Erstes soll betrachtet werden, wie weitere Informationssysteme in die Änderungspropagation eingebunden werden können. Dadurch können dem Zielsystem Daten bereitgestellt werden, die so vom Quellsystem nicht bereitgestellt werden können. Ein Propagationssystem kontaktiert dafür weitere Informationssysteme (Drittsysteme), die ihm diese Daten bereitstellen. Bisher ist man davon ausgegangen, dass nicht mehr als eine empfangene Änderungsbeschreibung von einem Propagationsprozess verarbeitet wird. In diesem Kapitel soll nun untersucht werden, wie mehrere aufgetretene Änderungen zusammen in einem Prozess verarbeitet werden können. Diese Art von Abhängigkeit wird M-zu-N-Abhängigkeit genannt. Hierfür werden mögliche Anwendungsszenarien identifiziert. Wie schon erwähnt wurde, ist ein Nachteil der Hub-and-Spoke-Architektur, dass das Integrationssystem einen Flaschenhals darstellt. Um dieses Problem zu beheben, soll untersucht werden, wie der Propagationsmanager verteilt werden kann.

In Abbildung 4.1 ist die Architektur der in diesem Kapitel besprochenen Erweiterungen des Gesamtsystems illustriert. Der Warteschlangenmanager und der Repository-Server bleiben von den Erweiterungen weitestgehend unberührt. Dem Repository-Server müssen nur eventuelle Sprachänderungen von XPDL bekannt gemacht werden, so dass er erweiterte Propagationsskripte auf ihre Gültigkeit überprüfen kann. Betroffen von den Erweiterungen sind vor allem der Propagationsmanager und der Abhängigkeitsmanager. Deren im vorigen Kapitel besprochene Kernfunktionalität wird durch die angesprochenen Module erweitert: Die Einbindung von Drittsystemen und die Realiserung von M-zu-N-Abhängigkeiten. Die Erweiterung des Abhängigkeitsmanagers steht dabei nicht im Fokus dieses Kapitels. Die Verteilung des Propagationsmanagers ist durch mehrere Propagationsmanager symbolisiert, die sich die Aufgabe der Änderungspropagation teilen.

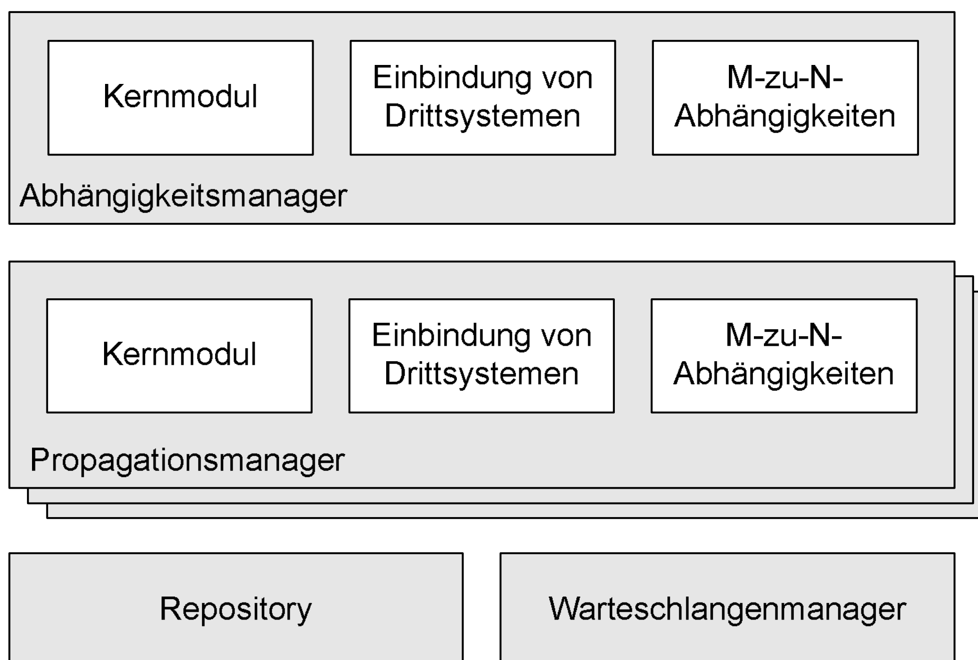


Abbildung 4.1: Erweiterung des Propagationssystems

4.1 Einbindung von Daten aus Drittsystemen

In diesem Abschnitt wird die Erweiterung des Propagationssystems durch die Einbindung von Daten in Änderungsbeschreibungen beschrieben, die so nicht von den Quellsystemen und Zielsystemen bereitgestellt werden können. Als Erstes wird die Problemstellung diskutiert, um dann auf die Verwendung eines Datendienstes einzugehen. Ein Datendienst ist der Dienst eines Drittsystems, der seine Daten bereitstellt. Weiterhin wird auf die Zugriffsarten eingegangen, mit denen auf die Daten eines solchen Datendienstes zugegriffen werden kann. Als Nächstes werden die Schritte von der notwendigen Beschreibung eines Datendienstes bis zu seiner Nutzung untersucht. Um Dienste zu beschreiben, wird eine Sprache entwickelt: die *Data Service Description Language* (DSDL). Bevor ein abschließendes Beispiel betrachtet wird, wird die Realisierung diskutiert.

4.1.1 Problemstellung

Im heterogenen Umfeld reicht die einfache Integration von Datenänderungen in vielen Fällen nicht aus. Dieses Problem ist in Abbildung 4.2 dargestellt. In diesem Beispiel wird eine Integration eines Fabriklayoutplanungswerkzeugs (Quellsystem) mit einem Digitale-Fabrik-System (Zielsystem) betrachtet. Im Quellsystem wird nun eine neue Maschine (eine Fertigungsressource) positioniert. Dies soll vom Quell- zum Zielsystem propagiert werden. Das Zielsystem benötigt für seine Änderungsbeschreibung (AB_{ZS}) noch zusätzliche Daten (O_{DP}), die nicht von der Änderungsbeschreibung des Quellsystems (AB_{QS}) bereitgestellt werden können. In diesem Fall werden die Attribute

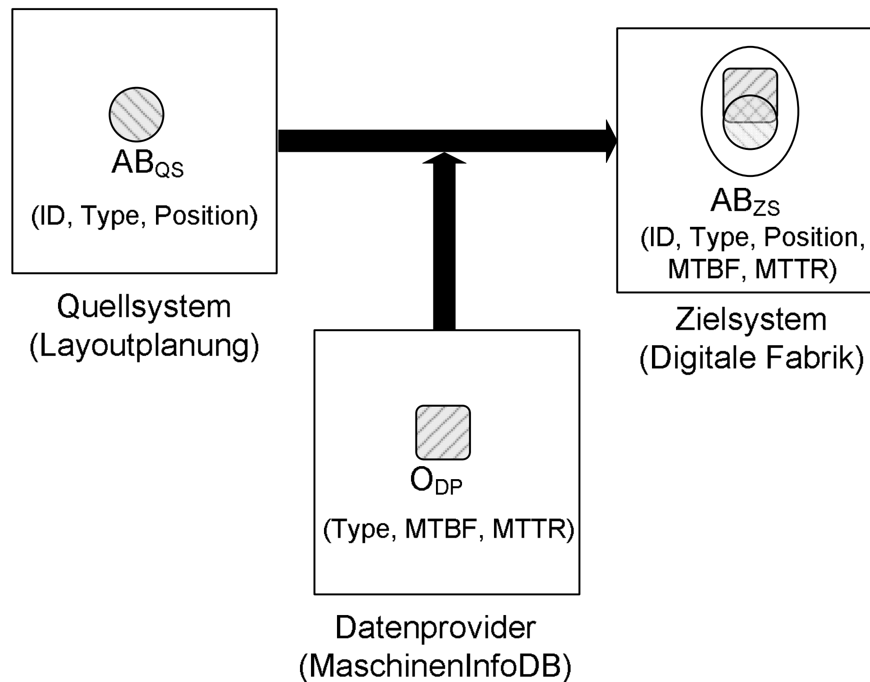


Abbildung 4.2: Problem der Einbindung von Daten von Drittsystemen

MTBF (meantime between failure) und MTTR (meantime to repair) benötigt. Da diese von dem Fabriklayoutplanungswerkzeug nicht bereitgestellt werden, müssen sie von einem dritten Informationssystem (hier: MaschinenInfoDB) geliefert werden. Diese Art von Informationssystem repräsentiert einen sogenannten Datenprovider. Das Drittsystem kann sich innerhalb des Unternehmens befinden oder ein Service einer externen Firma sein. Beispielsweise kann der Hersteller verwendeter Bearbeitungsmaschinen (z.B. Fräsmaschinen) Informationen über sie bereitstellen, die dann abgefragt werden können. Intern können zum Beispiel Datenbanken eingebunden werden, die bestimmte Informationen bereitstellen (z.B. eine Kundendatenbank).

Außerdem können mit dieser Einbindung von Drittsystemen Unterschiede in der Datenrepräsentation zwischen Quell- und Zielsystemen überwunden werden, indem sogenannte Mapping-Tabellen zur Anwendung kommen. Ein solcher Unterschied könnte beispielsweise die Länderrepräsentation in einer Adresse sein, wobei in einem System ein Ländernamen (z.B. Deutschland) und in einem anderen ein Ländercode (z.B. D) verwendet wird. Diese werden als Tupel in der Mapping-Tabelle gespeichert.

4.1.2 Verwendung eines Datendienstes

Der Datenprovider stellt durch einen Datendienst Daten bereit. Damit das Propagationssystem den Datendienst nutzen kann, stellt er eine Beschreibung seiner Metadaten im Propagationsrepository bereit. Dadurch wird es dem Abhängigkeitsmanager ermöglicht, den Anforderungen entsprechend nach einem Datendienst zu suchen.

Von der Definition der Beschreibung bis zur eigentlichen Nutzung des Datendiens-

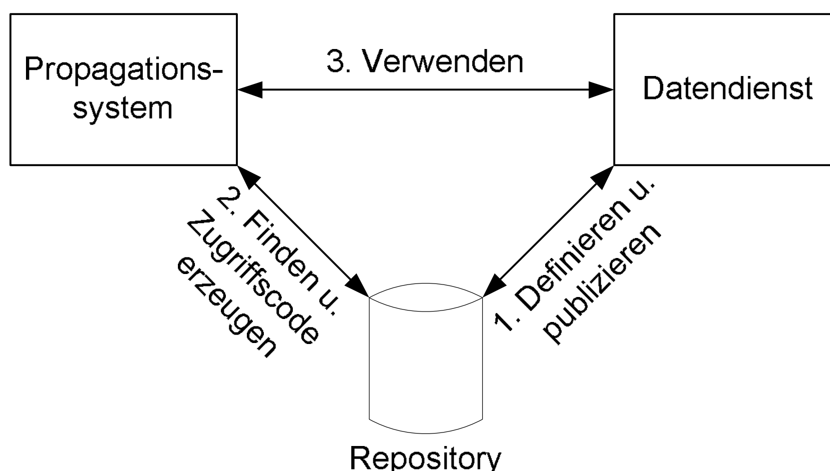


Abbildung 4.3: Übersicht über die Verwendung eines Datendienstes

tes werden drei Schritte benötigt, wie in Abbildung 4.3 dargestellt. In einem ersten Schritt wird die Beschreibung des Datendienstes erzeugt und publiziert. Die Beschreibung wird durch die im Rahmen dieser Arbeit entwickelte *Data Service Description Language* (DSDL) ermöglicht. Diese Beschreibungen werden im zentralen Propagationsrepository abgelegt. Zu einem späteren Zeitpunkt (Schritt 2) wird dann über den Abhängigkeitsmanager nach den Beschreibungen gesucht, damit die benötigten Datenprovider gefunden werden. Die Beschreibungen werden vom Abhängigkeitsmanager verwendet, um den entsprechenden Zugriffscode zu erzeugen und in bestehende Propagationsskripte einzubinden. Im letzten Schritt wird schließlich der erzeugte Zugriffscode verwendet, um auf den Datendienst zuzugreifen. Die Kommunikation mit dem Datendienst erfolgt über SOAP. SOAP ermöglicht den XML-basierten Austausch von Nachrichten.

In Abbildung 4.3 ist auch die Verwandtschaft zum Webservice (vgl. [ACKM04] Seite 146) zu erkennen. Allerdings sind Webservices eher gedacht um Dienste aufzurufen (gekapselter Zugriff durch einen Dienst) als direkt auf Daten eines Systems zuzugreifen. Sie definieren die Struktur von Nachrichten, die ausgetauscht werden, und weniger die internen Datenstrukturen des Dienstanbieters (z.B. relationale Tabellen) auf die zugegriffen werden kann. Bei Webservices wird ebenfalls ein Dreigestirn verwendet: ein *Service-Nutzer* (hier: Propagationsmanager) fragt beim *Service-Verzeichnis-Anbieter* (hier: Propagationsrepository) nach einem *Service-Anbieter* (hier: Datendienst). Dabei werden im Fall von Webservices folgende Technologien verwendet: WSDL (*Webservice Description Language*) zur Beschreibung des Service, UDDI (*Universal Description Discovery and Integration*) als Service-Verzeichnis-Anbieter und SOAP als Kommunikationsprotokoll. Allerdings wird bei Webservices SOAP auch zur Kommunikation mit dem UDDI verwendet. Dies erfolgt im Fall des Datendienstes für die Propagation durch die API des Repositorys. Da bis jetzt der einzige Datendienst-Verwender der Propagationsmanager ist, eignet sich besonders das Propagationsrepository, um die Beschreibungen der Datendienste abzulegen. Dies ist begründet in der zentralen Ablage aller propagationsrelevanten Metadaten.

4.1.3 Zugriffsarten

Ein Datenprovider stellt seine Daten über einen Datendienst bereit. Um möglichst flexibel zu sein, sollten verschiedene Zugriffsarten unterstützt werden. Beispielsweise sollte es möglich sein, auf Daten in relationalen Datenbanken zuzugreifen. In dem hier vorgestellten Ansatz werden eine Reihe von Zugriffsarten unterstützt, die aber durch weitere ergänzt werden können.

Standardmäßig werden vom Propagationssystem drei Arten unterstützt: SQL für relationale Datenbanken, XQuery für XML-Daten in XML-Dokumenten sowie XML-Datenbanken und SOAP-RPC (*Remote Procedure Call*). Um eine einheitliche Form des Zugriffs zu gewährleisten, sollte jede Zugriffsart den Zugriff über eine Datenanforderung ermöglichen, die eine Menge von Parametern hat. Das trifft bei einem RPC-Aufruf zu, weil dieser wie ein lokaler Prozeduraufruf Parameter haben kann. Aus diesem Grund wird von Webservices nur der RPC-Mode unterstützt. Verwendet man für SQL und XQuery parametrisierte Anfragen, so trifft dies auch für diese Zugriffsarten zu. Durch diesen Ansatz können dann zur Laufzeit die Parameter an die Daten der Änderungsbeschreibungen gebunden werden.

Falls diese Zugriffsarten nicht ausreichend sind, da beispielsweise objektorientierte Datenbanken unterstützt werden sollen, so soll das Propagationssystem um weitere Zugriffsarten ergänzt werden können. Dafür muss die DSDL-Sprache zuerst um Elemente erweitert werden, die für die Zugriffsart spezifisch sind. Da die DSDL-Dokumente als Ganzes im Repository abgelegt werden, müssen keine Änderungen im Propagationsrepository gemacht werden, außer der Integration des neuen DSDL-Schemas in das Repository, damit die DSDL-Dokumente entsprechend validiert werden können. Die Implementierung des entsprechenden XPDL-Befehls zum Aufruf des Datendienstes muss im Propagationssystem um die neue Zugriffsart erweitert werden. Dies erfolgt durch eine Plugin-Architektur.

4.1.4 Von der Definition zur Nutzung eines Datendienstes

Nachdem die Zugriffsarten für den Zugriff auf Datenprovider diskutiert wurden, soll nun der Prozess zur Definition und Nutzung eines Datendienstes beschrieben werden. Der Prozess unterteilt sich in zwei Bereiche (Abbildung 4.4): Entwicklungs- und Laufzeit. Zur Entwicklungszeit wird der Datendienst definiert (Schritt 1) und zu einem späteren Zeitpunkt wird der Zugriffscode generiert (Schritt 2). Zur Laufzeit wird schließlich der Datendienst genutzt, um Daten anzufordern, die in einem Propagationsprozess eine Änderungsbeschreibung mit zusätzlichen Informationen anreichern (Schritt 3).

Im ersten Schritt in Abbildung 4.4 wird zuerst die Beschreibung des Datendienstes definiert, die dann im zweiten Schritt verwendet wird, um den Zugriffscode zu generieren. Dafür werden zunächst Metadaten des Dienstes gesammelt, wie zum Beispiel das System, welches den Dienst implementiert, und die Zugriffsart, welche verwendet wird. Die nachfolgende Definition ist von der gewählten Zugriffsart abhängig. Wird ein RPC-Zugriff definiert, so wird dieser durch eine WSDL-Message beschrieben. WSDL stellt dabei Elemente für die Beschreibung der RPC-Nachrichten (Request und Response) bereit. Bei SQL wird der Zugriff in Form von Tabellenbeschreibungen und pa-

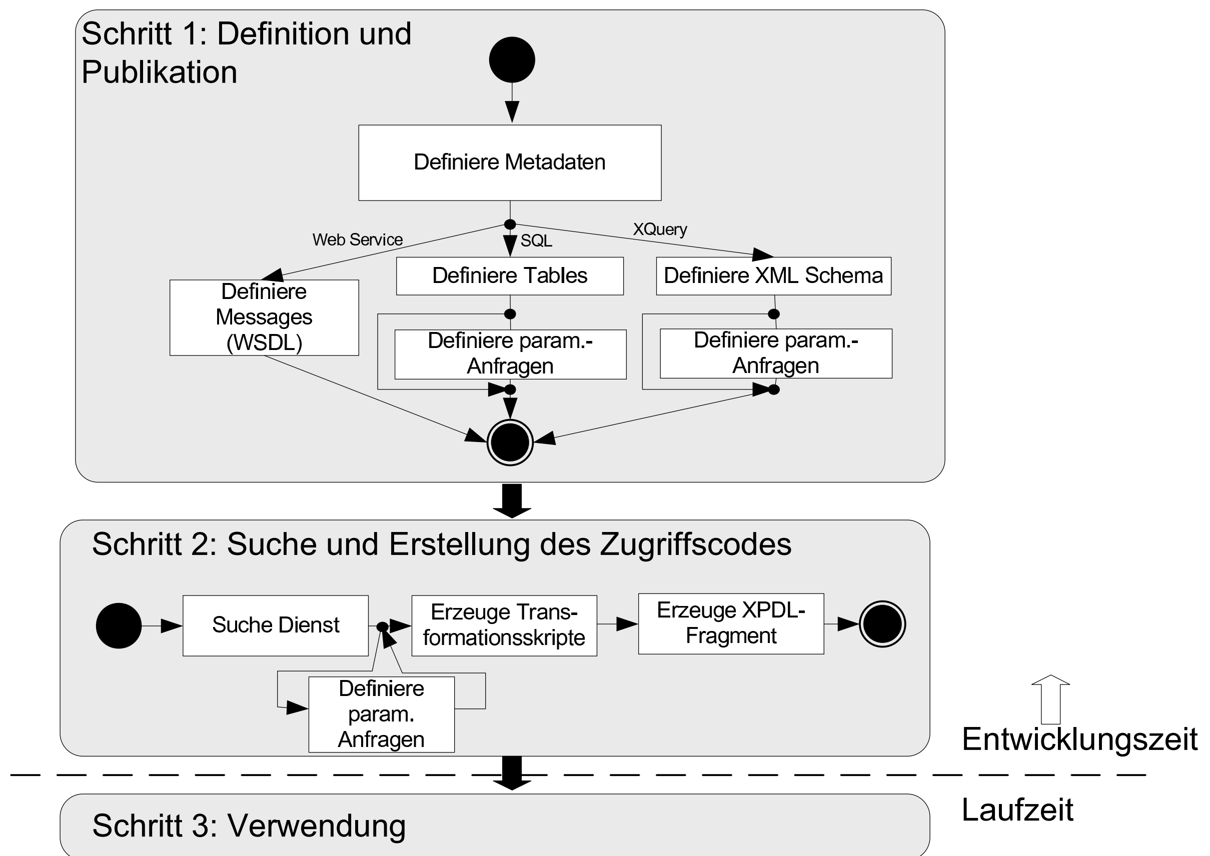


Abbildung 4.4: Prozess der Definition und Nutzung eines Datendienstes

rametrisierten Anfragen beschrieben. Die Tabellenbeschreibungen werden verwendet, um parametrisierte Anfragen durch spezialisierte Werkzeuge zu erzeugen. Die parametrisierten Anfragen können im ersten Schritt (Early Definition) oder erst im zweiten Schritt definiert werden (Late Definition). Für XML-Daten werden die Zugriffsmöglichkeiten durch die Strukturbeschreibung in Form von XML Schema angegeben. Ebenso wie bei SQL können parametrisierte Anfragen definiert werden, die aber auf XQuery basieren.

Die Struktur der Antwort wird anhand von XML Schema spezifiziert. Dies gilt auch für den Webservice-Zugriff, da seit WSDL 2.0 [Wor07] eine WSDL-Message in XML Schema beschrieben wird. Des Weiteren wird XML Schema schon zur Überprüfung der Zustände der Änderungsbeschreibungen verwendet und ermöglicht dadurch eine gute Integration in das Propagationssystem. Weiterhin unterstützen Mapping-Werkzeuge (z.B. Altova Mapforce) XML Schema für die Definition von Input und Output der Transformationen. Die Struktur bei den Zugriffsarten SQL und XQuery kann allerdings erst bei der Definition einer parametrisierten Anfrage erfolgen, da vorher die Antwortstruktur noch nicht feststeht. Die Antwortstruktur bei XQuery ist durch XQuery selbst bestimmt. Bei SQL gibt es mehrere Verfahren, wie Anfrageergebnisse in XML transformiert werden. Ein Beispiel hierfür ist der SQL/XML-Standard [EM02].

Die Zugriffscode-Erstellung in Schritt 2 (siehe Abbildung 4.4) wird gestartet, indem nach dem benötigten Datendienst gesucht wird. Die Suche nach Datendiensten kann durch eine Volltextsuche über Beschreibungstexte erfolgen oder durch eine strukturierte Suche über eine strukturierte Beschreibung der angebotenen Daten. Werden die Daten über die Zugriffsarten SQL oder XQuery angefragt, so muss nach einer passenden parametrisierten Anfrage gesucht werden. Ist diese nicht vorhanden, werden die Tabellenstrukturen bzw. XML Schemas für XML als Input zur Generierung einer parametrisierten Anfrage verwendet. In den nächsten zwei Schritten werden dann die Zugriffsmethoden in Form von Transformationsskripten und von einem Aufrufsfragment in XPDL erzeugt. Die Transformationsskripte erzeugen beispielsweise SOAP-Header für zusätzliche Informationen (z.B. Daten für die Authentizierung) und das Aufrufsfragment steuert dabei den Aufruf des Datendienstes.

Der generierte Zugriffscode von Schritt 2 wird schließlich in Schritt 3, bei Zugriff auf den Datendienst, verwendet. Dabei wird das Propagationsskript mit dem erzeugten Fragment ausgeführt.

4.1.5 Dienstbeschreibung

Um Schritt 1 und 2 des Erstellungsprozesses miteinander zu integrieren, müssen die Datendienste beschrieben werden. Des Weiteren müssen dem Propagationssystem Daten über den Datendienst zur Laufzeit bereitgestellt werden, was in Abschnitt 4.1.6.2 noch genauer erläutert wird. Dafür wurde eine Sprache entwickelt, die *Data Service Definition Language* (DSDL) heißt. Die Struktur dieser Sprache wird in Abbildung 4.5 in UML dargestellt. Die Sprache selbst ist in XML implementiert. Die im vorigen Abschnitt beschriebenen Metadaten des Dienstes werden direkt im DataService-Element beschrieben und bestehen unter anderem aus dem Namen des Datendienstes

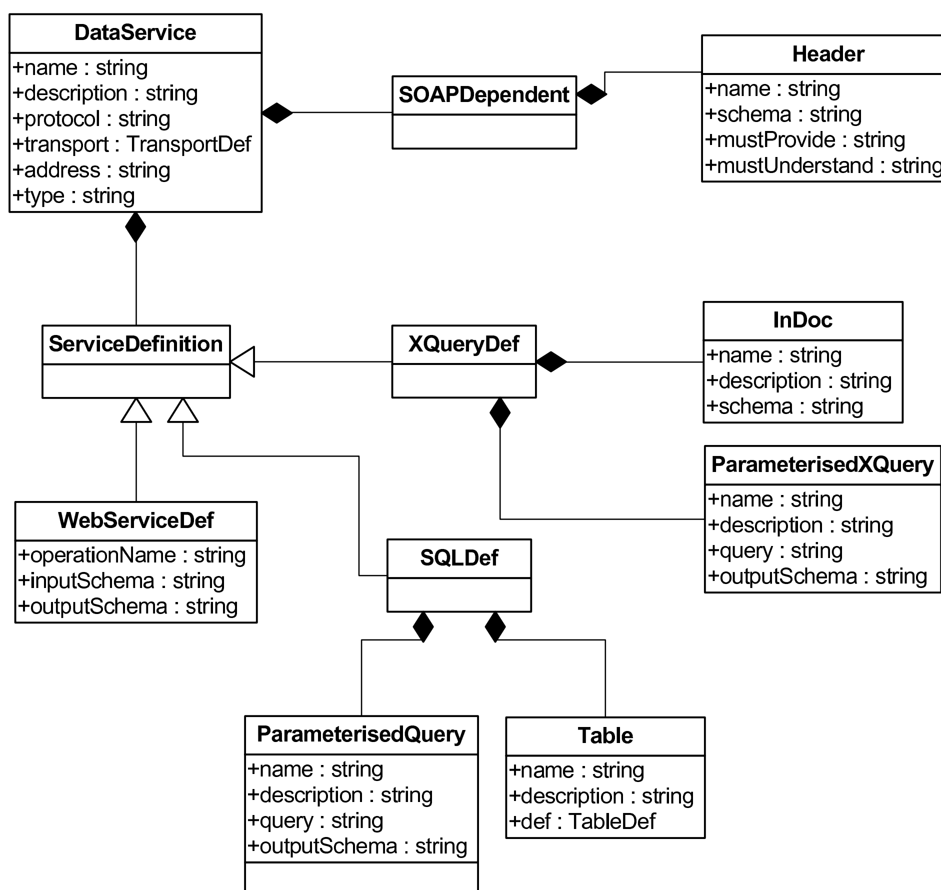


Abbildung 4.5: Beschreibungssprache für Datendienste

sowie dessen Beschreibung. Das Attribute *protocol* gibt an, welches Protokoll für die Kommunikation verwendet wird und ist standardmäßig auf SOAP gesetzt. Für den Transport sind persistente Warteschlangen und HTTP vorgesehen. Die persistenten Warteschlangen dienen zur Kommunikation mit den Informationssystemen, die idealerweise an das Propagationssystem angebunden sind. Dadurch muss das Informationssystem nur die vorhandene Schnittstelle erweitern und nicht eine neue Implementierung auf einer komplett anderen Technologie vorsehen. HTTP eignet sich besonders für unternehmensexterne Systeme, da mit HTTP das Firewall-Problem leichter gelöst werden kann, da der Standard-HTTP-Port oft in Firewalls frei geschaltet ist. Mit dem `type`-Attribut wird die Zugriffsart festgelegt, die für die nachfolgende Dienstbeschreibung ausschlaggebend ist. Unabhängig von der Zugriffsart, aber protokoll-abhängig, sind die sogenannten SOAP-Header. Diese können Informationen enthalten, die nicht über den Nachrichtenkörper (`body`) verschickt werden sollen, da sie zusätzliche Informationen, wie zum Beispiel Authentisierungsinformationen, darstellen.

Mittels der `WebserviceDef` wird die Zugriffsstruktur eines WS-RPC-Aufrufs definiert. Der Aufruf und die Antwort werden durch WSDL-Messages definiert, wobei diese die Parameter und die Rückgabe des RPC-Aufrufs definieren. Dabei verweist die `WebserviceDef` auf den Namen des XML Schemas. Der Name wird verwendet, um

das Schema aus dem Repository zu laden. Wie schon angemerkt, wird in WSDL 2.0 [Wor07] XML Schema zur Definition von Messages verwendet.

Die Beschreibung eines SQL-Datendienstes ist in zwei Bereiche unterteilt. Im ersten werden Tabellen oder Views definiert, auf die mittels Anfragen zugegriffen werden kann. Darauf aufbauend können parametrisierte Anfragen definiert werden, die eine bestimmte Datenanfrage erfüllen. Die Selektivität kann mittels der Parameter bestimmt werden. Für die Implementierung der parametrisierten Anfrage wird ein XML-Element definiert, das sogenannten „Mixed Content“ verwendet, d.h. Textknoten und Elemente wechseln sich ab. Dabei sind die SQL-Teile durch Textknoten repräsentiert und die Parameter durch Elemente, wobei ein Parameterelement genau an der Stelle steht, an der der Wert des Parameters eingefügt werden soll (z.B. ... `WHERE ID=<Parameter Name='ID'>`).

Der XQuery-Zugriff wird durch XML Schema und parametrisierte Anfragen definiert (vgl. Abbildung 4.5).

Für die Beschreibung des Datendienstes wurde nicht die bereits existierende Beschreibungssprache für Webservices verwendet, da diese zwar den Austausch der Nachrichten zwischen Klienten und dem Webservice beschreibt, nicht aber deren interne Strukturen wie Tabellen sowie Views für relationale Datenbanken und XML Schema für XML-Dokumente bzw. -Datenbanken bereitstellt. Da allerdings auch Webservices im RPC-Mode unterstützt werden sollen, wurden WSDL-Messages in die Datendienst-Sprache integriert, die jedoch ebenfalls in XML Schema beschrieben werden (WSDL 2.0).

4.1.6 Realisierung

In diesem Abschnitt wird die Realisierung der Datendienstunterstützung innerhalb des Propagationssystem diskutiert. Die Möglichkeiten zur Parameterbindung stellen dabei die Grundlage für die Realisierung von XPDL-Befehlen, die den Sprachumfang erweitern. Diese werden anschließend untersucht.

4.1.6.1 Parameterbindung

Um die Parameter zu binden und die SOAP-Anforderung zu erzeugen, wurden vier Ansätze identifiziert:

- Stub-Ansatz
  Der Zugriff auf Webservices wird oft durch sogenannte Stubs realisiert, sofern es sich um die RPC-Zugriffsart handelt. So ist es möglich die gesamte Komplexität des Aufrufs hinter einer Methode zu verstecken. Der Aufruf des RPCs erscheint für den Programmierer wie ein lokaler Aufruf. Der Stub wird dabei aus der Beschreibung des Webservices generiert, d.h. aus der WSDL-Datei.

  Ein ähnlicher Ansatz wäre ebenfalls für den Zugriff auf einen Datendienst denkbar, denn es existiert die WS-RPC-Zugriffsart, die Parameter wie eine Methode hat. Für die beiden anderen Zugriffsarten, SQL und XQuery, werden parametrisierte Anfragen verwendet. Daraus folgt, dass der Aufruf des Datendienstes auch

durch einen Stub erfolgen kann. Bei einem nachrichtenorientierten System, wie bei einem Datenpropagationssystem, ist der Stub-Ansatz nicht sinnvoll, da zur Entwicklungszeit Stubs erzeugt werden müssen, die aber schwer in ein nachrichtenorientiertes System einbindbar wären, da sie für ein anderes Programmierparadigma ausgelegt sind.

- Einfacher Transformationsansatz
  Als zweiten Ansatz könnte man Transformationen verwenden, um aus Änderungsbeschreibungen direkt die SOAP-Nachrichten für den Datendienst zu erzeugen. Die Antworten des Datendienstes in Form von SOAP-Nachrichten könnten dann mit der jeweiligen Änderungsbeschreibung über weitere Transformationen integriert werden. Dieser Ansatz ist allerdings sehr fehleranfällig, da die gesamte SOAP-Nachricht mittels Transformation erzeugt werden müsste und bei der Erstellung der dafür benötigten Transformationsskripte Fehler gemacht werden können.

- Transformationsansatz mit Schichten
  Um dies zu verbessern wurden in unserem ersten Ansatz[HCM05] mehrere Schichten eingeführt. Zuerst wird die eigentliche Datenanforderung mittels Transformation erzeugt und in der nächsten Schicht (SOAP-Schicht) in eine SOAP-Nachricht verpackt. In der darauffolgenden Schicht (Transportschicht) wird die SOAP-Nachricht verschickt und die Antwort empfangen. Die empfangene Antwort wird in der SOAP-Schicht analysiert und die extrahierten Daten können dann mittels Transformation mit einer Änderungsbeschreibung integriert werden. Dieser Ansatz ist nicht so fehleranfällig wie der reine Transformationsansatz. Allerdings können wegen der Erzeugung von Anforderungen (*Requests*) durch Transformationen immer noch Fehler auftreten. Diese Anfälligkeit kann weiter durch die Verwendung von speziellen Werkzeugen reduziert werden. Ein Nachteil dieses Ansatzes ist die Einführung von drei neuen XPDL-Befehlen. Dies führt dazu, dass der Ansatz für die Entwicklung von Propagationsskripten komplexer ausfällt.

- Paramaterbindungansatz zur Laufzeit
  Um dieses Manko zu beseitigen, wurde noch ein weiterer Ansatz entworfen, der soweit es geht auf Transformationen verzichtet und statt dessen Parameter zur Laufzeit bindet. Die Parameter repräsentieren die Parameter des RPC-Aufrufs oder der parameterisierten Anfrage von SQL oder XQuery. Um dies zu realisieren braucht man einen Befehl, der eine beliebige Anzahl von Parametern als Input nimmt und überprüft, ob diese vom entsprechenden RPC bzw. der parametrisierten Anfrage unterstützt werden. Der Befehl erzeugt dann dynamisch die SOAP-Nachricht und verarbeitet auch wieder die Antwort. Um die SOAP-Nachricht zu erzeugen, werden zur Laufzeit das DSDL-Dokument analysiert und die Parameter gebunden. Die Parameter können mit sogenannten PCL-Ausdrücken (vgl. Abschnitt 3.6.1) gesetzt werden. Falls PCL nicht ausreicht, kann man zuerst die Änderungsbeschreibung transformieren und dann den Parameter extrahieren. Der Ansatz benötigt nur einen neuen Befehl und ist nicht so fehleranfällig

wie die anderen zwei. Aus diesen Gründen wurde er für das Propagationssystem ausgewählt.

4.1.6.2 XPDL-Befehle

4.1.6.2.1 Erzeugung der Header und ggf. Parametervorbereitung. Um einen SOAP-Header zu erzeugen, wird ein `transform`-Befehl verwendet, wie er in Abschnitt 3.5.3.1 eingeführt wurde. Dieser hat folgendes Aussehen:

```
transform(in, out, script, reduceTo?, parameter*).
```

Wichtig dabei ist das reduceTo-Argument, um eine Update-Änderung mit zwei Zuständen auf einen Zustand zu reduzieren. Der `transform`-Befehl kann außerdem für die Parametervorbereitung eingesetzt werden, falls PCL für die Parameterzuordnung nicht ausreichend ist.

4.1.6.2.2 Aufruf eines Datendienstes. Nachdem die SOAP-Header und evtl. auch Parameter vorbereitet worden sind, kann der Data Service aufgerufen werden. Dies erfolgt mit folgendem Befehl:

```
call_data_service(name, partName, out, error_handler?, header*,
parameter*)

header(in, mustUnderstand)

parameter(name, in, expression)
```

Der Aufruf des Datendienstes benötigt zuerst den Namen des Datendienstes, um die entsprechende DSDL-Beschreibung im Repository zu finden. Damit die RPC-Prozedur oder die parametrisierte Anfrage gefunden werden kann, wird der `partName` verwendet. Das Ergebnis der Anfrage der analysierten Antwort wird dann unter dem Namen `out` zugänglich gemacht. Die optionale Angabe einer Fehlerauswertungskomponente ermöglicht die entsprechende Reaktion auf Fehler, die im Datendienst aufgetreten sind (vgl. Abschnitt 4.1.6.3). Mit `header`-Elementen können eine beliebige Anzahl von SOAP-Header angegeben werden. Der Inhalt des Header ist über die Änderungsbeschreibung `in` definiert. Das Attribut `mustUnderstand` gibt an, ob der Datendienst den Header verstehen und andernfalls die Bearbeitung verweigern muss. Es ist ein Standardattribut von SOAP. Die Parameter werden schließlich mit `parameter` angegeben, wobei der Name des Parameters mit dem in der DSDL-Beschreibung übereinstimmen muss. Die Nachricht, die den Parameterwert liefert, wird über `in` bestimmt und der Wert wird über den PCL-Ausdruck (siehe Abschnitt 3.6.1) `expression` extrahiert.

Der Befehl bekommt die Kontaktdaten des Datendienstes aus dessen DSDL-Beschreibung und muss deswegen im Befehl nicht spezifiziert werden.

4.1.6.2.3 Integration. Im letzten Schritt müssen noch die Änderungsbeschreibungen AB_{QS} (vgl. Abbildung 4.2) mit den angeforderten Daten integriert werden. Dies erfolgt mittels Transformationsskripten. Dabei können die zusätzlichen Daten mit der XPath-Bibliothek, die in Abschnitt 3.6.2 beschrieben wurde, importiert werden. Allerdings hat das jeweilige Transformationsskript dann mehrere XML Schemas als Input. Die Zuordnung von Schemas zu Transformationsskripten muss im Repository hinterlegt sein. Gründe hierfür sind vor allem die Konsistenz des Repository und das Erstellen von Transformationsskripten zum Beispiel mit Mapping-Werkzeugen.

4.1.6.3 Fehlerbehandlung

Bei der Anforderung von zusätzlichen Daten können Fehler auftreten, die behandelt werden müssen. Fehler können dabei im Datendienst auftreten, welche dann in einem sogenannten SOAP-Fault an das Propagationssystem zurückgegeben werden. Des Weiteren können Fehler bei der Kommunikation mit dem Datendienst auftreten, die etwa durch eine Zeitüberschreitung erkannt werden oder direkt durch das entsprechende Kommunikationssystem. Bei der Zeitüberschreitung kann der Programmierer entscheiden, ob abgebrochen werden soll oder ob die Ausführung des Propagationsskripts fortgesetzt werden soll. Letzteres kann beispielsweise dann gewählt werden, wenn die zusätzlichen Daten optional sind. Durch die XPath-Bibliothek (vgl. Abschnitt 3.6.2) kann das Vorhandensein von zusätzlichen Daten abgefragt und entsprechend reagiert werden. Tritt ein Fehler durch eine Zeitüberschreitung auf, kann evtl. auch ein zweiter Datendienst einbezogen werden.

Wird ein Fehler im Kommunikationssystem erkannt, so wird der Propagationsprozess wiederholt. Erst nach einer bestimmten Anzahl von Wiederholungsversuchen wird der Propagationsprozess abgebrochen und der Fehler protokolliert.

Tritt ein Fehler im Datendienst auf, welcher mit einem SOAP-Fault zurück an das Propagationssystem gesendet wird, kann der Ausgang des Propagationsprozesses durch ein optionales Fehlermodul (Handler) (vgl. Abschnitt 4.1.6.2) bestimmt werden. Es gibt dabei drei Möglichkeiten:

- Prozess abbrechen
- Prozess wiederholen
- Fehler ignorieren und Prozess fortsetzen.

Die ersten zwei entsprechen der Reaktion auf sich wiederholende und nicht wiederholende Fehler. Die dritte dagegen ermöglicht, wie oben, eine unvollständige aber dafür zeitnahe Version an das Zielsystem zu schicken oder gegebenenfalls einen anderen Datendienst einzuschalten.

Wird dagegen kein Fehlermodul (Handler) angegeben, kann nur eine Standardreaktion erfolgen, die unabhängig vom Fehler ist. Es wurde hierbei die Variante gewählt, bei der der Prozess wiederholt wird. Dies kann zwar bei einem sich wiederholenden Fehler zu einer unnötigen Systemlast führen, dafür wird alles versucht, um die Propagation dennoch erfolgreich abzuschließen.

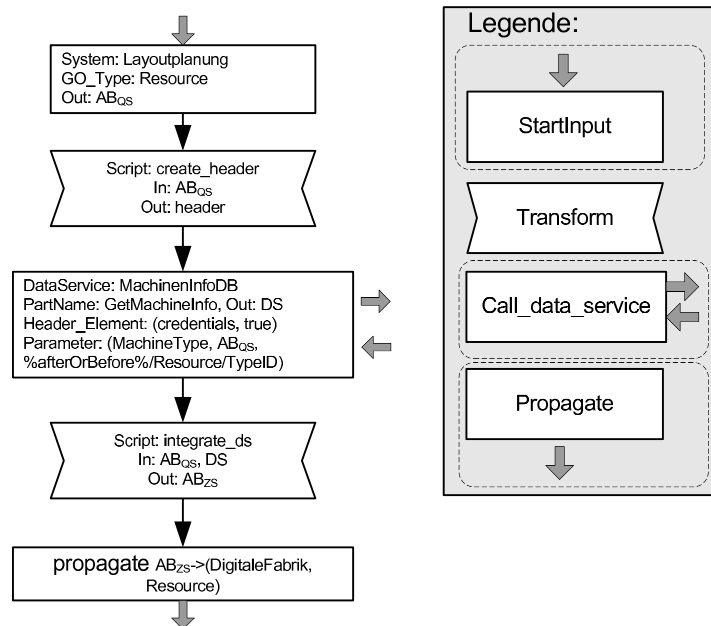


Abbildung 4.6: Beispiel für ein Propagationsskript mit der Integration eines Datendienstes

4.1.7 Beispiel

Als Beispiel wird das Integrationsszenario von Abbildung 4.2 wieder aufgegriffen. Bei diesem Szenario wird ein Layoutplanungssystem mit einem Digitale-Fabrik-System integriert, wobei eine neue Maschine (Ressource) vom Layoutplanungssystem zum Digitale-Fabrik-System propagiert wird. Außerdem wird die propagierte Ressource mit Daten aus der MaschinenInfoDB angereichert.

In Abbildung 4.6 ist das dazugehörige Propagationsskript in der Kontrollflussansicht (vgl. Abschnitt 3.7.3.2) dargestellt. Zuerst wird die getätigte Änderung (Neue Maschine) empfangen. Daraufhin werden die Zugangsdaten für den Datendienst in Form eines SOAP-Header erzeugt. Dieser wird im nächsten Schritt (Aufruf des Datendienstes) in die Datenanforderung integriert. In diesem Schritt wird außerdem noch die parametrisierte Anfrage aus Abbildung 4.7 aus der Datendienstbeschreibung extrahiert und die Parameter werden durch Werte aus der Änderungsbeschreibung AB_{QS} ersetzt. In diesem Fall gibt es einen Parameter 'MachineType', der durch den Wert in dem Änderungszustand '/Resource/TypeID' ersetzt wird, wobei dem Danach-Zustand Vorrang gewährt wird (vgl. Abschnitt 3.6.1).

4.2 Verarbeitung mehrerer Änderungen

Nachdem die Einbindung von Daten aus Drittsystemen diskutiert wurde, wird in diesem Abschnitt der Einsatz und Realisierung von M-zu-N-Abhängigkeiten diskutiert.

```
<Query>
    SELECT XMLELEMENT(NAME ‘MachineInfo’, XMLFORREST(
        MTBF AS ‘MTBF’, MTTR AS ‘MTTR’))
     FROM MachineInfoTable
     WHERE Type=<Parameter name="MachineType" type="Integer"/>
```

Abbildung 4.7: Beispiel einer parametrisierten Anfrage in DSDL-Beschreibung

4.2.1 Problemstellung

Bei den hier als M-zu-N-Abhängigkeiten bezeichneten Abhängigkeiten handelt es sich um Konstrukte, bei denen mehrere Änderungsbeschreibungen (M, mindestens 2) empfangen, verarbeitet und zu mindestens einem System (N) gesendet werden. Nun stellt sich die Frage, was man mit solchen M-zu-N-Abhängigkeiten realisieren kann. Bei einfachen Änderungen werden geänderte Daten propagiert, so dass ein Zielsystem seine Daten ebenfalls anpassen kann. Beim Empfang und Verarbeitung mehrerer Änderungen durch einen Propagationsprozess ist das nicht mehr der Fall. Ergebnisse von M-zu-N-Propagationsprozessen sind erst spät sichtbar, d.h. wenn die letzte erwartete Änderung empfangen wurde. Außerdem eröffnen sich durch die Beziehung der einzelnen Änderungsbeschreibungen neue Anwendungsmöglichkeiten, die über die reine Änderungspropagation hinausgehen.

Mittels M-zu-N-Abhängigkeiten können Geschäftsregeln implementiert werden, wie zum Beispiel: „Wenn der **neue Kunde** innerhalb von **24 Stunden** eine **neue Bestellung** aufgibt, so bekommt er einen **Rabatt von 25 %**“. Der *neue Kunde* ist das auslösende Änderungsereignis und die *neue Bestellung* ist das zweite erwartete Ereignis, das auftreten soll. Beide Ereignisse sollen maximal *24 Stunden* auseinanderliegen. Wenn diese Bedingungen erfüllt sind, so soll eine Änderungsbeschreibung *neuer Rabatt* an ein Zielsystem versendet werden.

Abstrakt gesehen ist eine M-zu-N-Abhängigkeit eine Art von Prozessmodell, wie in Abbildung 4.8 dargestellt. Nach dem Empfang einer Startänderungsbeschreibung werden k Unterprozesse gestartet. Der Wert k ist eine Steuerungsgröße, die unabhängig von M sowie N ist und die Anzahl von möglichen Prozessen bei gleicher Startänderungsbeschreibung angibt. Dadurch kann das obige Beispiel bei gleichem Kunden auf mehrere Bestellungen ausgeweitet werden, d.h. dem Kunden wird bei k gleich fünf Rabatt auf fünf Bestellungen gewährt. Dieser Wert wird innerhalb des Propagationsskripts festgelegt. Danach werden noch n-1 Änderungsbeschreibungen erwartet, sodass insgesamt n Änderungsbeschreibungen empfangen wurden. Die Auswahl der zu empfangenden Änderungsbeschreibungen erfolgt durch die Angabe eines System-GOTyp-Paares (Informationsystem und dessen Geschäftsobjekttyps) und eines optionalen PCL-Ausdrucks. Mithilfe des PCL-Ausdrucks können die empfangenen Änderungsbeschreibungen $I_1 - I_{n-1}$ mit der Startänderungsbeschreibung I_S in Beziehung gebracht werden, sodass auch die passenden Änderungsbeschreibungen empfangen werden. Nach dem obigen Beispiel bedeutet dies, dass die Bestellung empfangen wird,

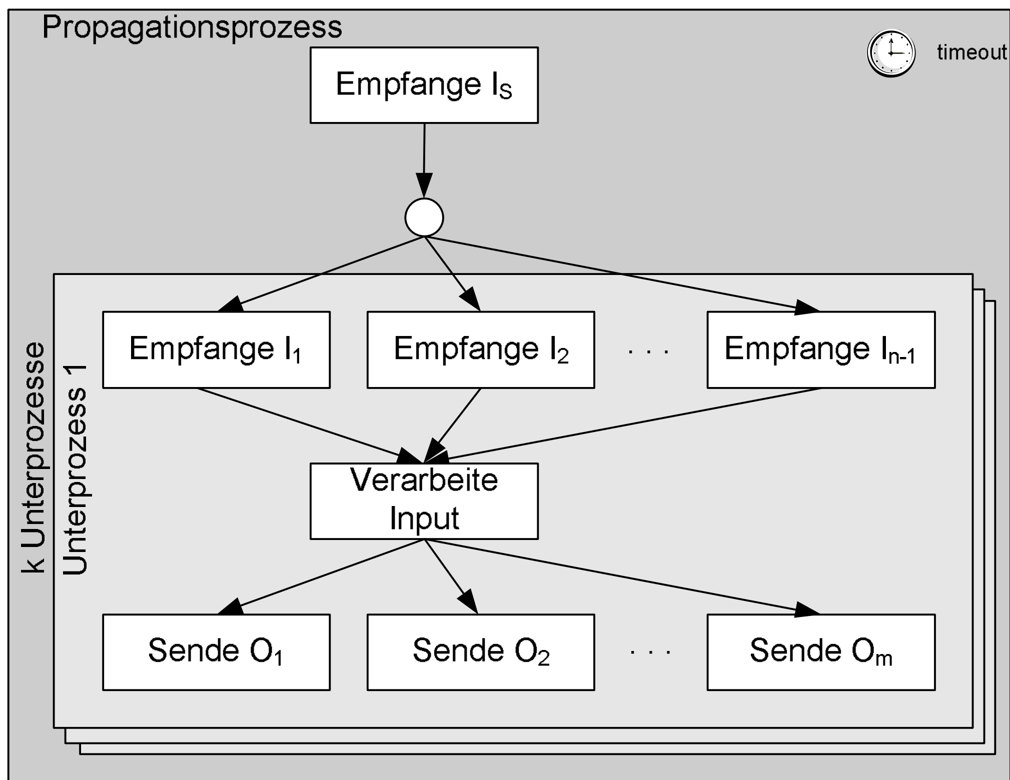


Abbildung 4.8: Prozessdarstellung einer M-zu-N-Abhängigkeit

die vom entsprechenden Kunden getätigt wurde. In einem Verarbeitungsschritt werden dann alle empfangenen Änderungsbeschreibungen verarbeitet und zum Output transformiert, der dann versendet wird. Der ganze Propagationsprozess wird so lange ausgeführt, bis alle benötigten Änderungsbeschreibungen empfangen wurden oder eine Zeitüberschreitung eingetreten ist (timeout). Um negierte Geschäftsregeln zu ermöglichen, können optional nach dem Eintreten der Zeitüberschreitung alle Unterprozesse mit unvollständigem Input gestartet werden. Eine negierte Geschäftsregel ist eine Geschäftsregel, die das Nicht-Eintreten eines Ereignisses abprüft und dann eine entsprechende Aktion auslöst. Ein Beispiel hierfür wäre, wenn der neu angelegte Kunde keine Bestellung innerhalb von 20 Stunden aufgibt, ihm eine Benachrichtigung zu senden, die ihn an das Angebot erinnert. Durch die Verwendung der XPath-Bibliothek (vgl. Abschnitt 3.6.2) kann das Vorhandensein von Änderungsbeschreibungen abgefragt und entsprechend reagiert werden.

4.2.2 Implementierungskonzept der M-zu-N-Erweiterung

Zuerst wird das Basiskonzept der Implementierung der M-zu-N-Erweiterung erklärt. Danach wird auf die Implementierung des M-zu-N-Managers eingegangen. Um die M-zu-N-Abhängigkeiten zu implementieren, muss der Prozessmanager um eine M-zu-N-Komponente erweitert werden, welche M-zu-N-Manager heißt. Eine große Herausforderung stellt dabei die Implementierung von Zeitüberschreitungen dar, deren Imple-

mentierungsmöglichkeiten anschließend diskutiert werden. Schließlich wird noch auf die Wiederherstellung von M-zu-N-Prozessen nach einem Systemcrash eingegangen.

4.2.2.1 Grundlegendes Konzept der M-zu-N-Realisierung

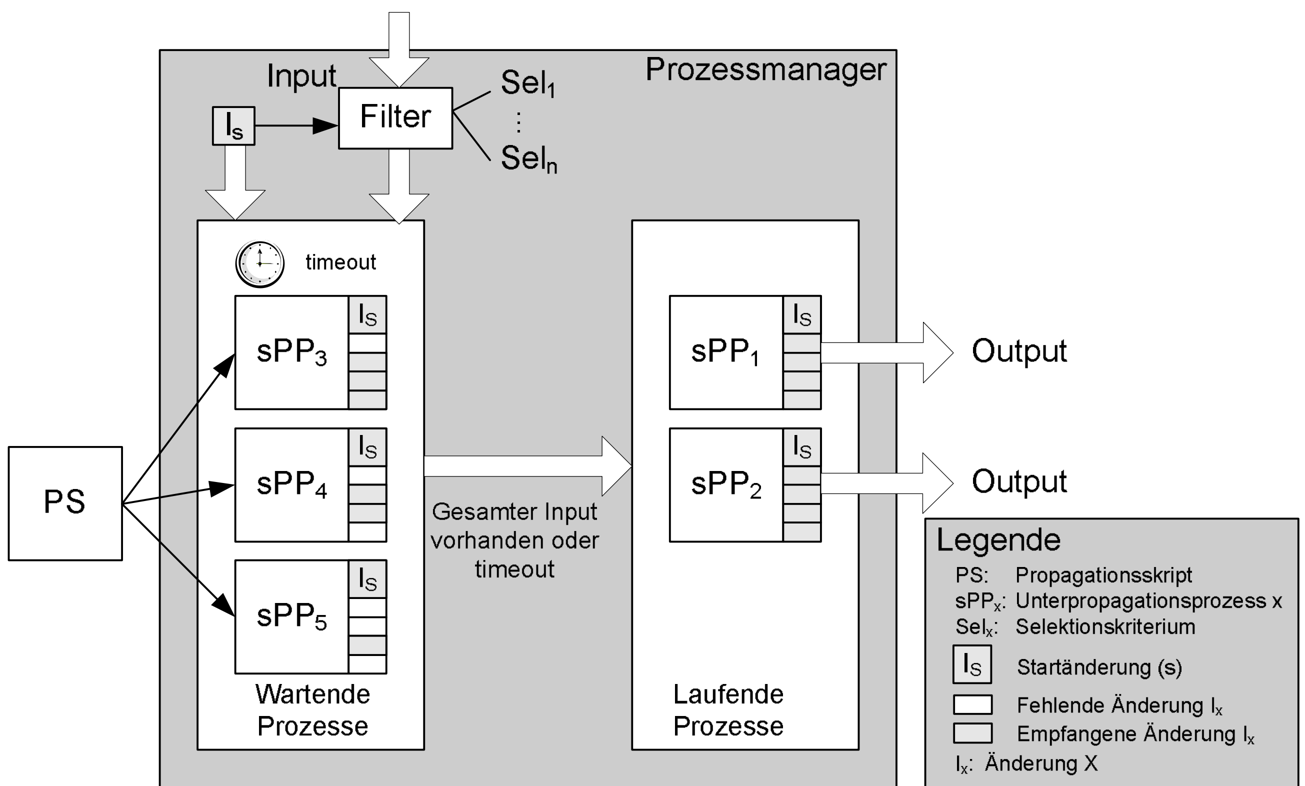
Da der Prozessmanager, so wie er bisher konzipiert wurde, die empfangenen Änderungsbeschreibungen entgegennimmt, Prozesse startet und ABs an die Prozesse verteilt, sollte er für das M-zu-N-Konzept ebenfalls im Mittelpunkt stehen. In Abbildung 4.9 ist das grundlegende Konzept dargestellt. Zuerst empfängt der Prozessmanager eine Startänderungsbeschreibung I_S und lädt das entsprechende Propagationsskript PS. Er initialisiert laut dessen Beschreibung k Unterprozesse ($sPP_1 - sPP_5$) und liest die Selektionskriterien Sel_x für den Filter aus. Die Startänderungsbeschreibung wird jedem Propagationsunterprozess zugeteilt. Kommen nun Änderungsbeschreibungen durch den Filter, werden sie entsprechend ihres Typs den Unterprozessen zugeteilt, und zwar nach der Reihenfolge der Initialisierung. Dies bedeutet, dass zuerst der erste Unterprozess seine Änderungsbeschreibung bekommt. Wenn dieser die Änderungsbeschreibung schon hat, bekommt sie der zweite usw. Besitzt ein Prozess (sPP_1 und sPP_2) alle benötigten Änderungsbeschreibungen, wird er vom Prozessmanager gestartet und der Output wird dann an die entsprechenden Systeme versendet. Dies ist die eigentliche Ausführung des Unterprozesses und kann innerhalb eines Microflows erfolgen.

Der unten in der Abbildung 4.9 aufgeführte Beispielablauf an eintreffenden Änderungsbeschreibungen führt zu der dargestellten Situation des Prozessmanagers.

Um M-zu-N-Abhängigkeiten zu implementieren, muss die Sprache XPDL (vgl. Abschnitt 3.5) erweitert werden, damit der zusätzliche Input ($I_1 - I_{n-1}$) definiert und die empfangenen Änderungsbeschreibungen integriert werden können. Weiterhin muss die Sprache für Änderungsbedingungen (PCL vgl. Abschnitt 3.6.1) angepasst werden, so dass Bedingungen Sel_x zwischen der Startänderung I_S und den zusätzlichen Änderungen I_x definiert werden können. Nicht nur die Sprachen zur Beschreibung von Abhängigkeiten und Bedingungen müssen erweitert werden, sondern auch die Funktionalität der Prozessmanager-Komponente des Propagationsmanagers. Dieser muss um die Filterkomponente erweitert werden, die innerhalb der Unterkomponente M-zu-N-Manager realisiert wird. Zusätzlich stellt die Wiederherstellung des Propagationsmanagers nach einem Systemabsturz gewisse Herausforderungen dar, die mit der Implementierung von Zeitüberschreitungen einhergehen.

4.2.2.2 M-zu-N-Manager

Der M-zu-N-Manager ist eine Unterkomponente des Prozessmanagers (vgl. Abschnitt 3.7.2) und verwaltet die Wartelisten der M-zu-N-Prozesse. Die Wartelisten geben an, welcher Unterprozess auf welche Änderungsbeschreibungen wartet und verwaltet damit die in Abbildung 4.9 dargestellte Situation an wartenden Prozessen mit den jeweiligen Filterbedingungen. Diese Warteliste muss so realisiert sein, dass der M-zu-N-Manager schnell bei einer eintreffenden Änderungsbeschreibung entscheiden kann, ob sie von einem M-zu-N-Prozess benötigt wird oder nicht. Dies erfolgt durch eine Vorselektion anhand des Systems und des Geschäftsobjekttyps. War die Vorselektion erfolgreich, so



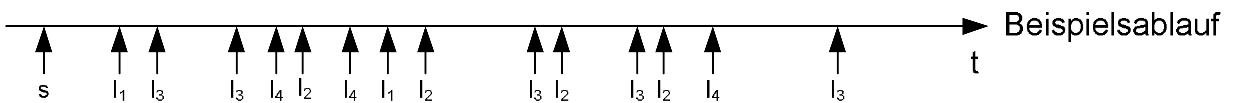


Abbildung 4.9: Implementierungsübersicht von M-zu-N-Abhängigkeiten

werden die Zustandsbeschreibungen der Geschäftsobjekte geparst und die Filterbedingungen Sel_x ausgewertet. Treffen diese Bedingungen auf die Änderungsbeschreibung zu, wird sie dem ersten Unterprozess zugeordnet. Weiterhin weckt der M-zu-N-Manager die M-zu-N-Prozesse auf, die über alle benötigten Änderungsbeschreibungen verfügen.

Der M-zu-N-Manager (`M2NManager`) verfügt über Methoden, die er zur Kommunikation mit dem M-zu-N-Prozess oder Prozessmanager bereitstellt. Um die Erstellung der Warteliste zu vereinfachen, meldet der Propagationsprozess die benötigten Filter für die Änderungsbeschreibungen an den M-zu-N-Manager. Dies begründet sich in der Kenntnis der XPDL-Engine über die XPDL-Sprache, die der Prozessmanager so nicht hat. Der Prozess, der in der XPDL-Engine initialisiert wurde, kann dadurch die `change_input` und `timeout`-Statements analysieren und dem M-zu-N-Manager mitteilen.

4.2.2.3 Zeitüberschreitungen

Die Zeitüberschreitung ist durch einen Zeitraum definiert. Ist der Zeitraum abgelaufen, kommt es zu einer Zeitüberschreitung (timeout), die dann über das Ausgehen des Propagationsprozesses entscheidet: entweder wird der Prozess abgebrochen (Geschäftsregel) oder trotzdem gestartet (negierte Geschäftsregel).

Die Problematik mit Zeitüberschreitungen ist die Definition der Zeitgrenzen, d.h. wann fängt der Zeitraum an und wann endet er. Es existieren mehrere Möglichkeiten. Der Anfang kann durch den Sendezeitpunkt der Startänderungsbeschreibung definiert werden, wobei man zwischen realem und effektivem Zeitpunkt unterscheiden kann. Der reale Zeitpunkt gibt dabei an, wann die Änderungsbeschreibung dem Warteschlangenmanager übergeben wurde. Zu diesem Zeitpunkt muss die Änderungsbeschreibung noch nicht sichtbar sein, da sie durch eine Transaktion erst bei deren Commit sichtbar wird. Der Zeitpunkt, zu dem die Änderungsbeschreibung sichtbar wird, ist der effektive Zeitpunkt. Weiterhin kann der Anfang des Zeitraums auch durch den Start des Prozesses definiert werden, d.h. wenn der Propagationsmanager die Änderungsbeschreibung sieht. Diese kann – je nach Länge der Eingangswarteschlange – deutlich von den anderen zwei Zeitpunkten abweichen.

Am besten wäre der effektive Sendezeitpunkt, da zu dieser Zeit die Änderungsbeschreibung sichtbar wird, d.h. zum Beispiel die Bestellung wird aufgegeben. Allerdings stellt das *Java Message Service* (JMS) diese Information nicht bereit, sondern nur den realen Sendezeitpunkt. Die Spezifikation sagt aus, dass es dem Warteschlangensystem überlassen wird, ob es den realen Sendezeitpunkt anbietet oder nicht. Der Zeitpunkt des Prozessstarts ist weniger geeignet, da er im Minutenbereich nach dem aktuellen Senden der Nachricht liegen kann.

Das Ende ist durch den Startzeitpunkt und die angegebene Zeitdauer definiert. Auch hier kann man zwischen zwei Implementierungsmöglichkeiten unterscheiden. Entweder kann sofort abgebrochen werden oder erst wenn alle aktuell vorhandenen Änderungsbeschreibungen abgearbeitet sind. Da der Startzeitpunkt durch den realen Sendezeitpunkt definiert ist, wird die letzte angenommene Änderungsbeschreibung auch durch den realen Sendezeitpunkt definiert. Dabei wird eine effektive Zeitdauer d_{eff} verwendet, die sich aus dem realen Sendezeitpunkt t_{AB}, der Prozessstartzeit t_P und ange-

gebenen Zeitdauer d ergibt $d_{eff} = d - (t_P - t_{AB})$. Die effektive Zeitdauer gibt dabei die noch verbleibende Wartezeit an. Die Berechnung der effektiven Zeitdauer ist deshalb notwendig, da bei Prozessstart schon eine gewisse Zeit vergangen ist, seit die Änderungsbeschreibung versendet wurde. Ein Timer benachrichtigt den M-zu-N-Manager über den Ablauf von effektiven Zeitdauern. Tritt ein solches Ereignis auf, können sich aber unverarbeitete Änderungsbeschreibungen in der Eingangswarteschlange befinden, die noch dem M-zu-N-Unterprozessen zugeordnet werden müssen, d.h. der reale Sendezeitpunkt liegt vor der aktuellen Zeit. Um die Verarbeitung zu ermöglichen wird beim Auftreten eines Timer-Ereignisses eine Timeout-Nachricht in die Eingangswarteschlange geschrieben. Alle Änderungsbeschreibungen vor dieser Nachricht werden noch angenommen, alle danach werden dem entsprechenden M-zu-N-Prozess nicht mehr zugeordnet. Falls gefordert, löst der Empfang der Timeout-Nachricht auch das Aufwecken der unvollständigen Unterprozesse aus, da zu diesem Zeitpunkt alle im korrekten Zeitraum empfangenen Änderungsbeschreibungen verfügbar sind. Danach kann der M-zu-N-Manager die Warteliste des jeweiligen Propagationsprozesses löschen.

4.2.2.4 Wiederherstellung

Nach einem Systemabsturz gilt es, die Änderungsbeschreibungen, die Prozessinformationen im M-zu-N-Manager und die Timer-Informationen wiederherzustellen. Die Wiederherstellung der Änderungsbeschreibungen übernimmt das Warteschlangensystem. Dies war einer der Gründe, warum ein persistentes Warteschlangensystem ausgewählt wurde.

Die Prozessinformationen im M-zu-N-Manager können anhand der Startänderungsbeschreibung und eines „Neustarts“ der jeweiligen Prozesse wiederhergestellt werden. Zusätzlich muss die zuverlässige Multicast-Warteschlange (vgl. Abschnitt 3.7.2.3), die zur Kommunikation mit den Prozessen dient, durchlaufen werden und die Verfügbarkeit der Änderungsbeschreibungen im M-zu-N-Manager markiert werden. Als eine alternative Implementierung bietet sich das Protokollieren der empfangenen zusätzlichen Änderungsbeschreibungen an. Dies würde die Wiederherstellungsphase beschleunigen, denn die Warteschlange zur Prozesskommunikation muss nicht durchlaufen werden. Allerdings verlangsamt es den M-zu-N-Manager bei der Verarbeitung von Änderungsbeschreibungen, da diese zusätzlich protokolliert werden müssen. Da eine Wiederherstellung seltener erforderlich ist, wurde die Verarbeitung in der „Normalphase“ beschleunigt und die Wiederherstellungsphase verlangsamt.

Der reale Sendezeitpunkt kann nicht durch die Multicast-Warteschlange ermittelt werden, da diese durch die Eingangswarteschlange definiert war. Außerdem kann die Prozessstartzeit nicht mehr ermittelt werden. Aus diesem Grund werden die Timer-Informationen protokolliert. Dadurch kann bei Wiederherstellung festgestellt werden, ob Prozesse in der Zwischenzeit einem Timeout unterlegen sind und für die anderen Prozesse den Timeout neu setzen. Alternativ können diese Werte auch in den Nachrichten-Properties von JMS gespeichert werden.

4.2.3 Erweiterung von XPDL

Um M-zu-N-Abhängigkeiten zu realisieren, muss die Eingabedeklaration des Propagationsskriptes ergänzt werden, damit auch der Empfang der Änderungsbeschreibungen I_1 bis I_{n-1} definiert werden kann. Des Weiteren soll eine Zeitbeschränkung in Form eines Timeouts angeben werden können. Für die Definition der zu empfangenden Änderungsbeschreibungen wird der folgende XPDL-Befehl eingeführt:

```
change_input(system (s), GO_Typ (GT), out, expression?).
```

Dieser Befehl unterscheidet sich kaum vom `start_input`-Befehl (vgl. Abschnitt 3.5.1). Allerdings sollte der Befehl als `expression` keine allgemeine Bedingung enthalten, sondern eine Bedingung, die die Startänderungsbeschreibung I_s mit der hier geforderten Änderungsbeschreibung I_x verknüpft. Ein weiterer Unterscheidungspunkt zu `start_input` ist, dass der Empfang keinen Prozessstart auslöst, d.h. der Prozessmanager lädt kein Propagationsskript und initialisiert keinen Propagationsprozess. Allerdings wird bei Empfang aller geforderten Änderungsbeschreibungen ein Unterprozess ausgeführt.

Der angesprochene Timeout wird mit dem folgenden Befehl ebenfalls in der Eingabedeklaration definiert:

```
timeout(duration, terminate).
```

Das `duration`-Attribut gibt die Zeitspanne an, wie lange der Prozess auf den Empfang von Änderungsbeschreibungen wartet. Das zweite Attribut (`terminate`) gibt an, ob der Prozess bei unvollständigen Ereignissen abgebrochen werden soll. Wie schon erwähnt wurde, können bei nicht Beendigung des Propagationsprozesses nach dem Auftreten einer Zeitüberschreitung, negierte Geschäftsregeln realisiert werden, die auf das Nicht-Eintreffen einer Änderungsbeschreibung reagieren.

Des Weiteren wird XPDL so erweitert, dass die maximale Anzahl von Unterprozessen definiert werden kann. Dies erfolgt im Wurzelelement (`propagationscript`).

Die Integration der Änderungsbeschreibungen wurde schon für die Einbindung von Daten aus Drittsystemen benötigt. In diesem Fall musste die empfangene Änderungsbeschreibung mit zusätzlichen Daten integriert werden. Bei M-zu-N dagegen handelt es sich bei den „zusätzlichen Daten“ ebenfalls um Änderungsbeschreibungen ABs[1]. Es wird ebenfalls eine Transformation für die Integration verwendet, wobei die Transformation über eine Hauptänderung und mehrere Nebenänderungen erfolgt. Die Hauptänderungsbeschreibung definiert dabei die Elemente S, GT, A und TS des Transformationsoutputs, d.h. diese Elemente werden direkt von der Hauptänderung übernommen. Die Zustände B und D werden transformiert und mit den Zuständen der anderen Änderungen integriert. Die Integration wird durch die XPath-Bibliothek ermöglicht (vgl. Abschnitt 3.6.2). Die Anzahl der zu transformierenden Zustände hängt von der Hauptänderung ab.

[1] $AB = (S, GT, A, B, D, TS)$ (vgl. Abschnitt 3.2.2)

4.2.4 Erweiterung von PCL

Die zusätzlich benötigten Änderungsbeschreibungen für einen M-zu-N-Prozess müssen mit der Startänderungsbeschreibung in Beziehung gesetzt werden. Dies erfolgt durch Bedingungen (`expression`) innerhalb der `change_input`-Deklaration. Da Bedingungen im Propagationssystem mit einer speziellen Bedingungssprache (PCL), die in Abschnitt 3.6.1 eingeführt wurde, definiert werden, wird auch für diesen Anwendungsfall PCL verwendet. Um die Art von Bedingungen genauer zu untersuchen, soll das Beispiel von Abschnitt 4.2.1 betrachtet werden, indem ein Kunde mit einer Bestellung in Beziehung gesetzt werden soll. Der Kunde verfügt über eine Kundennummer, die ihn eindeutig definiert und die auch in der Bestellung verwendet wird. Dann können Kunde und Bestellung mit

```
%startChange:afterOrBefore/Kunde/Kundennummer% =
%afterOrBefore%/Bestellung/Kundennummer
```

in Beziehung gesetzt werden. Der Kunde wird dabei eindeutig durch `startChange` identifiziert. Es können nur Bedingungen zwischen der Änderungsbeschreibung $I_1 - I_{n-1}$ (im PCL-Beispiel die Bestellung) und der Startänderungsbeschreibung I_s definiert werden, denn die Startänderung ist die einzige Änderungsbeschreibung, die auf jeden Fall vor allen anderen Änderungsbeschreibungen des M-zu-N-Prozesses empfangen wird. Damit können nur diese und die aktuelle Änderungsbeschreibung in Bedingungen verwendet werden. Weiterhin kann der PCL-*StartChange*-Teil, eingeschlossen durch die Prozentzeichen, schon während der Prozessinitialisierung durch einen Wert ersetzt werden, d.h. %startChange:/Kunde/kundennummer%' wird zum Beispiel durch 123 ersetzt.

Um diese Art von Umsetzung der PCL-Bedingungen zu ermöglichen, muss die Architektur des PCL-Compilers von Abschnitt 3.6.1 erweitert werden, sodass sie die Startänderung als Input hat. Dadurch sieht die Architektur wie in Abbildung 4.10 dargestellt aus. Dabei wurde die ursprüngliche Architektur so erweitert, dass der PCL-Compiler die Startänderung zur Verfügung hat.

4.2.5 Schlussfolgerungen

Die M-zu-N-Abhängigkeit ist eine Möglichkeit, um Geschäftsregeln mit einem Propagationssystem zu realisieren. Für die Implementierung von Geschäftsregeln bieten sich aber eher Workflow-Managementsysteme (WfMS, siehe Abschnitt 2.6) an. Diese Systeme bieten mehr Möglichkeiten, um sogenannte Geschäftsregeln zu implementieren, da die Prozesse flexibler gestaltet werden können. Diese basieren dann aber weniger auf geänderten Daten (Änderungsbeschreibungen). Ein Vorteil des hier vorgeschlagenen Ansatzes ist, dass trotz lang laufender Prozesse keine *Backward Recovery* (vgl. Abschnitt 2.6) benötigt wird, da die eigentliche Ausführung von kurzer Zeitdauer ist und innerhalb einer normalen Transaktion des Propagationsprozesses ausgeführt werden kann. Dies kann aber ebenfalls durch einen Workflow erreicht werden, indem die Verarbeitung bis auf die Empfangsoperationen in einer Transaktionssphäre ausgeführt wird. Änderungsbeschreibungen, die durch M-zu-N-Abhängigkeiten verarbei-

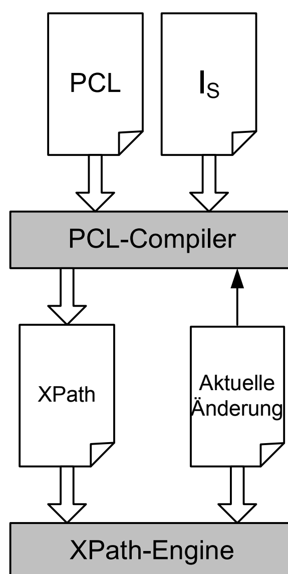


Abbildung 4.10: Die Architektur der Verarbeitung von PCL-Bedingungen für die Unterstützung von M-zu-N-Abhängigkeiten

tet werden, können nicht mit den anderen Änderungsbeschreibungen in Reihenfolge gebracht werden, da M-zu-N lang laufende Prozesse sind und sie die 1-zu-N-Prozesse lange verzögern würden, wenn die M-zu-N-Prozesse ebenfalls in Reihenfolge gebracht würden. Die Einhaltung der Reihenfolge stellt gerade bei einem Propagationssystem eine wichtige Eigenschaft dar. Außerdem liegt das Verarbeiten von M-zu-N-Abhängigkeiten außerhalb des „Kerngeschäfts“ des Propagationssystems. Aus diesen Gründen wird in dieser Arbeit die Implementierung der M-zu-N-Abhängigkeiten als Modul vorgeschlagen, das aktiviert und deaktiviert werden kann und nicht Kern eines jeden Propagationssystem ist.

4.3 Verteilte Propagation

In diesem Abschnitt wird die Verteilung des Propagationsmanagers diskutiert. Dadurch soll ein höherer Durchsatz an verarbeiteten Änderungsbeschreibungen erreicht werden. Um dieses Problem zu lösen, werden hier zwei Ansätze vorgestellt (Abbildung 4.11), der Load-Manager-Ansatz und der selbstorganisierte Ansatz. Beim Load-Manager-Ansatz gibt es eine zentrale Instanz, welche die Verteilung der Änderungsbeschreibungen übernimmt. Beim zweiten Ansatz erfolgt die Verteilung selbstorganisiert, indem jeder Propagationsmanager Änderungsbeschreibungen liest, sobald ihm es möglich ist. Das Warteschlangensystem garantiert, dass jede Änderungsbeschreibung von jeweils nur einem Propagationsmanager gelesen wird.

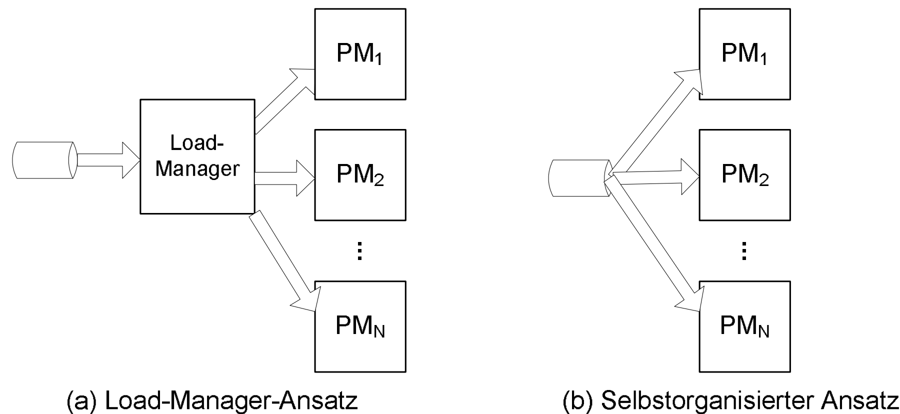


Abbildung 4.11: Ansätze zur Verteilung des Propagationsmanagers

4.3.1 Problemstellung

In manchen Einsatzszenarien im Unternehmen kann es sein, dass der Durchsatz oder die Verfügbarkeit des Propagationsmanagers nicht ausreichend ist. Für den ersten Fall kommt man in den meisten Fällen nicht an einer Verteilung vorbei. In seltenen Fällen kann auch die Optimierung von Propagationsmanager oder Repository ausreichend sein. Eine weitere Möglichkeit ist die Verwendung einer leistungsstärkeren Hardware. Der Abhängigkeitsmanager wirkt sich nicht auf die Performance von Änderungspropagationen aus, da er nur für die Entwicklungszeit zuständig ist und damit keinen Einfluss auf die Laufzeit hat. Für den zweiten Fall, die Verfügbarkeit, können auch sogenannte Hot-Backups eingesetzt werden, die die Aufgaben des Propagationsmanagers im Falle eines Ausfalls übernehmen. Diese Konstellation erhöht aber nicht die Performance, was durch die Verteilung des Propagationsmanagers erreicht werden kann. Allerdings kann die Verteilung die Verfügbarkeit nur bedingt erhöhen, da Beziehungen zwischen den einzelnen Änderungsanforderungen existieren, die eine freie Verteilung verhindern. Dies gilt insbesondere für die M-zu-N-Abhängigkeiten, da diese Abhängigkeiten zwischen den einzelnen Änderungsbeschreibungen aufweisen.

Hier wird von einer Verteilung des Propagationsmanagers als Ganzes ausgegangen. Alternativ kann man sich eine Verteilung seiner Komponenten, z.B. des *Transformer* oder des *PCL-Evaluator*, vorstellen. Dieser Ansatz erhöht sowohl den Kommunikationsbedarf innerhalb des Propagationsmanagers als auch die Fehleranfälligkeit. Weiterhin sind Propagationsprozesse keine lang laufenden Prozesse und eine Belastung des Systems kommt eher durch eine Vielzahl von Propagationsprozessen zustande. Außerdem ist die Lösung mit der Komponentenverteilung mit einem höheren Aufwand verbunden. Bei der Verteilung des Propagationsmanagers als Ganzes müssen lediglich n Propagationsmanager auf n Rechner installiert werden sowie eventuell zusätzlich eine spezielle Komponente, den sogenannten Load-Manager.

Es wird davon ausgegangen, dass für die einzelnen Warteschlangen (Queues) Strategien zur Performanceerhöhung existieren, wie zum Beispiel der Einsatz von RAID-Laufwerken und die Verteilung des Warteschlangensystems.

4.3.2 Einschränkungen der Lastverteilung

Die Verteilung der Last auf verschiedene Propagationsmanager wird durch Abhängigkeiten zwischen den einzelnen zu verteilenden Änderungsbeschreibungen eingeschränkt. Die Änderungsbeschreibungen sind im Grunde genommen sehr unabhängig und die Verteilung wird nur durch das Reihenfolgeproblem und M-zu-N-Abhängigkeiten eingeschränkt, was nachfolgend beschrieben wird.

4.3.2.1 Reihenfolgeproblem

Wie in Abschnitt 3.9 schon erwähnt wurde, muss die Reihenfolge von verarbeiteten Änderungsbeschreibungen eingehalten werden, sofern es die entsprechende Propagationsordnung verlangt. Ebenfalls wird in Abschnitt 3.9 eine Lösung für den zentralen Ansatz entworfen. Dieser Ansatz funktioniert im verteilten Fall nur, wenn diese Abhängigkeit zwischen den Änderungsbeschreibungen beachtet wird. Das bedeutet, dass eine zentrale Implementierung für die gewählte Propagationsordnung gebraucht wird, an die sich dann die einzelnen Propagationsmanager wenden. Dadurch können die Änderungsbeschreibungen weiterhin flexibel verteilt werden. Alternativ könnte der Load-Manager die Änderungsbeschreibung an die einzelnen Propagationsmanager verteilen, so dass die Reihenfolge eingehalten wird. Dies schränkt die Verteilbarkeit je nach Algorithmus deutlich ein und macht eine totale Propagationsordnung völlig unmöglich, da in diesem Fall alle Änderungsbeschreibungen an ein Propagationssystem gesendet werden müssten.

Aus diesem Grund wird eine freie Verteilung gewählt, bei der die Propagation von Änderungsbeschreibungen gegebenenfalls verzögert wird. Das bedeutet, dass der erste Teil (z.B. Transformationen und Bedingungen) schon ausgeführt wurde und nur die letzten Schritte verzögert werden, d.h. das Schreiben in die Ausgangswarteschlangen.

4.3.2.2 M-zu-N-Abhängigkeiten

Die in diesem Kapitel eingeführten M-zu-N-Abhängigkeiten stellen ebenfalls eine Beschränkung der Verteilungsfreiheit des Load-Managers dar, weil M-zu-N-Prozesse auf bestimmte Änderungsbeschreibungen warten, die der jeweilige Propagationsmanager auch erhalten muss. Die erwartete Änderungsbeschreibung wird dabei durch (S, GOTyp, BE) bestimmt, wobei S das Quellsystem, GOTyp der Geschäftsobjekttyp und BE die Bedingung ist, die den Inhalt einer Änderungsbeschreibung (B oder D) erfüllen muss. Wann immer eine solche Änderungsbeschreibung erkannt wird, muss diese an den entsprechenden Propagationsmanager gesendet werden, der auf diese Nachricht wartet.

Allerdings sollte es vermieden werden, M-zu-N-Abhängigkeiten in eine verteilte Lösung einzubinden, da sie die Abhängigkeiten zwischen den einzelnen Änderungsbeschreibungen stark erhöhen und damit auch die Leistungssteigerung bei der Verteilung senken. Eine Möglichkeit dies zu umgehen, wenn M-zu-N unterstützt werden soll, ist die Einrichtung eines Propagationsmanager oder einer Gruppe von Propagationsmanagern, die ausschließlich M-zu-N-Abhängigkeiten bearbeiten. Dadurch wird verhindert,

dass die einfachen Prozesse verlangsamt werden.

4.3.2.3 Auswirkungen

Ein verteiltes System, das Warteschlangen zur Kommunikation verwendet, kann sich selbstorganisieren, sofern keine Einschränkungen vorhanden sind. Dies bedeutet, dass einer der Propagationsmanager eine Änderungsbeschreibung aus der Eingangswarteschlange holt, diese verarbeitet und sobald er fertig ist, eine neue holt. Dadurch ist der Propagationsmanager nicht überlastet, da er nur soviel verarbeitet, wie er abarbeiten kann. Da mehrere Propagationsmanager eingesetzt werden, können diese die Last teilen. Kommen dagegen Abhängigkeiten zwischen den Änderungsbeschreibungen ins Spiel, ist eine selbstorganisierte Vorgehensweise nicht mehr sinnvoll, da bestimmte Änderungsbeschreibungen von bestimmten Propagationsmanagern verarbeitet werden müssen. Dies führt zu einem erhöhten Kommunikationsaufwand zwischen den Propagationsmanagern und zu einem Austausch der Änderungsbeschreibungen. Aus diesem Grund ist eine organisierte Verteilung unumgänglich, die vom oben erwähnten Load-Manager durchgeführt wird. Dies betrifft insbesondere die Unterstützung von M-zu-N-Abhängigkeiten.

Abhängigkeiten durch die Reihenfolge können allerdings durch Selbstorganisation erreicht werden, da ein zentraler Reihenfolge-Algorithmus die jeweiligen Prozesse verzögern könnte. Dieser muss den jeweiligen Propagationsmanagern vorgeschaltet werden. Zuerst wollen wir allerdings die Lösung mit einer organisierten Verteilung untersuchen, die einen *Load-Manager* verwendet und damit auch M-zu-N-Abhängigkeiten unterstützt.

4.3.3 Load-Manager-Ansatz

Als Erstes soll die organisierte Verteilung anhand eines *Load-Managers* untersucht werden. Der Load-Manager stellt eine zentralisierte Komponente dar, die die Last an Änderungsbeschreibungen gleichmäßig auf die Propagationsmanager verteilen soll. Er kann durch die Zentralisierung aber selbst einen Flaschenhals darstellen und ist ein Ausfallrisiko, was aber durch ein Hot-Backup abgeschwächt werden kann. Das Hot-Backup übernimmt bei einem Ausfall dessen Aufgaben [Yan04].

4.3.3.1 Architektur

Die Architektur eines Load-Managers für ein verteiltes Propagationssystem ist in Abbildung 4.12 dargestellt. Kern der Anwendung ist das Kernmodul, welches die anderen Module steuert. Für die Kommunikation mit der Außenwelt existieren zwei unterschiedliche Module. Die Warteschlangenschnittstelle dient zum Empfang der Änderungsbeschreibungen durch die zentrale Eingangswarteschlange und zum Senden der Änderungsbeschreibungen an die jeweiligen Warteschlangen der Propagationsmanager (PM_x). Im verteilten Fall mit Load-Manager-Ansatz hat jeder der Propagationsmanager eine Eingangswarteschlange, in der seine zu verarbeitenden Änderungsbeschreibungen stehen. Dadurch wird eine Entkopplung der Propagationsmanager vom Load-

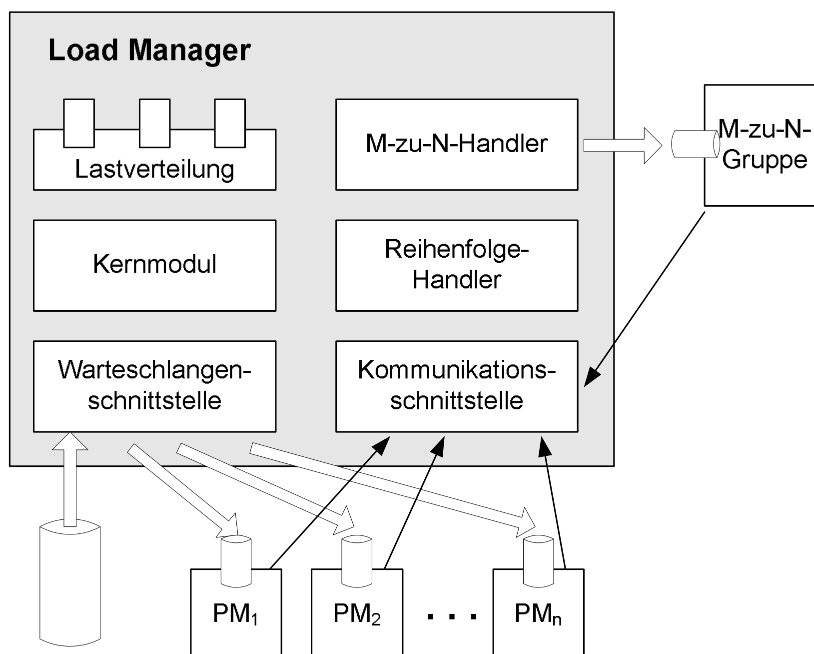


Abbildung 4.12: Architektur des Load-Managers

Manager erreicht. Auch in diesem Fall ist der Einsatz von persistenten Warteschlangen wichtig, um eine hohe Ausfallsicherheit und eine hohe Übermittlungsgarantie zu gewährleisten.

Das zweite Kommunikationsmodul (Kommunikationsschnittstelle) dient zum Empfang von Informationsnachrichten, wie zum Beispiel die aktuelle Lasten der Propagationsmanager oder Anfragen für die Reihenfolgeeinhaltung. Durch den Einbezug der aktuellen Last kann eine bessere Verteilung erreicht werden. Des Weiteren kann die Anzahl der Änderungsbeschreibungen in den Warteschlangen einbezogen werden, um eine bessere Verteilung zu ermöglichen.

Der Lastverteiler steuert die Verteilung der Änderungsbeschreibungen auf die einzelnen Propagationsmanager. Dafür existieren mehrere Strategien, die in den Untermodulen realisiert sind und vom Verteiler angesteuert werden können (z.B. Round-Robin). Beim Round-Robin Verfahren, werden die Änderungsbeschreibung der Reihe nach an die einzelnen Propagationsmanager verteilt, ohne dass die aktuelle Last berücksichtigt wird.

Weiterhin existieren noch zwei Module (M-zu-N- und Reihenfolge-Handler), die die Verarbeitung von M-zu-N-Abhängigkeiten ermöglichen und die Einhaltung der Reihenfolge garantieren.

4.3.3.2 M-zu-N-Verarbeitung

Das hier angestrebte Konzept basiert auf einer speziellen Gruppe von Propagationsmanagern, die die M-zu-N-Abhängigkeiten ausführen und damit die „einfachen“ Abhängigkeiten nicht ausbremsen. Des Weiteren wird den M-zu-N-Propagationsmanagern eine

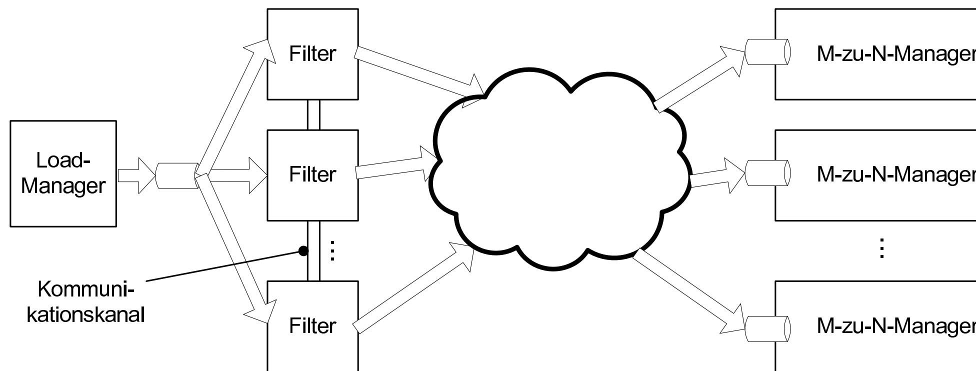


Abbildung 4.13: Verteilung der M-zu-N-Abhängigkeiten

Gruppe von Filtern vorgeschaltet, die nur benötigte Änderungsbeschreibungen an die entsprechenden Propagationsmanager durchlassen. Diese Filter dienen zur Entlastung der Propagationsmanager. Die Filter sind außerdem notwendig, da ansonsten das M-zu-N-Problem nicht verteilbar wäre, ohne den Propagationsmanager anzupassen. In Abbildung 4.13 ist die Verteilungsarchitektur dargestellt.

Das Prinzip dabei ist, dass der Load-Manager das Starten von neuen Prozessen steuert und die Verteilung auf die einzelnen Filter selbstorganisiert erfolgt, ermöglicht durch eine gemeinsame Warteschlange. Dadurch wird eine optimale Verteilung an die Filter ermöglicht.

Der Filter hat die Aufgabe, die Änderungsbeschreibungen anhand der Bedürfnisse der Propagationsmanager zu filtern und sie nur an interessierte zu senden. Interessiert ist ein Propagationsmanager, wenn er auf eine Änderungsbeschreibung wartet. Dafür muss eine Warteliste verwaltet werden, deren Einträgen aus Quellsystem, Geschäftsobjekttyp, Bedingung und Propagationssystem bestehen. Die Filter kommunizieren über einen Kommunikationskanal. Dadurch werden Änderungen in der Warteliste mitgeteilt. Um die Aufgabe des Filters zu erfüllen, müssen die Zustände geparst und die Warteliste ausgewertet werden. Wird dabei ein Eintrag gefunden, bei dem Quellsystem sowie Geschäftsobjekttyp gleich sind und die Bedingung erfüllt ist, wird die Änderungsbeschreibung an den entsprechenden Propagationsmanager weitergeleitet (Bedienung bestehender Propagationsprozesse). Zusätzlich muss der Filter noch die Änderungsbeschreibung an den vom Load-Manager ausgewählten Propagationsmanager weiterleiten, damit dieser neue M-zu-N-Prozesse starten kann. Eine weitere Aufgabe des Filters ist die Verwaltung der Warteliste, die neue Einträge erhält, wenn Prozesse gestartet werden.

Problematisch ist dabei die korrekte Abarbeitung der Warteliste, d.h. der Einbezug aller benötigten Einträge, auch wenn diese evtl. noch nicht vorhanden sind. Dies kann durch die Verteilung der Filter auftreten. Um den Einbezug aller Einträge zu garantieren, verfügen die Änderungsbeschreibungen über eine logische Uhr (t_{AB}). Eine logische Uhr [Lam78] ist durch einen Zähler implementiert. Die Warteliste verfügt ebenfalls über eine logische Uhr (t_{WL}), welche den letzten Stand der verarbeiteten Änderungsbeschreibung enthält. Wenn eine empfangene Änderungsbeschreibung keine Prozesse startet, werden auch keine Einträge der Warteliste hinzugefügt, aber dennoch

muss die logische Uhr der Warteliste auf den aktuellen Stand gebracht werden. Wenn jetzt ein Filter eine Änderungsbeschreibung bekommt, parst dieser die Zustände und überprüft die Einträge der Warteliste. Ist die Warteliste veraltet, wartet der Filterprozess bis die Warteliste aktualisiert wird und alle Änderungsbeschreibungen enthält, d.h. $t_{AB} \leq t_{WL} - 1$. Die hinzukommenden Einträge werden sukzessive abgearbeitet. Ein Filter besteht aus mehreren Filterprozessen (realisiert als Threads), die die Filteraufgaben übernehmen.

4.3.3.3 Der Umgang mit der Reihenfolge

Das Lösungsprinzip des Reihenfolgeproblems im verteilten Fall basiert auf der freien Verteilung der Änderungsbeschreibung und der Implementierung der Reihenfolge-Algorithmen (vgl. Abschnitt 3.9) im Load-Manager. Der Reihenfolger-Handler (Abbildung 4.12) implementiert die verschiedenen Algorithmen, die mittels Konfiguration ausgewählt werden können. Möchte ein Propagationsprozess eines Propagationsmanagers eine Änderungsbeschreibung propagieren, fragt er den Reihefolge-Handler über die Kommunikationsschnittstelle, ob er schon an der Reihe ist. Eine weitere Informationsnachricht informiert den Reihefolge-Handler über die abgeschlossene Verarbeitung einer Änderungsbeschreibung. Dadurch können die nächsten anstehenden Prozesse ihre Änderungsbeschreibungen propagieren.

4.3.3.4 Erhöhung der Zuverlässigkeit

Durch den Einsatz eines verteilten Systems wird auch die Zuverlässigkeit des Gesamtsystems erhöht, da der Load-Manager einem ausgefallen Propagationsmanager keine Arbeit mehr zuteilt. Dies ist allerdings nur in gewissem Rahmen möglich, denn die Reihenfolge und M-zu-N-Abhängigkeiten müssen trotzdem erfüllt sein. Das heißt, dass nur frei verteilbare Änderungsbeschreibungen von anderen Propagationsmanagern bearbeitet werden können. Um dieses Manko zu beseitigen, kann ein Pool mit Hot-Backup-Propagationsmanagern eingeführt werden [Yan04], um die Arbeit ausgefallener Propagationsmanager zu übernehmen. Ein solcher Propagationsmanager steht dabei in Bereitschaft und beobachtet, ob ein anderer Propagationsmanager ausgefallen ist und übernimmt in diesem Fall seine Arbeit. Sobald ein ausgefallener Propagationsmanager wieder verfügbar ist, überprüft dieser, ob er benötigt wird und ordnet sich andernfalls dem Hot-Backup-Pool zu.

Der Load-Manager muss zur Steigerung der Zuverlässigkeit auch mit Hot-Backups ausgestattet werden [Yan04].

4.3.4 Selbstorganisierter Ansatz

Der selbstorganisierte Ansatz nutzt die hohe Verteilungsmöglichkeit durch den Einsatz von Warteschlangen aus. Dabei versucht ein Propagationsmanager immer seine Propagationsprozesse abzuarbeiten und nur bei Bedarf neue Änderungsbeschreibungen anzunehmen. Dieser Ansatz schließt aber die Implementierung von M-zu-N-Abhängigkeiten aus, da diese nicht selbstorganisiert verteilt werden können. In Abbil-

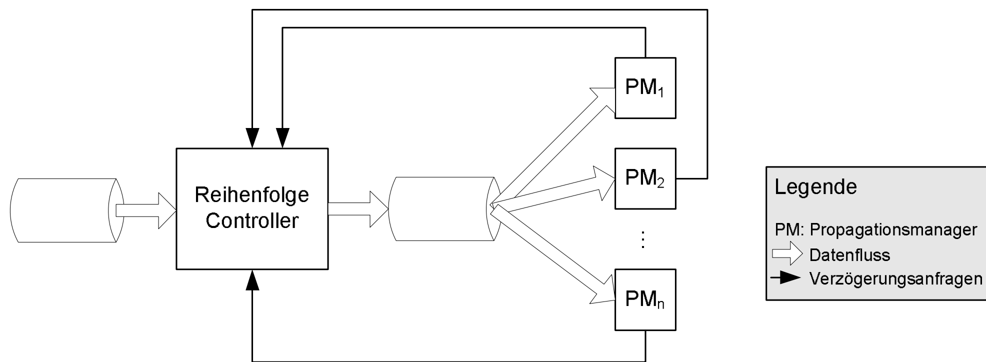


Abbildung 4.14: Selbstorganisierter Ansatz mit Reihenfolgebehandlung

dung 4.14 ist dieser Ansatz dargestellt. Den Propagationsmanagern ist ein Reihenfolge-Controller vorgeschaltet, der die Reihenfolge-Algorithmen realisiert. Die Propagationsmanager fragen diesen an, ob Änderungsbeschreibungen schon propagiert werden können (Verzögerungsanfragen). Zusätzlich informieren die Propagationsmanager den Reihenfolge-Controller über abgearbeitete Änderungsbeschreibungen. Dies wird ermöglicht indem die Änderungsbeschreibungen mit einer logischen Uhr [Lam78] versehen wird, die den Änderungsbeschreibungen eine eindeutige Zahl zuordnen. Diese Uhr ist durch einen Zähler implementiert. Dadurch kann der Reihenfolge-Controller entscheiden, ob ein Propagationsmanager seine ausgehenden Änderungsbeschreibungen propagieren kann.

4.4 Zusammenfassung

Im ersten Teil dieses Kapitels wurde eine Ergänzung eingeführt, mit der zusätzliche Daten in Änderungsbeschreibungen eingebunden werden können, die so in der propagierten Änderungsbeschreibung nicht vorhanden sind. Als Beispiel kann man hier die Integration eines Layout-Planungswerkzeuges für Fabriken mit einem Digitalen-Fabrik-System anführen. Beide verwalten Fertigungsressourcen, wobei das Layoutwerkzeug nur den Namen und die Koordinaten speichert, während das Digitale-Fabrik-System noch MTTR (Meantime to repair) und MTBF (Meantime between Failure) benötigt. Diese Informationen können beispielsweise von einer MaschinenInfoDB, die Produktbeschreibungen zu Ressourcen verwaltet, kommen. Der Ansatz basiert auf der Einbindung von Datendiensten, deren Daten mit SQL, XQuery oder SOAP-RPC abgefragt werden können. Dies ermöglicht eine breite Unterstützung von Informationssystemen und Datenbanken.

Im zweiten Teil wurde untersucht, wie mehrere Änderungen in einer Abhängigkeit bzw. einem Propagationsprozess kombiniert werden können. Diese werden M-zu-N-Abhängigkeiten genannt. Mit diesen Abhängigkeiten können zwei unterschiedliche Arten von einfachen Geschäftsregeln implementiert werden. Beispiel für den ersten Fall: Wenn ein Kunde angelegt wurde und er innerhalb von 24 Stunden bestellt, wird ihm ein Rabatt von 25% gewährleistet. Zweiter Fall (negierte Geschäftsregel): Wenn ein

Kunde angelegt wurde und er nicht innerhalb von 20 Stunden bestellt, wird er an das Angebot erinnert. Vorteil der Implementierung dieses Ansatzes ist die Erhaltung von Microflows, da Prozesse verzögert gestartet werden, sobald alle Änderungsbeschreibungen schon vorhanden sind. Die M-zu-N-Abhängigkeiten stellen nicht die Kernaufgabe eines Propagationssystem dar. Deshalb sollte das M-zu-N-Modul nur integriert werden, wenn dieses auch benötigt wird.

Im dritten und letzten Teil werden noch zwei Varianten zur Verteilung des Propagationsmanagers diskutiert, die selbstorganisierte und die mit Einsatz eines Load-Managers. Dazu wurden zwei Einschränkungen zur freien Verteilung diskutiert: das Reihenfolgeproblem und die M-zu-N-Abhängigkeiten.

KAPITEL 5

Evaluation des Propagationssystems

Nachdem das Propagationssystem grundlegend konzipiert und mögliche Erweiterungen diskutiert wurden, soll in diesem Kapitel der Ansatz evaluiert werden. Es soll ein Einsatz in der Praxis untersucht werden, Leistungsmessungen des Propagationssystems mit typischen Anforderungen aus der Industrie gegenübergestellt und das System mit gängigen EAI-Produkten verglichen werden.

5.1 Praxistest

Innerhalb des Sonderforschungsbereichs 467, in dem das Propagationssystem entstand, gab es verschiedene Integrationsaufgaben, bedingt durch die Vielzahl von eingesetzten Systemen (z.B. der Planungstisch für die Layoutplanung von Fabriken), die in den einzelnen Teilprojekten entstanden sind. Drei der Systeme wurden integriert und diese Integration als Praxistest verwendet. Der Integrationsansatz wurden in [CHB+05] veröffentlicht.

5.1.1 Integrationsszenario

Der Praxiseinsatz des Propagationssystems findet in einem fiktiven Unternehmen statt, das im Folgenden „Schwäbische Pumpenwerke“ [WZ09] genannt wird. Das Unternehmen fertigt in zwei Produktsegmenten Großpumpen und Kleinpumpen. Die aktuelle Marktlage für die Pumpenwerke ist durch einen drastischen Umsatzrückgang bei Kleinpumpen und einen leichten Anstieg bei Großpumpen gekennzeichnet. Nach einer gründlichen Analyse kommt die Marketing-Abteilung zu dem Schluss, dass die Marktveränderungen langfristig sind und die Produktion entsprechend umgestellt werden muss. Die Unternehmensleitung beschließt daraufhin, die Produktion den neuen Anforderungen anzupassen. Ziel der Umstrukturierung ist, die Kapazität der Großpumpenfertigung bei gleichzeitigem Abbau der Kleinpumpenkapazität zu steigern. Dafür

müssen neue Maschinen und Montageplätze für die Großpumpenfertigung beschafft und in der Fertigungshalle aufgestellt werden. Die Zuständigkeit im Projekt liegt bei der Fabrikplanung, die zwei Untersysteme hat: Planungstisch und Montage-Konfiguration. Der Planungstisch ist zuständig für die Positionierung der Maschinen in der Fertigungshalle und die Montage-Konfiguration gestaltet die Montage-Arbeitsplätze [CHB+05]. Die Montage hat spezielle Anforderungen (z.B. an die Ergonomie der Montageplätze), die vom Planungstisch nicht beachtet werden. Deshalb wird die Fabrikplanung folgendermaßen realisiert: der Planungstisch legt das Layout für die Fertigungsmaschinen fest und für die Montage wird nur ein Bereich zugeordnet, der anschließend von der Montage-Konfiguration im Detail geplant wird. Der dazugehörige Planungsprozess sieht im ersten Schritt die Fabrikplanung mit Zuordnung des Montagebereichs und im zweiten Schritt die Montage-Feinplanung vor. Allerdings kann es notwendig sein, diese Prozessschritte iterativ zu wiederholen, da zum Beispiel der zugewiesene Montagebereich nicht ausreichend ist.

Zur Realisierung dieses Planungsprozesses müssen beide Werkzeuge miteinander integriert werden. Dafür wurde bei den Schwäbischen Pumpenwerken eine Lösung für die Digitale Fabrik eingeführt, die alle planungsrelevanten Daten speichert und die Daten den Planungswerkzeugen zur Verfügung stellt. Die beiden Werkzeuge werden über das Propagationssystem mit der Digitalen Fabrik integriert. Die Aufgabenaufteilung ist im Anwendungsszenario streng getrennt, sodass eine verteilte *Master-Update*-Situation [GHOS96] vorliegt. Bei diesem System können einzelne Objekte oder Einheiten nur in einem System geändert werden. Dadurch müssen auch keine Konflikte erkannt und aufgelöst werden, da keine auftreten können (vgl. Abschnitt 3.8).

5.1.2 Digitale Fabrik und ihre Werkzeuge

In diesem Abschnitt wird die Digitale Fabrik eingeführt sowie zwei Planungswerkzeuge: der Planungstisch und der Montage-Konfigurator.

5.1.2.1 Die Digitale Fabrik

Die *Digitale Fabrik* unterstützt die Planung von Produktionsprozessen und Produktionsanlagen und stellt damit ein digitales Abbild der Fabrik bereit. Dadurch wird eine schnellere und kostengünstigere Einführung von neuen Produkten mit einer hohen Planungssicherheit ermöglicht als ohne digitalem Abbild der Fabrik. Bei der Digitalen Fabrik steht allerdings nicht die Entwicklung des Produktes im Vordergrund, sondern die Prozesse, mit denen die Produkte produziert werden, sowie die Planung der Fertigungsressourcen. Diese Ressourcen führen die Prozesse aus. Der Kern der Digitalen Fabrik besteht aus einer Datenhaltung für Produkte, Prozesse und Ressourcen. Die Daten werden in sogenannten Projekten organisiert. In diesen Projekten werden alle Daten verwaltet, die zu einem Planungsprojekt gehören. Um diesen Kern kann eine Vielzahl von Werkzeugen angesiedelt werden, die diese Daten verwenden oder Daten bereitstellen, zum Beispiel Werkzeuge zur Planung des Fabriklayouts (z.B. Fabrikplanungstisch Abschnitt 5.1.2.2) sowie Montage (Abschnitt 5.1.2.3) oder auch Simulationswerkzeuge, die zur Beurteilung der geplanten Fabrik anhand verschiedener Kennzahlen (z.B.

Auftragserfüllung) dienen. Innerhalb des Sonderforschungsbereichs wurde der Delmia Process Engineer (DPE) von Delmia als Datenhaltungssystem für die Digitale Fabrik eingesetzt.

5.1.2.2 Fabrikplanungstisch

Der Fabrikplanungstisch [WvB01, WW02] ist eines der Werkzeuge, die im Sonderforschungsbereich 467 „Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienfertigung“ entstanden sind. Dieses Werkzeug dient zur partizipativen Planung des Fabriklayouts. Das Fabriklayout definiert dabei die räumliche Anordnung von Ressourcen in Fabrikhallen. Durch die partizipative Planung soll es auch Planungslaien (z.B. Werkern) ermöglicht werden, bei Umstrukturierungen oder Neuplanungen mitzuwirken. Ziel ist dabei eine höhere Akzeptanzrate für die Umstrukturierung bei der Belegschaft zu erreichen. Diese partizipative Planung wird durch eine intuitive Planungsumgebung mit einer 2D- und 3D-Ansicht möglich. Die Ressourcen können mittels Klötzen, die mit einer reflektierenden Oberfläche versehen sind, verschoben werden, was über eine Kamera erfasst wird.

Die Daten der Ressourcen sind in einer Access Datenbank gespeichert und können über einen ODBC-Treiber ausgelesen und verändert werden. Die Software des Planungstischs verfügt über keine API. Ein zu entwickelnder Adapter (siehe auch Abschnitt 3.11) kann deshalb nur eine Datenbankschnittstelle verwenden.

5.1.2.3 Montage-Konfigurator

Auch der Montage-Konfigurator ist, wie der Fabrikplanungstisch (Abschnitt 5.1.2.2) im Sonderforschungsbereich 467 entstanden. Dieses spezialisierte Werkzeug dient der Planung und Gestaltung der Montage-Arbeitsplätze [CHB+05].

Die Montage entspricht dem letzten Produktionsschritt bei der Erstellung eines Produktes, bei dem mehrere Einzelteile kombiniert werden. Um Montage-Arbeitsplätze zu gestalten und zu planen, werden mehrere Eingabeparameter benötigt. Als Erstes muss die Fläche, auf der die Montage-Arbeitsplätze positioniert werden können, festgelegt werden. Diese Fläche wird durch den Fabrikplanungstisch (Abschnitt 5.1.2.2) zugeteilt. Außerdem werden Informationen über alle für die Montage relevanten Ressourcen benötigt, die durch die Digitale Fabrik bereitgestellt werden. Eine weitere wichtige Information für die Planung stellen die Montageprozesse dar, die ebenfalls aus der Digitalen Fabrik bezogen werden können. Aus diesen Informationen gestaltet der Montage-Konfigurator die Montage vom Layout der Montage-Arbeitsplätze bis zur Gestaltung einzelner Arbeitsplätze, bei der auch die Ergonomie betrachtet wird.

5.1.3 Integrationsplattform

Die Integrationsplattform besteht aus einer Digitalen-Fabrik-Lösung und zwei Planungswerkzeugen (Planungstisch und Montage-Konfigurator, Abbildung 5.1). Die einzelnen Planungswerkzeuge implementieren unterschiedliche Schritte im Planungsprozess und sind voneinander abhängig. Das bedeutet, dass der Output eines Planungs-

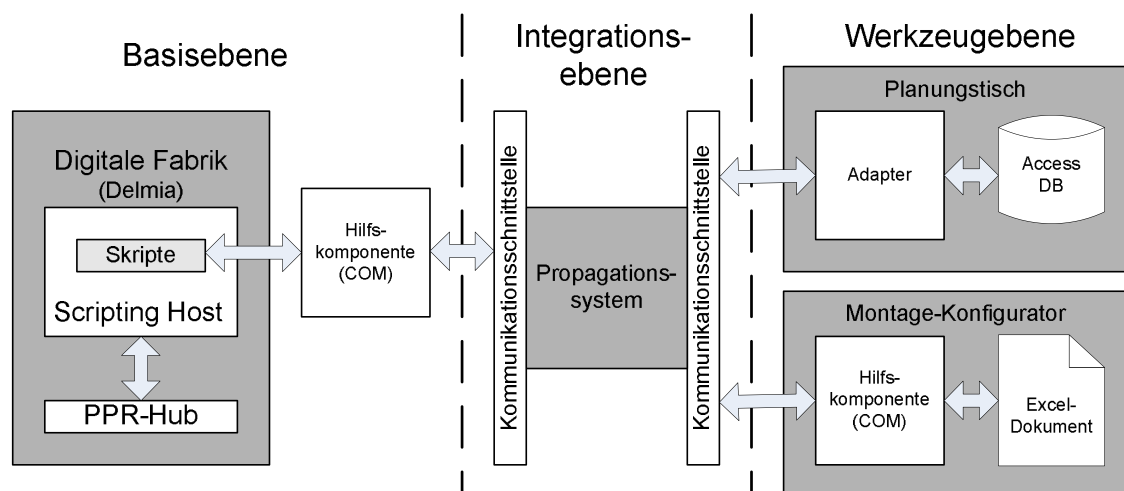


Abbildung 5.1: Architektur der Integrationsplattform

werkzeugs Input des anderen ist, da sich die Werkzeuge beim iterativen Planungsprozess abwechseln.

Die Digitale-Fabrik-Lösung stellt dabei Daten für den gesamten Planungsprozess bereit, d.h. es können noch weitere Planungswerkzeuge von der Digitalen-Fabrik-Lösung mit Daten versorgt werden oder Daten für sie bereitstellen.

Integriert werden die Planungswerkzeuge mit der Digitalen-Fabrik-Lösung (im vorliegenden Fall das Produkt von Delmia) anhand des hier konzipierten und implementierten Propagationssystems. Die Architektur der Integrationsplattform ist in Abbildung 5.1 illustriert. Um die Werkzeuge an das Propagationssystem anbinden zu können, wurden Adapter (vgl. Abschnitt 3.11) entwickelt. Diese erzeugen aus geänderten Daten Änderungsbeschreibungen und leiten diese über Warteschlangen an das Propagationssystem weiter. Empfangene Änderungsbeschreibungen müssen analysiert und im jeweiligen System angewendet werden. Der Adapter für die Digitale-Fabrik setzt sich aus einer Hilfskomponente und Skripten zusammen. Durch die Realisierung der Hilfskomponente mit der Microsoft COM-Technolgie, kann sie einfach durch die Skripte im Skripting Host eingebunden werden. Die COM-Technologie (*Component Object Model*) erlaubt eine Kommunikation zwischen Prozessen, aber auch das Einbinden von DLLs in Prozessen. Die Skripte laufen im *Windows Scripting Host* ab und greifen über eine Schnittstelle auf den Datenkern (PPR-Hub) zu. Die Hilfskomponente übernimmt dabei die Kommunikation mit den Warteschlangen des Propagationssystems und die Skripte übernehmen die Analyse und Verarbeitung der Änderungsbeschreibungen. Die Kommunikation erfolgt über die vom Warteschlangenprodukt bereitgestellte Kommunikationsschnittstelle. Ein ähnliches Vorgehen wird bei dem Montage-Konfigurator verwendet, der seine Daten in Excel verwaltet. Dabei wird dieselbe Hilfskomponente verwendet und die Analyse und Verarbeitung von Änderungsbeschreibungen mittels VBA-Makros realisiert. Ebenfalls VBA ermöglicht das Einbinden von COM-Komponenten. Der Planungstisch hat einen Adapter, der speziell für den Planungstisch entwickelt wurde. Dieser erkennt Änderungen in der Access DB und gibt sie direkt an die Eingangswar-

teschlange des Propagationssystems weiter. Außerdem liest er Änderungen aus seiner Warteschlange und wendet die Änderungsbeschreibungen auf die Access Datenbank an.

Obwohl es sich bei der Digitalen-Fabrik-Lösung (Delmia) um eine Hub-and-Spoke-Architektur handelt (zentrale Datenbank), ist es nicht sinnvoll, die einzelnen Planungswerkzeuge direkt anzubinden. Um die Daten der Werkzeuge mit Delmia zu intergrieren, eignet sich ein Integrationssystem, wie das hier vorliegende Propagationssystems. Dadurch bleiben die einzelnen Werkzeuge und auch Delmia unabhängig voneinander. Änderungen in den Datenmodellen können innerhalb des Propagationssystem behandelt werden, ohne dass die einzelnen Werkzeuge angepasst werden müssen. Gerade Digitale-Fabrik-Lösungen sind auf eine hohe Anpassungsmöglichkeit ihrer Modelle ausgelegt. Des Weiteren kann durch eine lose Kopplung einzelne Werkzeuge leichter ersetzt werden.

5.1.4 Integration der Digitalen Fabrik und des Planungstisches

Um den Planungstisch anzubinden, müssen zwei Richtungen realisiert werden: von der Digitalen-Fabrik zum Planungstisch und zurück. Bei der ersten Richtung werden Daten von der Digitalen-Fabrik exportiert. Beim Export werden zuerst alle Fertigungsressourcen eines Projektes zum Planungstisch übertragen. Durch diesen Ansatz muss der Planungstisch nicht alle Projekte vorhalten, sondern die Digitale-Fabrik-Lösung kann sie bei Bedarf bereitstellen. Nachdem die Daten exportiert wurden, wird der Planungsschritt im Planungstisch durchgeführt. Danach wird schließlich auch die zweite Richtung benötigt, das Zurückspielen von Änderungen, sodass die Digitale Fabrik wieder auf dem aktuellen Stand ist. Die Erkennung der Änderungen soll automatisch erfolgen. Der Zeitpunkt der Erkennung muss allerdings manuell angestoßen werden, da nicht automatisch erkannt werden kann, wann der Planungsschritt abgeschlossen ist und die Daten konsistent sind. Bei Planungssystemen können innerhalb des Planungsprozesses Phasen existieren, in denen die Daten inkonsistent sind und nicht den Erwartungen der Zielsysteme entsprechen. Ist eine Planung abgeschlossen, befindet sie sich wieder in einem konsistenten Zustand. Deshalb sollte in den inkonsistenten Phasen nicht synchronisiert werden.

Um Änderungskonflikte zu vermeiden, wurde eine Master-Update-Situation verwendet, bei der nur der Planungstisch Positionen von Ressourcen ändern darf, diese aber nicht direkt in der Digitalen-Fabrik-Lösung verändert werden. Auf der anderen Seite ändert der Planungstisch keine weiteren Daten der Ressource.

Es ergaben sich bei der Integration der beiden Systeme folgende Schwierigkeiten, die über die Schema-Heterogenität hinausgingen. Die erste war die Anordnung und Positionierung von Ressourcen in beiden Systemen. Die Digitale-Fabrik-Lösung verwendet eine Ressourcenhierarchie mit relativen Koordinaten zur übergeordneten Ressource. Der Planungstisch dagegen verwendet eine flache Struktur mit absoluten Koordinaten. Der Unterschied zwischen beiden Systemen ist in Abbildung 5.2 dargestellt. Um die Richtung von der Digitalen-Fabrik aus zu realisieren, müssen absolute Koordina-

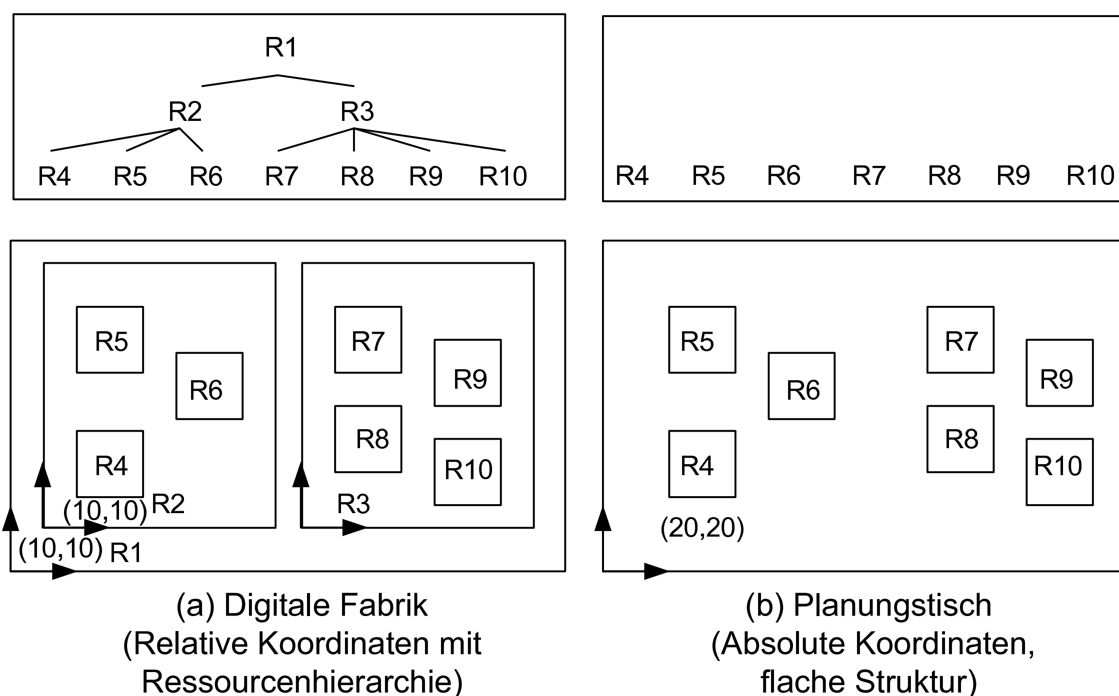


Abbildung 5.2: Gegenüberstellung der Koordinatensysteme von Digitale-Fabrik-Lösung und Planungstisch

ten errechnet werden, indem in der Hierarchie von der aktuellen Ressource bis zur Wurzel durchlaufen wird. Bei der Rückrichtung müssen die Änderungen auf die Positionen angewendet werden. Hierbei muss zwischen dem Erzeugen einer Ressource und dem Ändern einer Ressource unterschieden werden. Beim Erzeugen wird die Ressource zunächst direkt in der Fabrikhalle abgelegt, die die absoluten Koordinaten hat (die Wurzel). Eventuell kann es notwendig sein, die Ressource in einer untergeordneten Ressource zu organisieren. Dies kann von Hand oder semi-automatisch erfolgen, indem dem Planer eine Zuordnung anhand der Koordinaten vorgeschlagen wird. Dabei wird untersucht, in welcher Ressourcenfläche sich die hinzugefügte Ressource befindet. Wird eine Ressource umstrukturiert müssen neue Koordinaten errechnet werden, die der gleichen Position entsprechen. Diese Koordinatenermittlung ist notwendig, da die Ressource mit einer neuen Vaterressource auch ein neues Koordinatensystem bekommt. Während der Umstrukturierung müssen auch neue Koordinaten ermittelt werden. Bei der Änderung einer Ressource wird eine relative Verschiebung im Propagationssystem aus Davor- und Danach-Zustand berechnet. Diese relative Verschiebung ist in beiden Systemen gleich und kann deshalb auch auf die relativen Koordinaten angewendet werden. Allerdings kann die Verschiebung einer Ressource dazu führen, dass sie umstrukturiert werden muss, d.h. einer anderen übergeordneten Ressource zugeteilt wird. Dies kann wieder semi-automatisch erfolgen. Die Ressource kann der Ressource zugeordnet werden, in deren Fläche sie sich befindet.

Die zweite Schwierigkeit bestand darin, dass die Koordinatenursprünge an unterschiedlichen Positionen sind. Um dies auszugleichen, musste eine einfache Koordina-

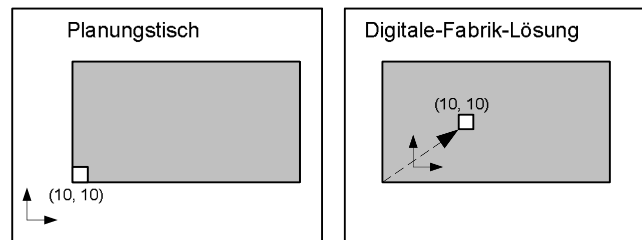


Abbildung 5.3: Ermittlung des Translationsvektors

tentranslation durchgeführt werden. Allerdings war die Ermittlung des Translationsvektors eine Herausforderung, da die Koordinatenursprünge nur implizit vorhanden sind. Deshalb wurde die Ermittlung folgendermaßen gelöst. Man nimmt einen markanten Punkt (z.B. eine Ecke der Fabrikhalle) und ermittelt die Koordinaten; im zweiten System wird von diesen Koordinaten bis zum markanten Punkt gemessen. Dies ergibt den Translationsvektor (siehe Abbildung 5.3). Allerdings muss beachtet werden, dass der Translationsvektor abhängig von der Integrationsrichtung ist, wobei in die andere Richtung einfach der Gegenvektor verwendet wird.

5.1.5 Integration der Digitalen-Fabrik und des Montage-Konfigurators

Die Digitale-Fabrik-Lösung verwaltet auch die Daten für den Montage-Konfigurator. Deshalb werden am Anfang eines Montage-Konfigurationsprojektes die notwendigen Daten aus der Digitalen-Fabrik-Lösung exportiert. Bei den notwendigen Daten handelt es sich um alle montagerelevanten Ressourcen und die zur Verfügung gestellte Fläche. Nachdem der Export von Delmia (Digitale-Fabrik-Lösung) erfolgte und vom Propagationssystem verarbeitet wurde, können die Daten vom Montage-Konfigurator importiert werden. Dafür wird von Excel ein VBA-Makro angestoßen. Dieses Makro holt sich die exportierten Daten aus der entsprechenden Warteschlange und trägt die Daten in die Excel-Arbeitsmappe ein.

Im Konfigurationsschritt werden die Montage-Arbeitsplätze und die Anordnung der Arbeitsplätze geplant. Ist dieser Schritt abgeschlossen, werden die Änderungen in die Digitale-Fabrik-Lösung zurückgespielt.

5.1.6 Schlussfolgerungen

Die im Sonderforschungsbereich erfolgreich durchgeführte Integration der Informationssysteme Digitale-Fabrik-Lösung, Planungstisch und Montage-Konfigurator hat die Einsetzbarkeit des Propagationssystems in der Praxis unter Beweis gestellt. Teilweise wurden aber lange Synchronisationszeiten festgestellt, da eine Vielzahl von Objekten exportiert wurde. Dies ist für die geringe Häufigkeit von Synchronisationen zwischen den Werkzeugen und die Dauer im unteren Minutenbereich für den Export durchaus verkraftbar. Allerdings muss angemerkt werden, dass der Export von gesamten Datenbeständen nicht im Konzept vorgesehen ist und dafür eigentlich andere Technologien

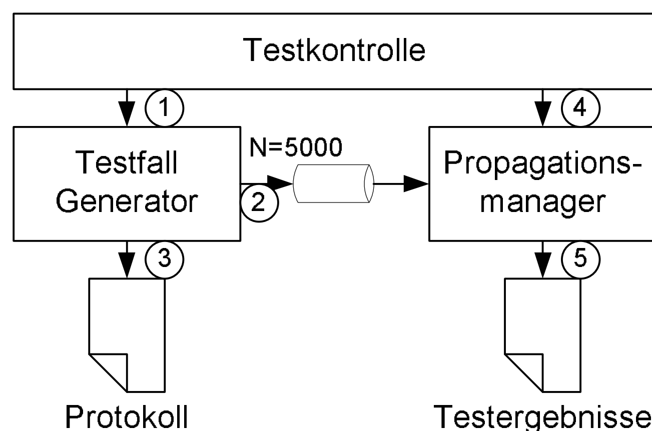


Abbildung 5.4: Aufbau der Messumgebung

verwendet werden sollten, die für den Austausch von Massendaten geeigneter sind. Die Performance des Propagationssystems wird im nächsten Abschnitt genauer untersucht.

5.2 Evaluierung der Performance

Um das vorgeschlagene Propagationssystem zu beurteilen, ist es wichtig, die Leistung dieses Systems zu kennen. Die Leistung wird anhand des Durchsatzes an propagierten Veränderungen und der durchschnittlichen Verarbeitungsdauer erfasst. Beide Kennzahlen werden in Abhängigkeit von der Größe des Geschäftsobjektes erfasst. Für die Messungen wurde als Geschäftsobjekt ein Kundenauftrag gewählt, da dieser durch eine beliebige Anzahl von Auftragspositionen, eine beliebige Größe annehmen kann. Die Verarbeitungsdauer und der Durchsatz sind abhängig von vielen Größen, die in diesem Abschnitt genauer betrachtet werden sollen. Als Erstes wird die Messmethodik vorgestellt. Danach wird auf die Testumgebung (verwendeter Rechner und Systeme) und auf die Realisierung der zuverlässigen Broadcast-Warteschlange (vgl. Abschnitt 3.7.2.3) eingegangen. Schließlich werden eine Reihe von Messungen des Systems mitsamt ihren Ergebnissen vorgestellt, die danach mit Anforderungen aus der Industrie verglichen werden.

5.2.1 Messmethodik

In Abbildung 5.4 ist die grundsätzliche Messumgebung dargestellt. Die Komponente *Testkontrolle* steuert und kontrolliert die Testumgebung. Bevor ein Testfall gestartet wird, stellt die Testumgebung sicher, dass alle Warteschlangen leer sind, d.h. keine Einträge haben, so dass die Umgebung in einem definierten Zustand ist. Danach startet die Komponente *Testkontrolle* die Komponente *Testfall-Generator* (1), der dann die Eingangswarteschlange des Propagationsmanagers mit 5000 Änderungsbeschreibungen füllt (2). Es wurde ein große Anzahl gewählt, damit die Messergebnisse genauer sind und eventuelle Ausreißer nicht so stark ins Gewicht fallen. Nachdem alle Nachrichten

erzeugt wurden, wird ein Protokoll geschrieben, welches die Statistik über die erzeugten Nachrichten enthält (3). Wurden diese Aufgaben erledigt, kann der *Propagationsmanager* durch die *Testkontrolle* gestartet werden (4). Dieser findet nun eine volle Eingangswarteschlange vor, die er abarbeitet. Dadurch wird sichergestellt, dass der maximale Durchsatz tp ermittelt wird. Dieser ergibt sich aus dem Zeitpunkt t_s (Verarbeitungsstart der ersten Änderung), dem Zeitpunkt t_e (Verarbeitungsende der letzten Änderung) und n der Anzahl von Änderungen:

$$tp = \frac{n}{t_e - t_s} \tag{5.1}$$

Um sicherzustellen, dass keine Effekte, wie zum Beispiel das Laden von Klassen, das Testergebnis beeinflussen, werden die ersten 50 Änderungsnachrichten nicht berücksichtigt, d.h. der Propagationsmanager setzt die Messungen zurück. Hat der Propagationsmanager alle Änderungsnachrichten abgearbeitet, protokolliert er die Ergebnisse in einer Datei (5).

Um eine hohe Qualität der Testergebnisse zu garantieren, wird ein solcher Test dreimal wiederholt und die Ergebnisse gemittelt, nachdem eventuelle Ausreißer (mehr als 20% Abweichung) entfernt wurden. Durch die Protokollierung der Ausgaben jeder beteiligten Komponente können aufgetretene Fehler erkannt werden.

5.2.2 Testumgebung

Der Performancetest wurde auf einem Zweiprozessor Intel-Xeon-System mit je 2 Gigahertz und einem Hauptspeicher von 2 Gigabyte durchgeführt. Implementierungssprache war Java 1.5, die auch die Laufzeitumgebung stellte. Als Warteschlangensystem wurde IBM Websphere MQ 5.7 verwendet, auf das mit dem Java Message Service (JMS) zugegriffen wurde. Für das Parsen von XML wurde Apache-Xerces 2.6.2 und für die Transformation mittels XSLT und XQuery[1] wurde Saxon 8.1.1 eingesetzt. Für die Erzeugung von SOAP-Nachrichten für den Aufruf von externen Datendiensten wurde SAAJ (SOAP with Attachments API for Java) in der Version 1.2.1 verwendet.

5.2.3 Realisierung der zuverlässigen Multicast-Warteschlange

Da die zuverlässige Multicast-Warteschlange, so wie sie in Abschnitt 3.7.2.3 eingeführt wurde, nicht von den gängigen Warteschlangen-Herstellern unterstützt wird und auch nicht in JMS angedacht ist, wurde eine Simulation implementiert. Sie ist durch eine Nachricht pro Multicast-Empfänger gekennzeichnet. Dies bedeutet aber, dass z.B. bei einer 1-zu-5-Abhängigkeit mit 5 Multicast-Empfängern auch 5 Nachrichten verschickt werden müssen, was eine langsamere Ausführung zur Folge hat.

[1] XQuery wurde allerdings in diesem Test nicht eingesetzt.

5.2.4 Testfälle

Um die Propagationsumgebung zu beurteilen, werden Änderungsnachrichten unterschiedlichster Größe erzeugt. Diese Änderungsnachrichten stellen Änderungen eines Geschäftsobjektes *Kundenauftrag* dar. Dieser kann erstellt, gelöscht oder geändert werden. Bei der Änderung können zum Beispiel die Adresse des Kunden angepasst werden oder Auftragspositionen storniert werden. Die Geschäftsobjektgröße wird hauptsächlich durch die Anzahl von Auftragspositionen reguliert. Ausnahme hiervon sind die kleinen Größen (200 bzw 500 Bytes), da hier selbst ohne Positionen die Geschäftsobjekte noch größer als die angestrebte Größen sind. Deshalb wurden weitere Informationen des Kunden weggelassen. Daraus folgt, dass Änderungsbeschreibungen mit 200 und 500 Bytes in der Praxis eher selten vorkommen dürften. Größen ab 1000 sind realistischer. Die oberen Grenzen der Tests sind bei 10000 und 20000 Bytes. Letztere erreicht 46 Auftragspositionen. Die Änderungen werden mit den drei Änderungsarten (*create*, *update* und *delete*) verschickt, wobei sich bei der Änderungsart *update* noch die Größe der Änderungsbeschreibung verdoppelt, was mit den zwei mitgegebenen Zuständen zusammenhängt (vgl. Abschnitt 3.2.2).

Da es sich um ein Propagationssystem für das heterogene Umfeld handelt, sollte die propagierte Änderungsbeschreibung sich von der empfangenen unterscheiden (Heterogenität). Die Transformation verändert die Namen der Bezeichner und passt die Struktur an, indem die Hierarchie der Elemente verändert wird. Außerdem wird die Summe des Einkaufs berechnet, welche sich aus den einzelnen Mengen und Einzelpreisen aufsummiert über alle Auftragspositionen ergibt.

Das Transformationsskript wurde in XSLT 2.0 entwickelt und transformiert den Kundenauftrag zum CustomerOrder.

Wenn nicht anderes angegeben, so wird mit einer Beschränkung von maximal 10 Propagationsprozessen zu jedem Zeitpunkt gearbeitet. Die Cache-Hitrate wird konstant auf 100 Prozent gehalten, da eine Vergleichbarkeit nur durch einen konstanten Wert gewährleistet ist und dieser durch die unterschiedlichsten Testfälle nur schwer einhaltbar ist, wenn es sich nicht um die zwei extremen Werte 0 oder 100 Prozent handelt. Die Hitrate sagt aus mit wieviel Prozent der Cache Metadaten (z.B. Propagationsskript) liefern konnte und nicht auf das Repository zurückgegriffen werden musste. Die Hitrate wird im Testfall 5 untersucht. Die Propagationsskripte befinden sich nicht im Debug-Modus (vgl. Abschnitt 3.5.3.2), d.h. die Überprüfung auf Schema-Konformität der propagierten Änderungen findet nicht statt.

Im Folgenden werden neun Testfälle vorgestellt, die zeigen, welche Performance das System unter verschiedenen Bedingungen aufweist.

5.2.4.1 Testfall 1: Update-Häufigkeit

Das Ziel von Testfall 1 ist die Ermittlung des Einflusses der Update-Häufigkeit auf die Performance. Die Änderungsart *Update* stellt eine besondere Rolle im Propagationssystem dar, da diese Änderungsart zwei Zustände hat (vgl. Abschnitt 3.2.2). Dies hat zur Folge, dass beim Parsen und beim Transformieren mehr Daten verarbeitet werden müssen. Da das Schema für beide Zustände gilt, muss jeder Zustand überprüft werden.

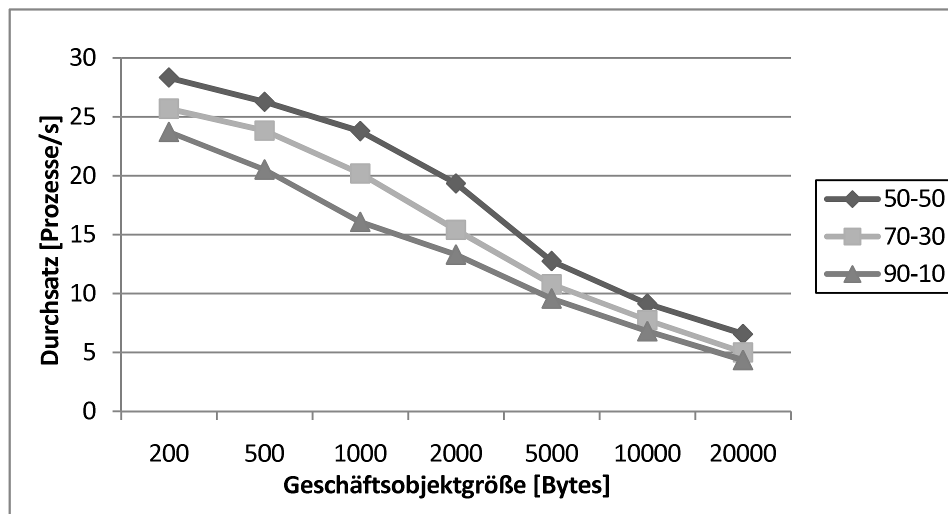


Abbildung 5.5: Update-Häufigkeit und ihr Einfluss auf den Durchsatz

In der Realisierung des Propagationssystems werden beide Zustände in einem XML-Dokument abgebildet. Das bedeutet, dass dieses XML-Dokument zuerst geparst und die beiden Zustände extrahiert werden. Danach kann jeder Zustand validiert werden. Das Validieren von DOM-Bäumen wird von DOM-Level-3 unterstützt. Da aber die XML-Werkzeuge (z.B. XQuery-Engine) zum Zeitpunkt der Messung nur DOM-Level-2 unterstützten, muss das Parsen von DOM-Bäumen simuliert werden, indem der Baum serialisiert und geparst wird. Dies führt allerdings zu Performanceeinbußen.

Es lässt sich zur Update-Häufigkeit anmerken, dass die Änderungsart *update* bei Stammdaten in den meisten Fällen häufiger auftritt als die beiden anderen Änderungsarten (*create* und *delete*). Diese Aussage gilt allerdings nur für Stammdaten (z.B. Kunde). Bei Bewegungsdaten (z.B. Kundenauftrag) finden weniger Updates statt. Um den Einfluss der Update-Häufigkeit zu untersuchen, wurden drei Fälle ausgewählt. Im ersten Fall wird eine Update-Häufigkeit von 50% verwendet, wobei sich der Rest gleichmäßig auf *creates* und *deletes* verteilt. Der zweite Fall untersucht 70% und der dritte 90%. Das verwendete Propagationsskript ist ein einfaches und setzt sich aus einem Input, einer Transformation und einer Propagation zusammen.

Die Ergebnisse der Untersuchung sind in zwei Diagrammen dargestellt, die den Einfluss auf den Durchsatz (Abbildung 5.5) und die Prozesszeiten (Abbildung 5.6) vermitteln. Letztere repräsentiert die Zeit, die der Propagationsprozess braucht, um ein Propagationsskript abzuarbeiten.

Es fällt auf, dass der Durchsatz im Messbereich logarithmisches Verhalten hat, denn die Kurve ist linear in einem Diagramm mit einer logarithmischen Skala, d.h. die Auswirkungen werden bei größerer werdenden Geschäftsobjekten geringer. Die Prozesszeiten haben im Messbereich dagegen ein lineares Verhalten, was aus den erfassten Messwerten geschlossen werden kann. Die Kurven laufen beim Durchsatz-Diagramm bei ansteigender Geschäftsobjektgröße zusammen, d.h. der Einfluss der Update-Häufigkeit wird geringer. Bei den Prozesszeiten gehen die Kurven auseinander, d.h. der Einfluss wird mit steigender Geschäftsobjektgröße stärker.

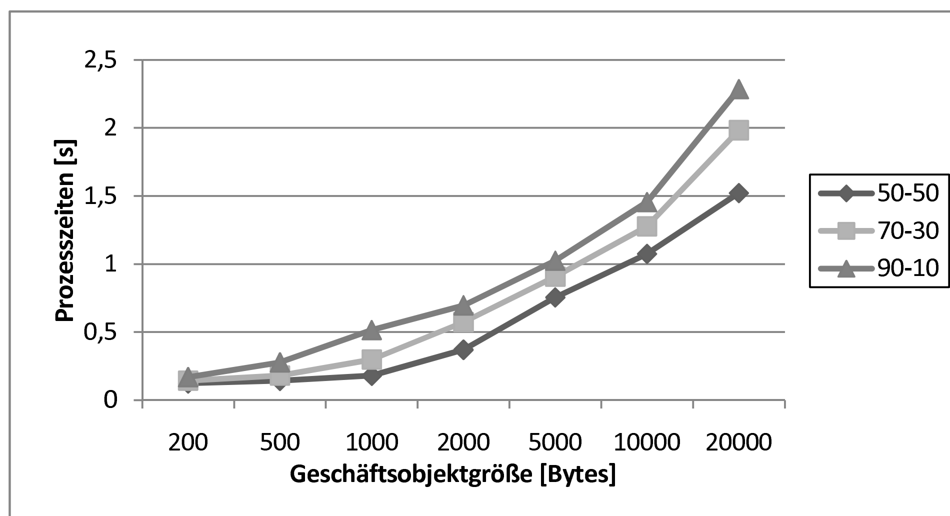


Abbildung 5.6: Update-Häufigkeit und ihr Einfluss auf die Prozesszeiten

Zu den möglichen Verbesserungen gehört das Umstellen auf XML-Werkzeuge, die DOM-Level-3 unterstützen, welche zum Zeitpunkt der Messung aber noch nicht zur Verfügung standen.

Für die nachfolgenden Tests wurde der Fall 70-30 gewählt, da von einer leicht höheren Update-Häufigkeit ausgegangen wurde.

5.2.4.2 Testfall 2: Begrenzung der nebenläufigen Propagationsprozesse

Ziel dieses Testfalls war die Ermittlung des Einflusses der Prozessbegrenzung auf die Performance. Das Propagationssystem verfügt über einen Prozesspool, der Propagationsprozesse zur Verfügung stellt. Ein solcher Propagationsprozess ist als Thread realisiert. Dadurch können Ressourcen, wie Repository-Verbindung und Warteschlangenmanager, wiederverwendet werden. Ebenfalls wird hier die maximale Anzahl der Prozesse beschränkt und dadurch können Überlastsituationen verhindert werden, bei denen die Gesamtleistung drastisch abnimmt.

Der Test wurde in zwei Unterfälle unterteilt. Zuerst wurde gemessen, wie sich das System verhält, wenn ein Propagationsprozess pro Änderung gestartet wird (siehe Abbildungen 5.7 und 5.8). Um die Last zu erhöhen, wurden im zweiten Unterfall 5 Prozesse pro Änderung gestartet (Abbildungen 5.9 und 5.10). Bei diesem Fall wird die Änderung an 5 verschiedene Zielsysteme propagiert. Jeder dieser Unterfälle wurde mit 5 Prozessbegrenzungen gemessen: maximal 1, 5, 10, 15 und 20 Prozesse gleichzeitig. Für das Propagationsskript wurde dasselbe Skript wie in Testfall 1 verwendet, d.h. eine Transformation und eine Propagation. Bei der Transformation handelt es sich um die Standardtransformation, wie in Abschnitt 5.2.4 beschrieben wurde. Es wurde die 70-30 Verteilung der Update-Häufigkeit von Testfall 1 verwendet.

Was im Unterfall 1 zuerst auffällt, ist dass mit zunehmender Anzahl von Prozessen der Performancegewinn beim Durchsatz (Abbildung 5.7) abnimmt. Der größte Performancegewinn wird noch von einem Prozess auf fünf Prozesse erreicht. Der Performan-

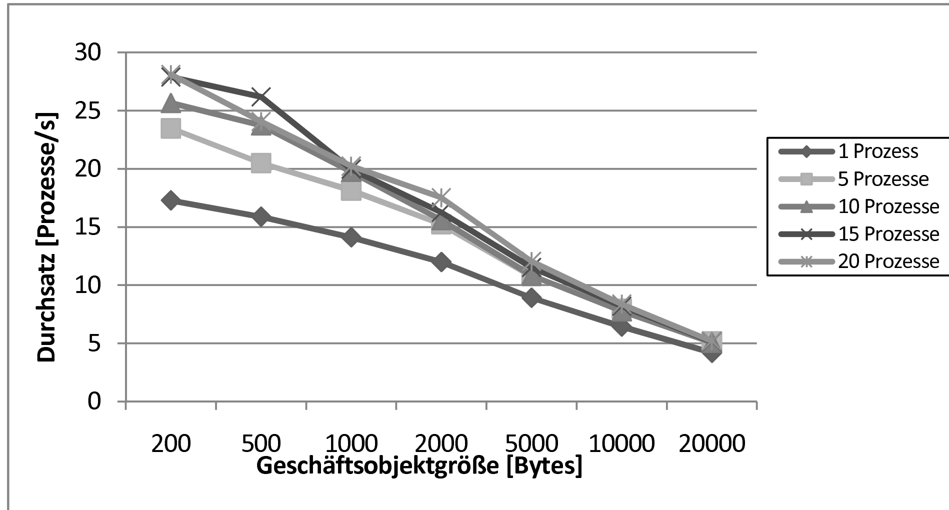


Abbildung 5.7: Einfluss der Prozessbegrenzung auf den Durchsatz (1 Prozess pro Änderung)

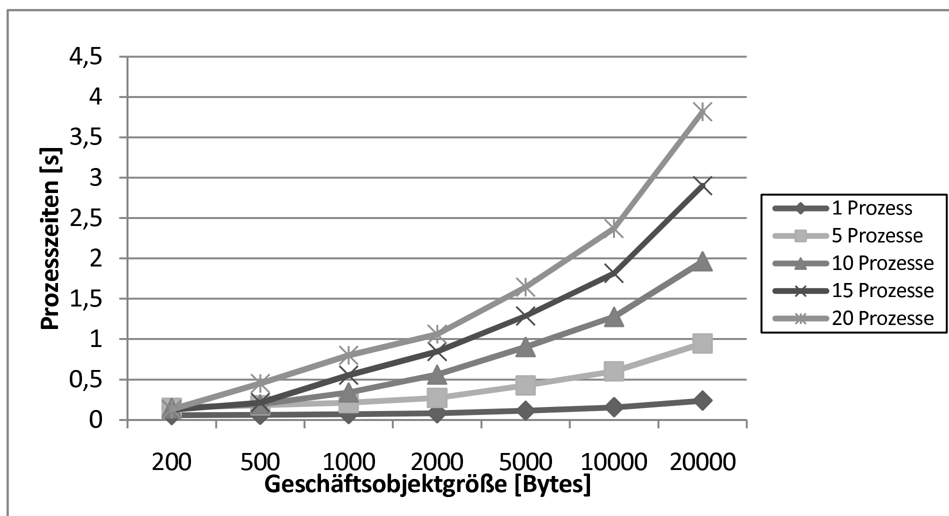


Abbildung 5.8: Einfluss der Prozessbegrenzung auf die Prozesszeiten (1 Prozess pro Änderung)

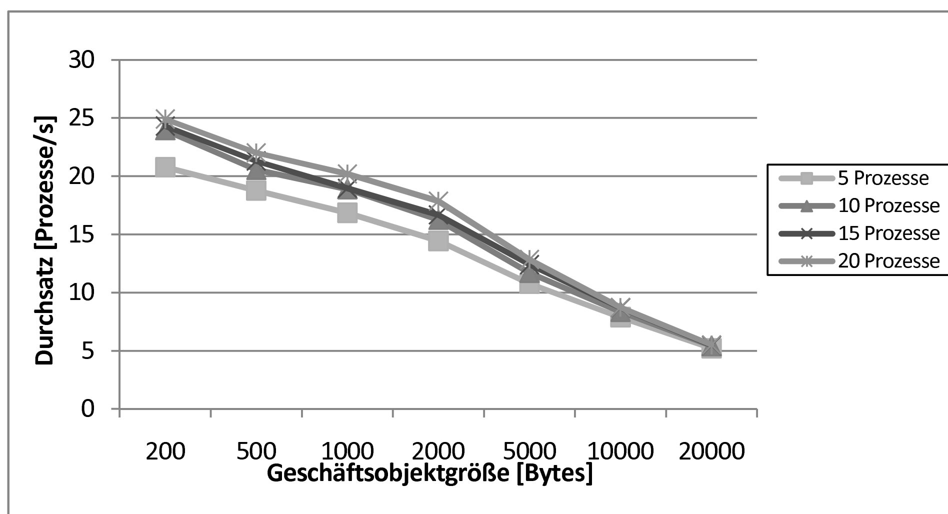


Abbildung 5.9: Einfluss der Prozessbegrenzung auf den Durchsatz (5 Prozesse pro Änderung)

cegewinn sinkt auch bei der Zunahme der Größe des Geschäftsobjektes, das geändert wird. Im Gegensatz dazu nehmen die Prozesszeiten in den hohen Prozessbegrenzungen kaum mehr zu. Dies könnte damit zusammenhängen, dass sich das System durch das Starten eines Prozesse pro Änderung selbst reguliert, da nicht alle frei verfügbaren Prozesse genutzt werden können.

Bei Unterfall 2 mit 5 gestarteten Prozessen pro Änderung ist das Ergebnis ähnlich, nur dass die Performance in diesem Fall leicht geringer ausfällt als im Unterfall 1. Das Durchsatzverhalten ist dabei in Abbildung 5.9 und die Prozesszeiten in Abbildung 5.10 dargestellt. Der große Abstand beim Durchsatz zwischen den 1 und 5 Prozessbegrenzungen, kommt dadurch zustande, dass bei 1 die Prozessbegrenzung unterhalb der Anzahl der zu startenden Prozessen pro Änderung ist.

Für die nachfolgenden Testfälle wird eine Prozessbegrenzung von 10 Prozessen verwendet, da diese einen guten Durchsatz bei akzeptablen Prozesszeiten hat.

5.2.4.3 Testfall 3: Anzahl der Transformationen

Ziel dieses Testfalls ist die Ermittlung des Einflusses der Transformationsanzahl pro Propagationsprozess auf die Gesamtperformance des Systems. Wichtig hierbei waren nicht die einzelnen Transformationszeiten, sondern wie sich mehrere Transformationen auf die gesamte Prozesszeit und auf den Durchsatz auswirken, da diese Werte entscheidend für die Integration sind. In einem Propagationsskript können vom Empfang bis zum Senden der Änderungen mehrere Transformationen ausgeführt werden. Dies ist dadurch begründet, dass die Komplexität der einzelnen Transformationen verringert werden soll, d.h. mehrere einfache Transformationen statt einer komplexen. Ein weiterer Grund ist ein allgemeines Zwischenformat, das als Grundlage für die Integration mit den einzelnen Systemen verwendet wird. Die Transformationen werden beschleunigt, da im Propagationssystem ein internes Format verwendet wird (Abschnitt 3.4.2), wo-

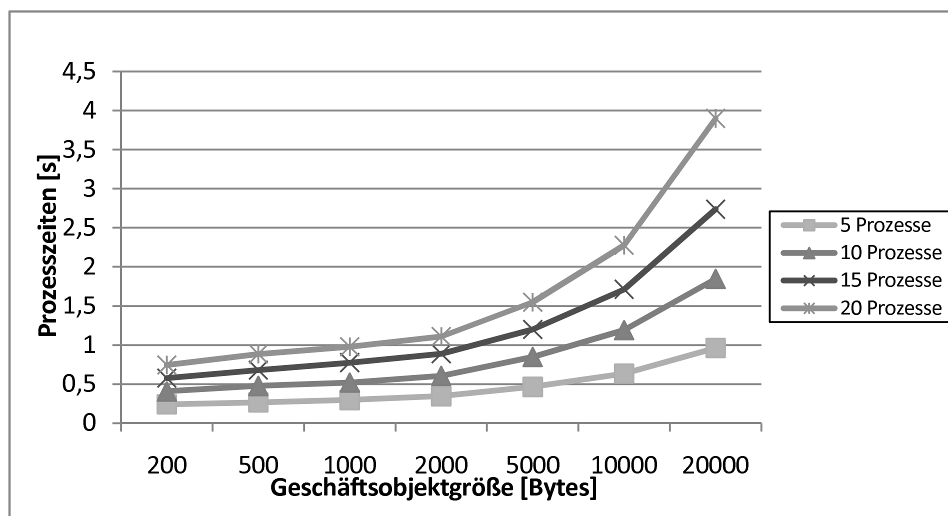


Abbildung 5.10: Einfluss der Prozessbegrenzung auf die Prozesszeiten (5 Prozesse pro Änderung)

durch die Zustände zwischen den einzelnen Transformationen nicht serialisiert werden müssen. Das interne Format des Prototypen ist DOM-Level-2.

In diesem Testfall wurden 5 Fälle untersucht: 1-5 Transformationen. Den Sonderfall „keine Transformation" wurde hier nicht untersucht, da für das Propagationssystem von einem heterogenen Umfeld und nicht von einem homogenen ausgegangen wird. Das Propagationsskript entspricht dem der vorangegangenen Testfälle bis auf die Anzahl der Transformationen. Das heißt, es enthält eine Input-Deklaration, 1-5 Transformationsbefehle und eine Propagation. Die Transformationen sind so definiert, dass sie zwischen Input- und Output-Format hin und her transformieren. Als weitere Parameter der Messungen wurden wieder die 70-30-Verteilung der Update-Häufigkeit (Abschnitt 5.2.4.1), die Begrenzung auf 10 Prozesse und die Cache-Hitrate 100% verwendet.

Als Ergebnis lässt sich anmerken, dass beim Durchsatz (Abbildung 5.11) bei größer werdenden Geschäftsobjekten der Einfluss der Transformationsanzahl abnimmt. Bei den Prozesszeiten (Abbildung 5.12) nimmt der Einfluss dagegen zu. Greift man nun wieder die Geschäftsobjektgröße von 1000 Bytes heraus, so ergibt sich bei 1 Transformation einen Durchsatz von 19,7 Prozesse pro Sekunde, während sich bei 5 Transformation ein Durchsatz von 10,7 ergibt, d.h. fast halbiert. Aber dennoch ist der Einfluss geringer als man annimmt. Die Prozesszeiten sind dagegen 325 ms und 915 ms, was dem dreifachen entspricht. Bei einer mittleren Geschäftsobjektgröße von 5000 Bytes beträgt der Durchsatz 11 Prozesse pro Sekunde bei 1 Transformation und 7 Prozesse bei 5 Transformationen, was ungefähr 36% weniger Durchsatz bedeutet. Bei den zwei Werten (1000 und 5000) in Bezug auf den Durchsatz fällt schon die Einflussverkleinerung auf, die auch in Abbildung 5.11 zu erkennen ist. Die Prozesszeiten sind bei 5000 Bytes 888 ms und 1423 ms, was 37% entspricht. Dies lässt sich dadurch erklären, dass zwischen den Kurven sich der Abstand kaum verändert, aber der prozentuale Einfluss geringer wird bei höheren Werten.

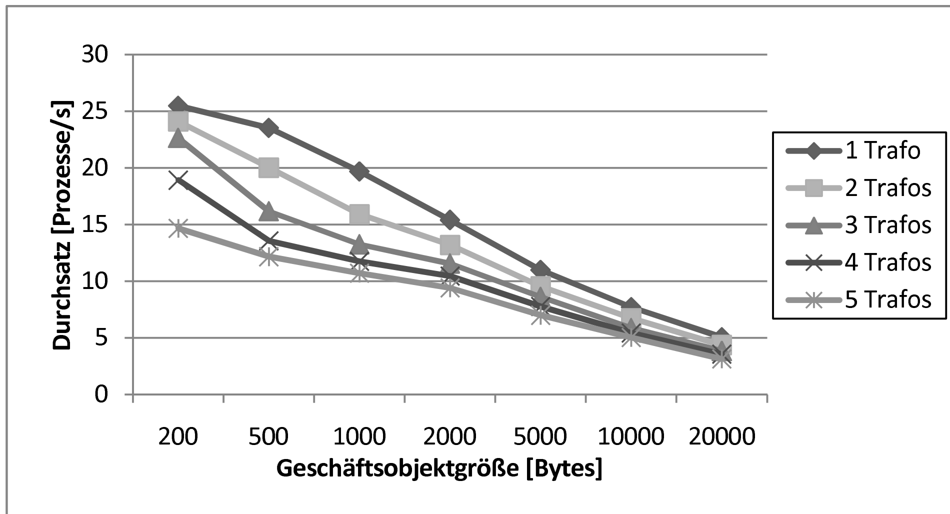


Abbildung 5.11: Einfluss der Transformationsanzahl auf den Durchsatz

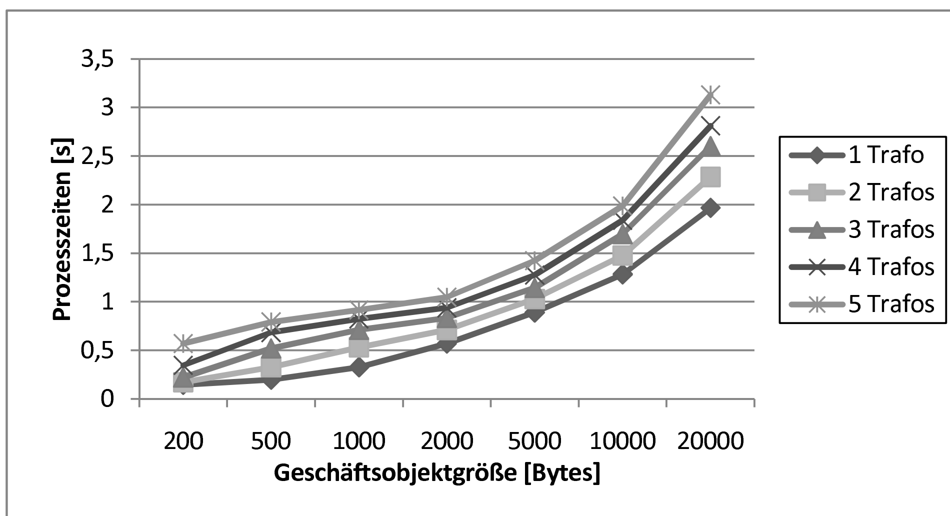


Abbildung 5.12: Einfluss der Transformationsanzahl auf die Prozesszeiten

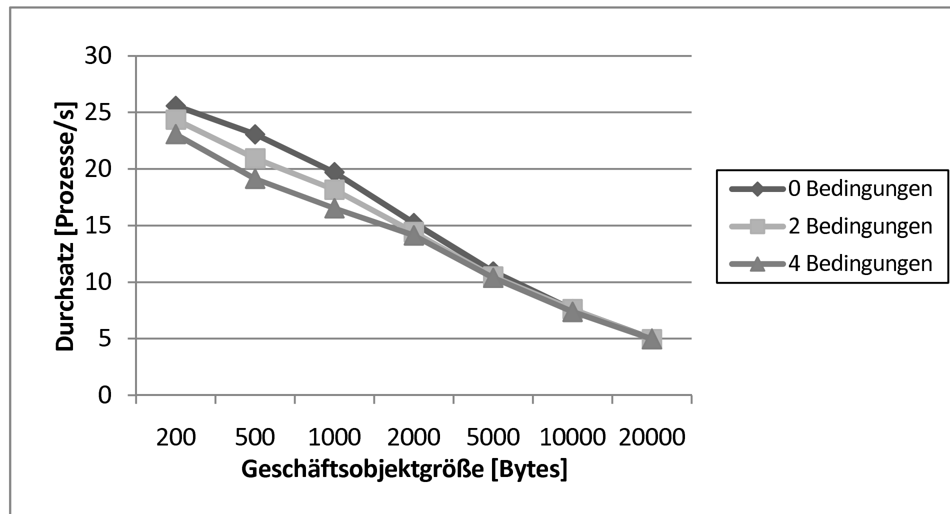


Abbildung 5.13: Einfluss der Bedingungsanzahl auf den Durchsatz

5.2.4.4 Testfall 4: Anzahl der Bedingungen

Testfall 4 ist dem Testfall 3 ähnlich. Allerdings ist hier das Ziel, den Einfluss der Bedingungsanzahl im Propagationsskript zu messen. Diese Bedingungen treten dann auf, wenn in bestimmten Fällen unterschiedliche Transformationen und Propagationen ausgeführt werden müssen oder wenn nur bei bestimmten Werten propagiert werden soll. Das entspricht der sogenannte Filter-Funktionalität, die durch bedingte Ausführung realisiert werden kann. Als Propagationsskript wurde das Standardpropagationsskript um die Anzahl von Bedingungen (maximal 4) erweitert. Die Bedingungen sind so gewählt, dass bei dem gegebenen Input der Pfad so durchlaufen wird, dass einmal transformiert und propagiert wird. Die einzelnen Bedingungen beziehen sich auf das Datum des Auftrags, wobei immer ein bestimmter Wert extrahiert und mit einer Zahl verglichen wird. Das heißt für die Bedingungen 1-4:

1. Extraktion des Jahres und Gleichheitsvergleich

2. Extraktion des Monats und Gleichheitsvergleich

3. Extraktion des Tags und Gleichheitsvergleich

4. Extraktion des Tags und Ungleichheitsvergleich.

Der Einfluss auf die Performance hat sich als äußerst gering herausgestellt und diese machen sich nur im mittleren Bereich bemerkbar. Der Einfluss auf den Durchsatz ist in Abbildung 5.13 und der Einfluss auf die Prozesszeiten ist in Abbildung 5.14 dargestellt. Der geringe Einfluss lässt sich auf das Vorliegen der Änderungsnachricht im internen DOM-Format zurückführen.

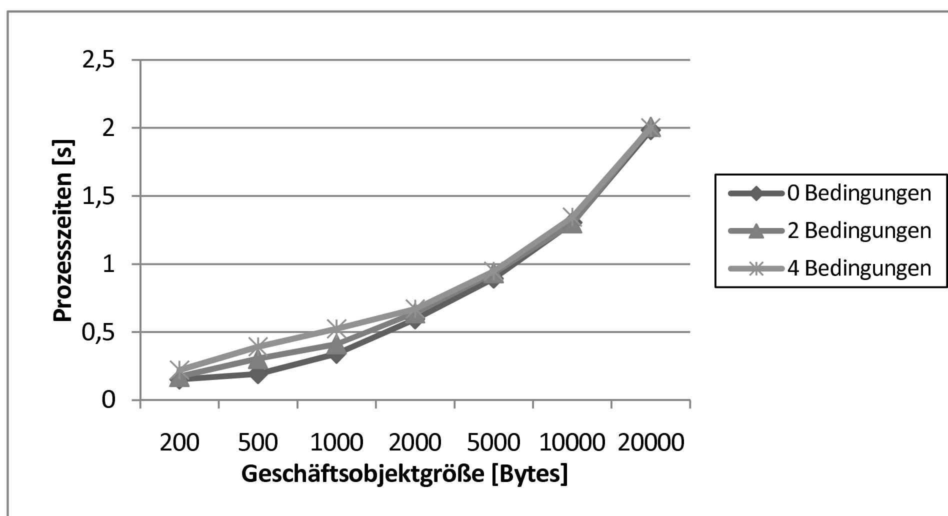


Abbildung 5.14: Einfluss der Bedingungsanzahl auf die Prozesszeiten

5.2.4.5 Testfall 5: Cache-Hitrate

In diesem Testfall soll der Einfluss der Verwendung eines Cache auf die Performance des Systems ermittelt werden. Durch den Cache können Zugriffe auf die Datenbank des Repositories verhindert werden und die Zugriffszeiten auf die Metadaten verkürzen sich. Dabei spielt die erreichte „Hitrate" ein große Rolle. Sind viele Abhängigkeiten vorhanden, dann können nicht alle Abhängigkeiten im Cache vorgehalten werden und es wird eine niedrige Hitrate erreicht. Dies kommt durch das Fehlen und Austauschen von Inhalten des Cache zustande.

Die Untersuchung unterteilt sich in verschiedene Hitraten, die zwischen den zwei Extremen 0% und 100% liegen. Diese zwei Extreme repräsentieren „kein Cache" und „ausreichend großer Cache". Für die anderen Testreihen wurden 25%, 50% und 75% als Hitrate angestrebt. Die angestrebte Hitrate ist die Hitrate, die für den Test vorgesehen wurde und die reale Hitrate ist die, die durch den Cache-Manager festgestellt wurde.

Bei der theoretischen Hitrate von 25% wurde schließlich eine Hitrate von 28% erreicht. Diese wurde durch 4 verschiedene Abhängigkeiten realisiert, die jeweils auf Änderungsbeschreibungen mit einem System-GOTyp-Paar reagieren. Die Verteilung wurde auf 25% pro Änderung festgelegt.

Bei einer angestrebten Hitrate von 50% wurde effektiv 46% erreicht. Hier wurden zwei verschiedene Änderungen mit je einer Abhängigkeit realisiert. Jede der zwei Änderungen kommt mit einer Wahrscheinlichkeit von 50% vor. Die angestrebte Hitrate von 75% wurde ebenfalls durch zwei Änderungen mit Abhängigkeiten erreicht. Diesmal wurde eine Verteilung der beiden Änderungen von 97% der einen Änderung und 3% der anderen Änderung festgelegt. Effektiv wurde hierbei eine Hitrate von 77% erreicht.

Für alle Abhängigkeiten wurde das Standardpropagationsskript wie bei den anderen Test verwendet. Das bedeutet für die Transformationsskripte, dass auch die Standardtransformation verwendet wurde. Es wurden sowohl der Durchsatz als auch die Prozesszeiten ermittelt.

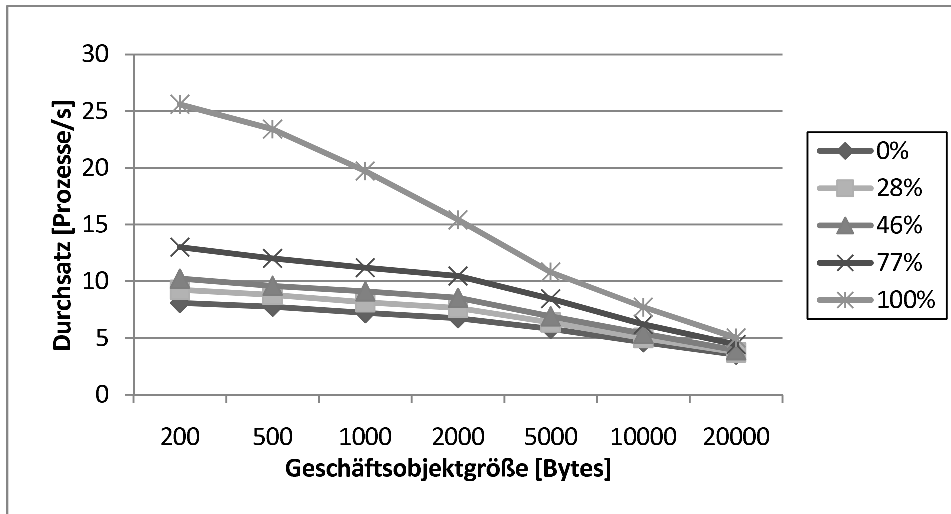


Abbildung 5.15: Einfluss der Cache-Hitrate auf den Durchsatz

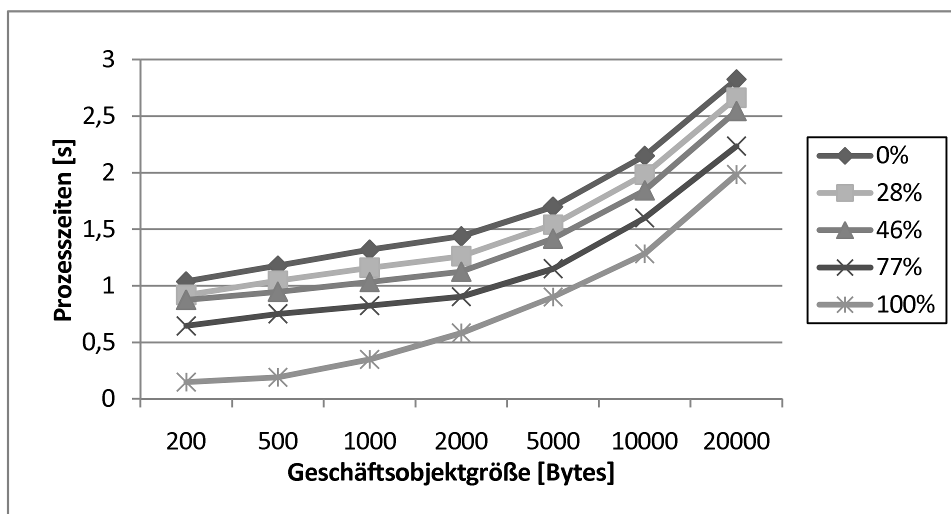


Abbildung 5.16: Einfluss der Cache-Hitrate auf die Prozesszeiten

Die beiden Messungen Durchsatz (Abbildung 5.15) und Prozesszeiten (Abbildung 5.16) ergaben, dass der prozentuale Einfluss bei steigender Geschäftsobjektgröße geringer wird. Beim Durchsatz ist dies auch beim realen Einfluss deutlich zu sehen. Bei kleinen und mittleren Größen des geänderten Geschäftsobjektes tritt ein optimaler Einfluss erst bei sehr hohen Werten ein, dann aber mit sehr großen Performancesteigerungen. Im Umkehrschluss bedeutet es, dass die Zugriffszeiten bei kleinen Geschäftsobjekten einen großen Einfluss haben.

Schaut man sich das Verhalten bei einer Geschäftsobjektgröße von 1000 Bytes genauer an, so wird bei 0% ein Durchsatz von 7,2 Änderungen pro Sekunde erreicht und bei 100% 19,71 Änderungen pro Sekunde, d.h. der Durchsatz erhöht sich um das 2,66-fache. Betrachtet man dagegen 10000 Bytes, so ergeben sich Durchsätze von 4,6 zu 7,71 Änderungen pro Sekunde, d.h. der Durchsatz steigert sich um das 1,67-fache, was deutlich geringer ist als die oben erreichte Performancesteigerung.

Die Cache-Hitrate hat Einfluss auf die Prozesszeiten, da der Propagationsprozess die Metadaten, die er benötigt, selber holt.

5.2.4.6 Testfall 6: Vergleich einer 1-zu-N- mit N x 1-zu-1-Abhängigkeiten

Mit Testfall 6 sollen die zwei Arten, eine Änderung an mehrere Zielsysteme zu propagieren gegenübergestellt werden. Einerseits ist es möglich, eine Änderung an N Zielsystemen mit N Propagationsskripten zu propagieren, die jeweils eine Propagation enthalten, d.h. es gibt N 1-zu-1-Abhängigkeiten. Anderseits können die Propagationen in einem Propagationsskript zusammengefasst werden, sodass eine 1-zu-N-Abhängigkeit entsteht. Allerdings muss hier erwähnt werden, dass die Semantik im Fehlerfall eine andere ist (vgl. Abschnitt 3.10.3), da jeder Abhängigkeit eine Transaktionssphäre zugeordnet ist. D.h. für die beiden Realisierungen, dass im ersten Fall eine Propagation und im zweiten Fall alle Propagationen im Fehlerfall zurückgesetzt werden.

Die Propagationsskripte wurden so gehalten, dass sie dem jeweiligen Vergleichsszenario entsprechen. D.h. für das Szenario mit zwei Zielsystemen, dass bei zwei Propagationsskripten (1-zu-1-Abhängigkeiten) jeweils eine Transformation und eine Propagation enthalten sind. Bei einem Propagationsskript (1-zu-2-Abhängigkeit) existieren zwei parallele Pfade, die jeweils eine Transformation und eine Propagation enthalten.

Theoretisch gesehen unterscheiden sich die beiden Varianten nur durch die Anzahl der Lesezugriffe. Bei dem N 1-zu-1-Fall sind es N Zugriffe während es bei dem 1-zu-N-Fall nur einer ist. Da die zuverlässige Multicast-Warteschlange, so wie sie in Abschnitt 3.7.2.3 eingeführt wurde, nicht vorhanden ist und diese simuliert werden muss, kommen noch N Schreibzugriffe für die zuverlässige Multicast-Warteschlange hinzu. Um die beiden Varianten zu vergleichen, wurde der Durchsatz an Änderungen gemessen. Der seither verwendete Prozessdurchsatz kann hier nicht zum Vergleich herangenommen werden, da sich die beiden Ansätze durch die Anzahl von Prozessen pro Änderung unterscheiden. Weiterhin können die Prozesszeiten nicht verwendet werden, da diese sich anhand der zugeordneten Aufgaben stark unterscheiden würden.

Das Ergebnis der Messung (Abbildung 5.17) ist, dass die 1-zu-N-Variante mit einem Propagationsskript eine höhere Performance hat als die N x 1-zu-1-Variante mit N Propagationsskripten. Dieser Vorsprung würde sich noch verringern, wenn – wie oben

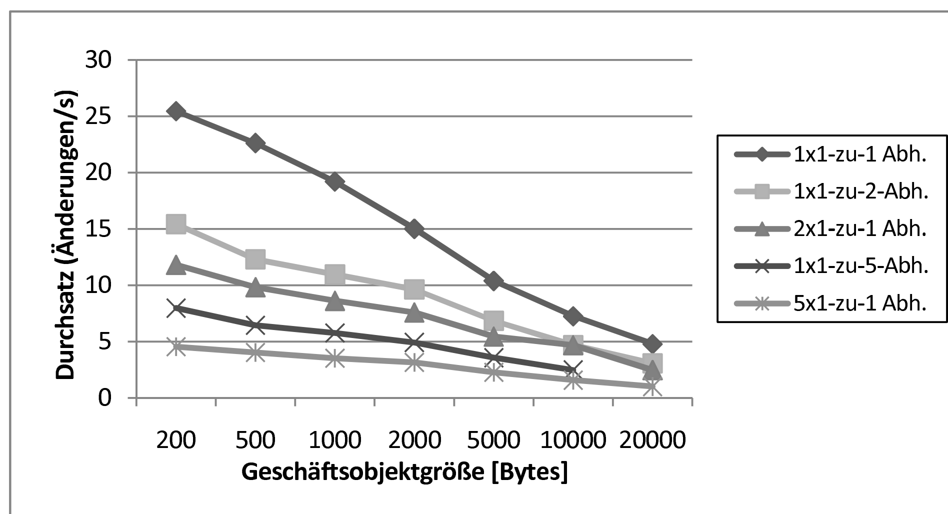


Abbildung 5.17: Einfluss der Abhängigkeitsart (1-zu-N zu N x 1-zu-1) auf den Durchsatz

erwähnt – die zuverlässige Multicast-Warteschlange optimiert würde. Betrachtet man wieder eine mittlere Geschäftsobjektgröße von 1000 Bytes, so ergeben sich folgende Durchsatzwerte (Änderungen pro Sekunde): 19,19 für eine 1-zu-1-Abhängigkeit, 10,94 für eine 1-zu-2-Abhängigkeit, 8,62 für zwei 1-zu-1-Abhängigkeiten, 5,75 für eine 1-zu-5-Abhängigkeit und 3,51 für fünf 1-zu-1-Abhängigkeiten. Die beiden Varianten unterscheiden sich durch eine 27-prozentige Performancesteigerung bei zwei Propagationen und eine 64-prozentige Performancesteigerung bei fünf Propagationen.

Lässt man die Transaktionssemantik außer acht, ist die Variante die Propagationen in einem Propagationsskript zu kombinieren der N-Propagationsskriptvariante vorzuziehen.

5.2.4.7 Testfall 7: Anzahl der wartenden M-zu-N-Prozesse

Ziel dieses Tests war die Messung des Einflusses der wartenden M-zu-N-Prozesse (vgl. Abschnitt 4.2) auf die Performance (Durchsatz). Wenn mehrere M-zu-N-Prozesse auf Änderungsbeschreibungen warten, müssen Änderungsbeschreibungen für die Zuordnung zu M-zu-N-Prozessen gefiltert werden, was Rechenzeit kostet. Für das Filtern müssen die Zustände geparst und die PCL-Bedingungen (siehe Abschnitt 3.6.1 und 4.2.4) evaluiert werden.

Die Messungen wurden so durchgeführt, dass jede ankommende Änderungsbeschreibung von jedem M-zu-N-Prozess anhand von System und Geschäftsobjekttyp erwartet wurde, aber dennoch die Filterbedingungen nicht erfüllt waren. Es wurden Messungen durchgeführt mit folgender Anzahl von wartenden M-zu-N-Prozessen: 0, 25, 50, 75, 100. Die recht niedrigen Zahlen haben den Hintergrund, dass dieses Konzept nicht so häufig für die Propagation von Änderungen gebraucht wird. Die M-zu-N-Prozesse werden am Anfang gestartet, was durch die Versendung von neuen Kunden passiert. Der M-zu-N-Prozess wartet schließlich auf das Eintreffen eines Kundenauftrags. Es werden

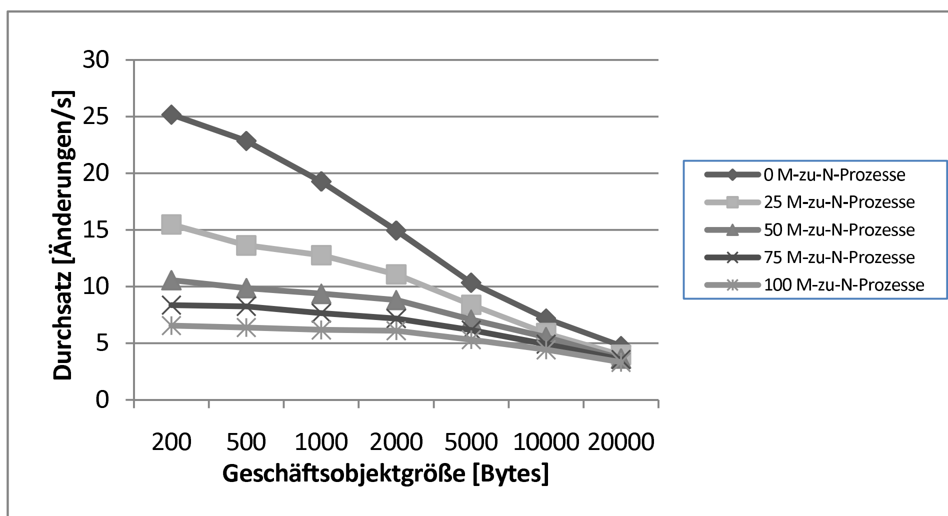


Abbildung 5.18: Einfluss der wartenden M-zu-N-Prozesse auf den Durchsatz

beim anschließenden Test nur Kundenaufträge verschickt, d.h. jede Änderung ist eine potenzielle Änderung für die wartenden M-zu-N-Prozesse, die Filterbedingung tritt aber nie ein, so dass der Prozess nie gestartet wird und so dem System erhalten bleibt und die Anzahl wartender M-zu-N-Prozesse konstant bleibt.

Schaut man sich die Ergebnisse des Tests in Abbildung 5.18 (Durchsatzverhalten) an, so stellt man fest, dass der größte Einbruch zwischen 0 und 25 wartenden Prozessen liegt. Dies ist durch das benötigte Parsen begründet, das einen relativ hohen Zeitbedarf gegenüber dem relativ kleinen bei der Bedingungsevaluation hat. Weiterhin konnte festgestellt werden, dass der Einfluss mit steigender Geschäftsobjektgröße abnimmt.

5.2.4.8 Testfall 8: Integration externer Daten

Bei diesem Testfall sollte ermittelt werden, wie sich die in Abschnitt 4.1 eingeführte Integration von Daten aus Drittsystemen auf die Performance des Gesamtsystems auswirkt. Es wurden ebenfalls die beiden Implementierungsvarianten, schichtenbasierter Transformationsansatz [HCM05] und Parameterbindung zur Laufzeit (vgl. Abschnitt 4.1.6.1), verglichen.

Gemessen wurde mit drei unterschiedlichen Propagationsskripten: keine Einbindung von externen Daten, 1 und 2 externe Datenanbindungen. Um die Messungen durchzuführen, wurde wieder die Propagation von Kundenaufträgen verwendet. Das externe System liefert den aktuellen Kreditstand (Datenanbindung 1) und die Kreditobergrenze (Datenanbindung 2) des Kunden. Diese sollen dann in den Kundenauftrag eingetragen und an das Zielsystem verschickt werden. Es gibt folgende Transformationen in den Propagationsskripten: Zugriffsberechtigung, Anfragen erzeugen und Integration der Ergebnisse mit der Änderungsbeschreibung.

Das externe System muss zuerst die Anfrage entpacken und dann die Zugriffsberechtigung überprüfen. Die Tabelle, welche die Zugriffsberechtigungen enthält, besteht aus 100 Einträgen. Ist diese Überprüfung erfolgreich, so wird die enthaltene SQL-Anfrage

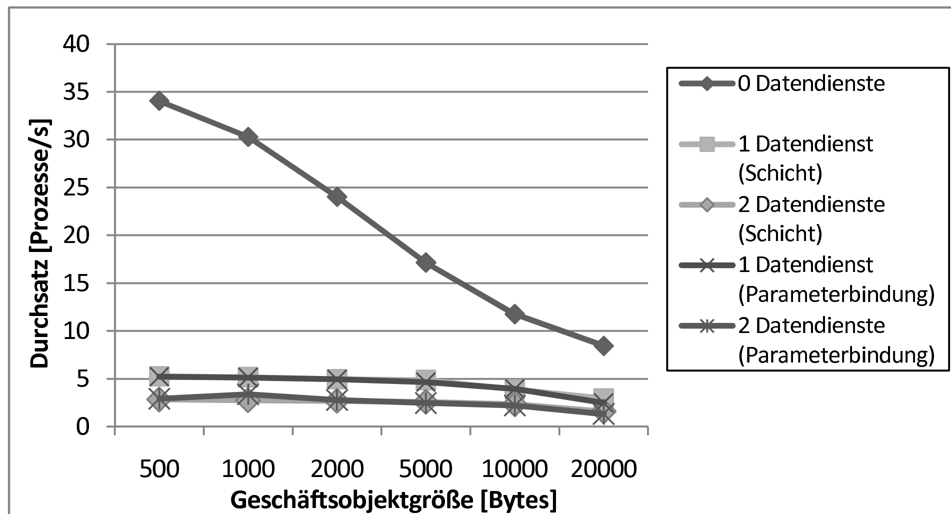


Abbildung 5.19: Einfluss der Integration von externen Daten auf den Durchsatz

ausgeführt und das Ergebnis in XML verpackt. Dabei wird auf eine Kundentabelle zugegriffen, die 1000 Einträge hat. Das Ergebnis wird schließlich an das Propagationssystem zurückgesendet.

Außerdem wurden in den Messungen die zwei vorgestellten Arten, um Datendienste aufzurufen verglichen (vgl. Abschnitt 4.1.6.1): Transformationsansatz mit Schichten (Schicht) und Paramaterbindungsansatz zur Laufzeit (Parameterbindung).

Die Messungen ergaben, dass der Durchsatz (Abbildung 5.19) bei niedriger Geschäftsobjektgröße stark einbricht. Allerdings wird davon ausgegangen, dass durch die hohen Wartezeiten der Durchsatz verbessert werden kann, indem die maximale Anzahl von Prozessen bei dem Zugriff auf externe Daten kurzfristig hoch gesetzt wird. Dies muss allerdings noch durch eine entsprechende Implementation und Messungen evaluiert werden.

Die Prozesszeiten (Abbildung 5.20) sind bei dem Zugriff auf externe Daten deutlich länger, verhalten sich aber weitgehend konstant im unteren Bereich. Dies ergibt sich durch die ausschlaggebenden Kommunikationszeiten mit dem Drittsystem. Da zu diesen Zeiten das Propagationssystem auch nicht ausgelastet ist, könnten in diesen Fällen neue Prozesse gestartet werden, die über die Prozessbegrenzung hinausgehen, was dann zu einem höheren Durchsatz führen würde.

5.2.4.9 Testfall 9: Einhaltung der Änderungsreihenfolge

Im Testfall 9 wurde der Einfluss der Reihenfolgeeinhaltung (siehe Abschnitt 3.9) auf die Performance ermittelt. Es wurde die FIFO-Propagationsordnung verwendet, wobei der Test so ausgelegt war, dass alle Änderungsbeschreibungen geordnet werden müssen. Die Messungen ergaben für die Reihenfolgeeinhaltung – bis auf feststellbare Wartezeiten (längere Prozesszeiten) – keinen Einfluss auf den Durchsatz. Dies ist dadurch begründet, dass ein wartender Prozess keine Belastung für den Prozessor darstellt und dadurch die anderen Prozesse schneller abgearbeitet werden können.

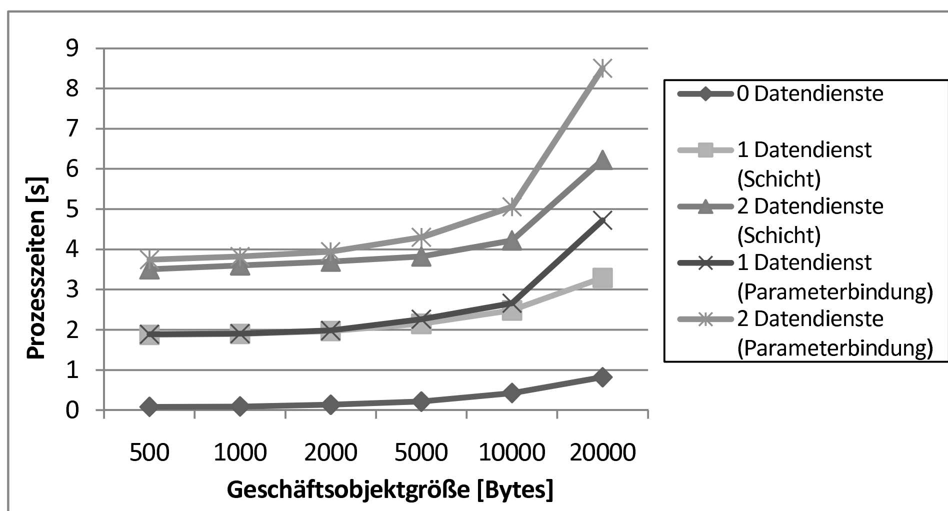


Abbildung 5.20: Einfluss der Integration von externer Daten auf die Prozesszeiten

5.2.5 Zusammenfassung

In diesem Abschnitt wurden mehrere Testfälle untersucht. Bei einem Update wird die Änderungsbeschreibung durch zwei Zustände größer und damit auch die Performance kleiner. In Testfall 1 wurde der Einfluss der Update-Häufigkeit auf die Performance untersucht. Dabei stellte sich heraus, dass der Einfluss auf die Prozesszeit mit höher werdenden Geschäftsobjektgrößen stärker wird und beim Durchsatz abnimmt. Bei Testfall 2 (Prozessbegrenzungen) wurde festgestellt, dass ein Deadlock auftritt, wenn die Prozessbegrenzung kleiner ist als die Anzahl der gestarteten Prozesse pro Änderung. Um dies zu verhindern, musste der Ablauf des Prozessmanagers angepasst werden, so dass die Propagationsprozesse erst nach der Transaktion gestartet werden. Während die Anzahl der Transformationen pro Propagationsskript einen merklichen Einfluss besitzt, hat die Anzahl der Bedingungen keinen ausschlaggebenden Einfluss. Der Einfluss der Cache-Hitrate nimmt mit größeren Geschäftsobjekten ab und wirkt sich erst bei großen Hitraten aus. Außerdem wurde festgestellt, dass 1 Propagationsskript (1-zu-N-Abhängigkeit) N Propagationsskripten (1-zu-1-Abhängigkeit) vorzuziehen ist, wenn man die Transaktionssemantik außer acht lässt.

5.2.6 Vergleich mit Anforderungen aus der Industrie

Mit den Messungen in diesem Kapitel sollte ein erster Eindruck von der Leistungsfähigkeit des Propagationssystems ermittelt werden. Diese Werte sollen nun mit gängigen Werten aus der Industrie verglichen werden. Nimmt man zum Beispiel die Verkaufszahlen von Mercedes-Benz, Smart und Maybach zusammen (573.900 verkaufte Autos im Jahr 2005 [Süd05]), so ergibt das ungefähr 573.900 Kundenaufträge. Rechnet man dies auf eine Sekunde herunter, ergeben sich ungefähr 0,03 Kundenaufträge pro Sekunde (bei

einer 5-Tage-Woche und Dreischichtbetrieb)[2]. Schaut man sich die Testergebnisse an, so ist dies ein Durchsatz, der selbst durch Einbindung von zwei Drittsystemen erreicht werden kann. Schwer abzuschätzen ist, wie viele Produktionsaufträge aus den Kundenaufträgen entstehen und welche Belastung hieraus entsteht. Ein Produktionsauftrag ist ein interner Auftrag zur Herstellung eines Produktes. Bei der Beziehung zwischen Produktionsaufträgen und Kundenaufträgen handelt es sich um eine M-zu-N-Beziehung. Weitaus höhere Zahlen werden Daten aus MDE- und BDE-Systemen (Maschinendatenerfassung und Betriebsdatenerfassung) ergeben. Bei der Maschinendatenerfassung werden Daten von Produktionsressourcen und bei der Betriebsdatenerfassung Rückmeldungen zu aktuell abgearbeiteten Produktionsaufträgen erfasst. Die anfallenden Daten bei der BDE hängen hauptsächlich von der Anzahl von Meldepunkten und Produktionsaufträgen ab. Diese Zahl lässt sich wiederum schwer abschätzen, da keine genauen Zahlen zu Produktionsaufträgen vorliegen. Deswegen wird eine grobe Abschätzung durchgeführt. Gehen wir aber von 30 Produktionsaufträgen pro Kundenauftrag und 10 Meldepunkten aus, so ergeben sich 9 Meldungen pro Sekunde. Untersucht man daraufhin die Messungen von z.B. Abbildung 5.5, so stellt man fest, dass dies bis zu einer Geschäftsobjektgröße von 5000 Bytes erreicht wird. Diese Größe sollte für die Verarbeitung von Meldungen ausreichen. Ermittelt man nun die Summe aus Kundenaufträgen, Produktionsaufträgen und Auftragsmeldungen, so ergibt sich eine leicht höhere Zahl von 9,93 Propagationen pro Sekunde. Dies wird auch von dem erwähnten Testfall bis ungefähr 5000 Bytes erreicht. Nimmt man jetzt allerdings noch an, dass 5 Zielsysteme (vgl. Abbildung 5.17) pro Propagation geändert werden sollen, so reicht die Leistung des Propagationsmanagers nicht aus und es muss an eine Verteilung (Abschnitt 4.3) oder schnellerer Hardware gedacht werden. Ebenfalls kritisch ist die Einbindung von Drittsystemen, da dort der Durchsatzeinbruch durch Wartezeiten begründet ist. Der Durchsatz müsste aber durch eine intelligente Wahl von Prozessbegrenzungen erhöht werden können.

5.3 Vergleich mit EAI-Produkten

Es existiert auf dem Markt eine Vielzahl von EAI-Produkten (*Enterprise Application Integration*, vgl. Abschnitt 2.3) bzw. Enterprise-Service-Bus-Produkten (ESB-Produkte), die die Integration von Informationssystemen ermöglichen. Diese können für die Entwicklung von Integrationsanwendungen verwendet werden, die meist anhand von Prozessen definiert werden, d.h. die Integrationsebene ist vor allem die Prozessebene. Beim Propagationssystem werden allerdings Informationssysteme auf der Datenebene durch die Weiterleitung von Datenänderungen integriert. Das bedeutet aber nicht, dass diese Produkte nicht ebenfalls für die Integration von Daten eingesetzt werden können, was von den Herstellern auch teilweise vorgeschlagen wird. Bei den Produkten handelt es sich bei den ausgetauschten Daten um Daten ohne bestimmte Semantik, d.h. es muss sich nicht unbedingt um eine Datenänderung handeln. Im Ver-

[2]Anmerkung: Kundenaufträge werden nicht im Dreischichtbetrieb angenommen und eingepflegt. Allerdings trifft es auf die Produktionsdaten zu, die später eingeführt werden und die ein größeres Aufkommen haben als die Kundenaufträge

gleich dazu existiert beim Propagationssystem die Semantik, dass ausgetauschte Objekte Änderungsbeschreibungen von Geschäftsobjekten sind. Die Nachrichten der EAI- und ESB-Produkte sind außerdem für nur einen Objektzustand gedacht. Durch eine entsprechende Definition der Nachrichten kann der Transport von zwei Zuständen simuliert werden. Allerdings müssen die Transformationsskripte und die Korrektheitsüberprüfung entsprechend definiert werden, d.h. sie müssen die Hilfsstruktur kennen und haben dadurch keine Transparenz in der Verarbeitung. Die Hilfsstruktur mit integrierten Geschäftsobjektzuständen muss schließlich für alle Integrationsanwendungen bzw. Propagationsskripte definiert werden, was eine große Fehlerquelle darstellt.

Die betrachteten Produkte sind der BizTalk-Server von Microsoft, die SOA-Suite von Oracle und der Websphere Message Broker von IBM. Dies ist nur eine Auswahl von EAI-Produkten. Die meisten Produkte sind auf einer abstrakten Ebene sehr ähnlich. Sie unterscheiden sich in ihren Produkteigenschaften, arbeiten aber auf den gleichen Prinzipien. Das Produkt, welches von den dreien am meisten abweicht, ist der Websphere Message Broker, da dieser nicht einen reinen Geschäftsprozessansatz hat.

5.3.1 BizTalk

Der BizTalk Server [WML+05, AHH+02, Mic08] ist eine Lösung zur Implementierung von Geschäftsprozessen. Im Vordergrund steht dabei die Orchestration von Geschäftsprozessen, d.h. das Zusammenspiel mehrerer Geschäftsprozesse. Ein Beispiel hierfür ist der Geschäftsprozess für eine Bestellung, der direkt mit dem Prozess des Lieferanten zusammenarbeitet, der die Kundenaufträge bearbeitet. Die Sprache zur Definition von Geschäftsprozessen ist XLang. Die Geschäftsprozesse können in BPEL [ACD+03, Oas07] exportiert oder importiert werden. Allerdings kommt der Entwickler mit der darunter liegenden Sprache nicht in Kontakt, da er ein graphisches Werkzeug (Orchestration Designer) verwendet. Die Geschäftsprozesse basieren auf dem Austausch von Nachrichten, die mit verschiedenen Technologien kommuniziert werden können (z.B. Warteschlangen). Nachrichten können mittels Transformationsskripten angepasst werden, die über den BizTalk-Mapper erzeugt werden. Der BizTalk-Mapper ist ein graphisches Werkzeug zur Definition von Transformationen, vergleichbar mit Altova MapForce. Die Mapping-Definitionen (eigenes Mapping-Format) werden schließlich in XSLT überführt.

Grundsätzlich kann der BizTalk-Server auch für den Verwendungszweck EAI eingesetzt werden. In diesem Fall werden mit dem Orchestration Designer keine Geschäftsprozesse im eigentlichen Sinn erzeugt, sondern EAI-Anwendungen. Gerade die Vielzahl von bereitgestellten und verfügbaren Adaptern und Transportmedien ermöglicht den Einsatz als EAI-Werkzeug.

Vergleicht man jetzt BizTalk-Server mit dem Propagationssystem, so fällt als Erstes das Anwendungsgebiet auf. Der BizTalk-Server dient zur Integration von Geschäftsprozessen sowie EAI. Das Propagationssystem dient zur Datenintegration auf Basis von Änderungspropagationen. Prinzipiell lassen sich solche Änderungsbeschreibungen auch mit dem BizTalk-Server austauschen, allerdings ist der BizTalk-Server nicht für diesen Anwendungsfall spezialisiert. Dies zeigt sich zuerst einmal in der Unterstützung von

zwei Zuständen im Propagationssystem. Dies muss direkt im Nachrichtenformat codiert werden und der BizTalk-Server bietet keine Transparenz für dessen Behandlung, d.h. der Mapping-Ersteller muss Wissen sowohl über die Hilfsstruktur als auch über die Struktur des Zustandes haben. Das Gleiche gilt für die XML Schemas, die ebenfalls auf der Ebene der Hilfsstruktur definiert werden müssen. Des Weiteren fehlt in BizTalk die Anwendungstransparenz, d.h. das Quellsystem müsste direkt mit dem jeweiligen Propagationsskript kommunizieren bzw. ein Integrationsentwickler könnte dies mittels der spezifischen Konfiguration von Channels simulieren, was jedoch im Propagationssystem automatisch funktioniert. Außerdem werden in BizTalk keine Konflikte behandelt, was aber durch den hier vorgestellten Ansatz (vgl. Abschnitt 3.8) leicht realisierbar wäre, da dieser Ansatz auf zwei Zuständen basiert und im Adapter gehandhabt wird. Ein weiterer Vorteil des Propagationssystems ist die Möglichkeit einer recht einfachen Darstellung der Propagationsskripte durch die Abhängigkeitsansicht (vgl. Abschnitt 3.7.3.3).

5.3.2 Oracle SOA Suite

Die Oracle SOA Suite [Ora06, Ora07, Ora08] ist ein Produkt, mit dem ebenfalls Geschäftsprozesse realisiert werden können. Im Vordergrund steht die Realisierung einer Service-oriented Architecture (SOA). Außerdem können Web Services über Orchestrationen realisiert werden. Die Grundlage für die Orchestration bildet die Sprache BPEL [ACD+03, Oas07]. Dadurch wird die Verwandschaft zu BizTalk klar. Besonders erwähnenswerte Technologien sind die *Domain Value Maps* und *Cross Reference Tables*. Mit Domain Value Maps können Werte, wie beispielsweise Abkürzungen auf ausgeschriebene Worte, abgebildet werden. Cross Reference Tables dienen zum Mapping der IDs aus unterschiedlichen Systemen. Diese beiden Technologien sind in dem Propagationssystem nicht realisiert, sondern müssen durch Drittsysteme (vgl. Abschnitt 4.1) eingebunden werden. Dies kann aber in zukünftigen Erweiterungen realisiert werden.

Der größte Unterschied stellt, wie bei BizTalk, das Anwendungsgebiet dar. Die Fokussierung liegt hier auf der Orchestration von Web Services, statt auf der Integration von Informationssysteme auf der Datenebene. Um Oracle als Datenintegrationssystem einzusetzen, müsste die Anwendungstransparenz aufgegeben werden bzw. durch zusätzliche Definition von *Routing Rules* im Oracle Mediator simuliert werden. Des Weiteren müsste die Hilfsstruktur zusammen mit den beiden Zuständen kodiert werden und XML Schemas und Transformationsskripte könnten nicht auf der Ebene der Zustände definiert werden. Ebenfalls könnte der hier entwickelte Konflikterkennungsmechanismus zum Einsatz kommen, da auch hier Adapter eingesetzt werden, die die Konflikterkennung realisieren können.

5.3.3 Websphere Message Broker

Der Websphere Message Broker [IBM08b, IBM08a, IBM08c] ist ein Enterprise Service Bus (ESB) von IBM, wobei es noch zwei weitere ESB-Produkte von IBM gibt. Dieses Produkt wurde zum Vergleich ausgewählt, da es Websphere MQ verwendet, sowie das

Propagationssystem.

Das Prinzip von Websphere Message Broker ist, dass man Nachrichtenflüsse (Message flows) definiert. Diese entsprechen dem Datenfluss eines Workflows. Diese Nachrichtenflüsse basieren auf einer IBM-spezifischen Sprache. Es existieren Quellen und Senken, Transformationen mit XSLT oder sogenannte Nachrichten-Maps. Die Nachrichten-Maps können graphisch erstellt werden. Zusätzlich gibt es die Sprache ESQL (Extended SQL). Sie ist eine Sprache zur Datendefiniton und Manipulation von Daten innerhalb eines Nachrichtenflusses. Diese Sprache kann zum Beispiel in Berechnungsknoten und Filterknoten eingesetzt werden. Des Weiteren können Funktionen in ESQL erzeugt werden, die wiederum in Mappings eingesetzt werden können.

Wie bei den anderen Produkten steht hier im Vordergrund der Austausch von Nachrichten, die noch keine spezifische Semantik haben. Dies hat zur Folge, dass die Zustände in einer Nachricht codiert werden müssen, was Auswirkungen auf die Transparenz der Schema-Definition und die Transformation hat. Des Weiteren existiert keine Integrationsanwendungstransparenz, wie beim Propagationssystem. Das bedeutet, dass ein Adapter die Integrationsanwendungen kennen muss, während bei dem hier vorgestellten Ansatz nur das Propagationssystem bekannt sein muss. Das Propagationssystem verteilt dann selbständig die Änderungsbeschreibungen auf die einzelnen Prozesse. Der in dieser Arbeit entwickelte Mechanismus zur Konflikterkennung könnte auch hier zum Einsatz kommen, sofern zwei Zustände verwendet werden. Die Dokumentation macht keine Aussage über eventuelle Algorithmen zur Einhaltung der Reihenfolge.

5.3.4 Schlussfolgerung

Die hier vorgestellten Produkte lassen sich nur bedingt vergleichen, da hier oftmals die Integration von Prozessen (Orchestration) im Vordergrund steht und nicht wie im Propagationssystem die Integration von Daten. Die Kodierung von zwei Zuständen in den Nachrichten ergibt allerdings Probleme, da dadurch Transparenz aufgegeben werden muss. Noch problematischer ist die Aufgabe der Integrationsanwendungstransparenz, d.h. Informationssysteme müssen die Integrationsanwendung kennen bzw. den Propagationsprozess. Für die von den Produkten angestrebte Anwendung ist dies aber nicht von Nachteil, sondern stellt sich eher als Vorteil heraus. Für ein reines Änderungspropagationssystem ist dies aber ein Nachteil, da in diesem Fall die entsprechenden Propagationsprozesse den angebundenen Informationssystemen unbekannt bleiben und das Propagationssystem die Verteilung intern regelt. Die Konfliktbehandlung ist von allen drei untersuchten Systemen nicht vorgesehen. Über die Reihenfolgeeinhaltung wird in den Systembeschreibungen keine Aussage gemacht. Sie kann deswegen nicht genauer untersucht werden. Bezüglich der vergleichbaren Punkte sind diese Systeme also nur bedingt als Propagationssystem einsetzbar, da wichtige Eigenschaften fehlen, wie die Transparenz der Integrationsanwendung, die Anzahl von Änderungszuständen, die Einhaltung der Reihenfolge sowie die Konflikterkennung und -auflösung.

KAPITEL 6

Schlussfolgerung und Ausblick

In diesem Kapitel werden die wichtigsten Punkte zusammengefasst und Schlussfolgerungen aus der Arbeit gezogen. Des Weiteren wird ein Ausblick auf zukünftige Forschungsarbeiten gegeben, die im Zusammenhang mit der Änderungspropagation für heterogene Systeme durchgeführt werden könnten.

6.1 Schlussfolgerungen

In dieser Arbeit wurde ein Integrationssystem entwickelt, das Geschäftsobjektänderungen propagiert. Dadurch soll erreicht werden, dass die Daten zwischen den einzelnen Informationssystemen konsistent bleiben. Konsistenz ist dabei so definiert: Befindet sich das Gesamtsystem in einem inkonsistenten Zustand und es erfolgen keinerlei externe Änderungen mehr, die weitere Inkonsistenz verursachen, wird das Gesamtsystem nach einer gewissen Zeit wieder in einem konsistenten Zustand sein, sofern alle benötigten Abhängigkeiten definiert wurden. Die Zeitdauer bis das Gesamtsystem wieder konsistent ist, hängt von folgenden Faktoren ab: Die Dauer bis eine Änderung in einem Quellsystem erkannt wird, die Warteschlangenlängen vor dem Propagationssystem sowie den Zielsystemen, Verarbeitungsdauer im Propagationssystem und die Zeitdauer zum Einspielen der Änderung im Zielsystem. Die Technologie *Update Propagation* von replizierten Datenbanken wurde genauer untersucht (Abschnitt 3.1), da bei dieser Technologie viele Varianten zur Propagation von Änderungen erforscht wurden und deshalb gewisse Verfahren auch für ein solches Propagationssystem geeignet sind (z.B. Lazy Replikation). Die Integration mittels des Änderungspropagationssystems befindet sich also auf der Datenebene, d.h. es werden Daten miteinander integriert. Ziel der Arbeit war es die Autonomie der integrierten Informationssysteme soweit wie möglich beizubehalten und eine hohe Heterogenität zwischen den Systemen zuzulassen. Die Autonomie wurde durch eine starke Entkopplung der Informationssysteme erreicht. Die Informationssysteme benötigen keine Kenntnis von anderen Informationssystemen

mit denen sie integriert sind. Des Weiteren kann durch den Einsatz von Adaptoren ihre Unabhängigkeit vom Änderungspropagationssystem erreicht werden. Die Verwendung von persistenten und asynchronen Warteschlangen ermöglicht weiterhin eine Entkopplung der Informationssysteme vom Propagationssystem. Die Heterogenität wird durch Transformation der Änderungen und die Einbindung von Daten aus Drittsystemen über sogenannte Datendienste (Abschnitt 4.1) überwunden. Weitere wichtige Eigenschaften eines solchen Propagationssystem sind eine hohe Zuverlässigkeit, die Einhaltung der Reihenfolge und die Erkennung und Auflösung von Änderungskonflikten.

Die wichtigsten Beiträge der Arbeit sind im Folgenden aufgelistet:

- Die Entwicklung eines Propagationssystems für autonome und heterogene Informationssysteme auf Basis der Technologie *Update Propagation* (Abschnitt 3.1).
- Die Definition von Änderungsbeschreibungen (Abschnitt 3.2.2), die der Propagation von Geschäftsobjektänderungen zwischen heterogenen und autonomen Systemen gerecht werden.
- Die Propagation von zwei Zuständen (vor und nach der Änderung, 3.2.2). Dies ermöglicht die Berechnung von Änderungsdeltas innerhalb des Propagationssystems und das Erkennen der Änderungsarten von Implementationsobjekten. Geschäftsobjekte können aus mehreren Implementationsobjekten bestehen, die abweichende Änderungsarten haben. Zum Beispiel können bei der Änderung einer Bestellung Bestellposten hinzukommen oder gelöscht werden.
- Das Quellsystem muss keine Kenntnis haben, in welchen Integrationsanwendungen es teilnimmt oder welche Propagationsskripte für seine Änderungen existieren: Integrationsanwendungstransparenz
- Die Entwicklung der Sprache *XML Propagation Definition Language* (XPDL, Abschnitt 3.5) zur Definition von Propagationsskripten.
- Die Sprache *Propagation Condition Language* (PCL, Abschnitt 3.6.1) auf Basis von XPath für die Definition von Bedingungen zwischen Zuständen.
- Der Entwickler der Propagationsskripte, Transformationsskripte und Schemas muss kein Wissen über die Anzahl der Änderungszustände der verarbeiteten Änderungsbeschreibung haben. Transformationsskripte und Schemas werden für einen Zustand entwickelt. Dies wird hier Zustandstransparenz bezeichnet.
- Eine Konflikterkennung für Änderungskonflikte auf Basis von Zuständen (Abschnitt 3.8)
- Eine Abhängigkeitsansicht (Abschnitt 3.7.3.3) zur einfachen Erstellung von Propagationsskripten, die die Abhängigkeiten der einzelnen Geschäftsobjekte/Systeme darstellt.
- Die Definition und Einbindung von Datendiensten für das Propagationssystem (Abschnitt 4.1).

Bei einem Einsatz des entwickelten Propagationssystems innerhalb des Sonderforschungsbereichs, bei dem drei Systeme integriert wurden, hat sich die Praxistauglichkeit des Ansatzes herausgestellt (Abschnitt 5.1). Die durchgeführte Evaluation der Performanz (Abschnitt 5.2) sind zu einem positiven Ergebnis gekommen, die sich mit gängigen Anforderungen der Industrie (Abschnitt 5.2.6) messen lassen können. Allerdings muss angemerkt werden, dass die Erkennung von Änderungen in manchen Informationssystemen schwierig sein könnte, so dass häufig Eingriffe in das System nötig werden, wie z.B. Skripte anpassen oder neu entwickeln. Allerdings ist das langfristige Ziel, dass Informationssysteme einen Propagationsstandard unterstützen und damit die Erkennung von Änderungen direkt im System gehandhabt wird, ohne dass die Informationssysteme angepasst werden müssen und die erkannten Änderungen dann an ein Propagationssystem weitergeleitet werden.

6.2 Ausblick

Der Propagationsmanager kann erweitert werden, um die Technologien *ID Mapping* (*Cross Reference Tables*) und *Domain Value Mapping* (Abschnitt 5.3.2) zu unterstützen. Bei Ersterem werden die IDs der unterschiedlichen Systeme aufeinander abgebildet. Das Domain Value Mapping ermöglicht die Transformation von Werten. Zum Beispiel kann damit „Bundesrepublik Deutschland“ auf „BRD“ abgebildet werden und umgekehrt. Das Mapping von Werten wird bisher im Propagationssystem durch die Einbindung von Drittsystemen realisiert. Allerdings könnte durch direkte Speicherung der Tabellen im Repository, die Performanz erhöht werden.

Die zustandsbasierte Konflikterkennung muss in einem solchen Propagationssystem noch genauer erforscht werden. Außerdem könnte die Konfliktauflösung durch so genannte Merge-Funktionen teilweise automatisiert werden [TTP+95].

Gerade die Erstellung von Transformationen lässt sich durch die Einbindung von *Automatic Schema Matching* (Abschnitt 2.10.2) deutlich beschleunigen und vereinfachen. Des Weiteren ist die Untersuchung von Schema-Evolution und deren Auswirkungen auf Transformations- und Propagationsskripten von großem Interesse, wobei das von Boris Stumm vorgeschlagene System [Stu10] zum Einsatz kommen könnte.

Die Begrenzung der maximal gleichzeitig laufenden Propagationsprozesse kann anhand der wartenden Propagationsprozesse (Einbindung von Drittsystemen) intelligent angepasst werden. Dies führt wahrscheinlich zu einer deutlichen Steigerung des Durchsatzes bei der Einbindung von Drittsystemen.

Die Umsetzung des gesamten Systems mit SOA-Technologien lässt eine weitere Verbesserung hinsichtlich Änderbarkeit, Erweiterbarkeit und Einsatzmöglichkeiten des gesamten Propagationssystems erwarten. Erste Ansätze dazu wurden in [MJHM09] genauer betrachtet und zusammengestellt.

Literaturverzeichnis

[ABF02] Paulo Sergio Almeida, Carlos Baquero, und Victor Fonte. Version Stamps – Decentralized Version Vectors. In *22nd International Conference on Distributed Computing Systems (ICDS)*, Seiten 544–551, Wien, Österreich, 2002.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, und Sanjiva Weerawarana. Business Process Execution Language for Web Services – Version 1.1. Specification, Microsoft, IBM, Siebel Systems, BEA, SAP, 2003. Verfügbar bei: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf [30.06.2005].

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, und Vilay Machiraju. *Web Services – Concepts, Architecture and Applications.* Springer, 2004.

[ACM00] Paolo Atzeni, Luca Cabibbo, und Giansalvatore Mecca. Database Cooperation: Classification and Middleware Tools. *Journal of Database Management*, 11(1), 2000.

[AFHS95] Oksana Arnold, Wolfgang Faisst, Martina Härtling, und Pascal Sieber. Virtuelle Unternehmen als Unternehmenstyp der Zukunft? *HMD - Praxis der Wirtschaftsinformatik*, (185), September 1995.

[AHH+02] Susie Adams, Dilip Hardas, Akhtar Hossein, Clifford R. Cannon, Rand Morimoto, Kevin Price, Stephan Tranchida, Bill Martschenko, Rick Pearson, Tom Lake, Rob Oikawa, Cuneyt Havlioglu, Charlie Kaiman, und Larry Wall. *BizTalk – Unleashed.* Sams Publishing, 2002.

[AL80] Michel E. Adiba und Bruce G. Lindsay. Database Snapshots. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, Seiten 86–91. IEEE Computer Society, 1980.

[AT89] A. El Abbadi und S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[ATS+05] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, und Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, Seiten 565–576. VLDB Endowment, 2005.

[BA99] Carlos Baquero und Paulo Sérgio Almeida. Towards efficient time-stamping for autonomous versioning. In *Actas informais do EPCM'99, Encontro Português de Computação Nómada*, 1999.

[BBC+07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, und Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Recommendation, World Wide Web Consortium, 2007.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, und Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, World Wide Web Consortium, 2007.

[BD94] Philip A. Bernstein und Umeshwar Dayal. An Overview of Repository Technology. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, Seiten 705–713, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[Ber98] Philip A. Bernstein. Repositories and object oriented databases. *SIGMOD Record*, 27(1):88–96, 1998.

[BF97] Hans-Jörg Bullinger und Klaus-Peter Fähnrich. *Betriebliche Informationssysteme – Grundlagen und Werkzeuge der methodischen Softwareentwicklung.* Springer Verlag, 1997.

[BG82] Philip A. Bernstein und Nathan Goodman. Concurrency control algorithms for multiversion database systems. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Seiten 209–215, New York, NY, USA, 1982. ACM Press.

[BG83] Philip A. Bernstein und Nathan Goodman. The failure and recovery problem for replicated databases. In *PODC '83: Proceedings of the second annual ACM Symposium on Principles of Distributed Computing*, Seiten 114–122, New York, NY, USA, 1983. ACM Press.

[BGK+02] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, und I. Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Fifth International Workshop on the Web and Databases (WebDB 2002)*, 2002.

[BHP00a] Phillip A. Bernstein, Alon Y. Halevy, und Rachel A. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[BHP00b] Phillip A. Bernstein, Alon Y. Halevy, und Rachel A. Pottinger. A vision for management of complex models. Technical Report MSR-TR-2000-53, Microsoft Research, 2000.

[BK97] Yuri Breitbart und Henry F. Korth. Replication and consistency: being lazy helps sometimes. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seiten 173–184, New York, NY, USA, 1997. ACM Press.

[BKR+99] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, und Avi Silberschatz. Update propagation protocols for replicated databases. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Seiten 97–108, New York, NY, USA, 1999. ACM Press.

[BM04] Paul V. Biron und Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, World Wide Web Consortium, 2004.

[BMN02] Geert Jan Bex, Sebastian Maneth, und Frank Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[BMPQ04] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, und Christoph Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38–43, 2004.

[BMW01] Joseph A. Brady, Ellen F. Monk, und Bret J. Wagner. *Concepts in Enterprise Resource Planning*. Course Technology - Thomson Learning, 2001.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, und John Cowan. Extensible Markup Language (XML) 1.1

(Second Edition). W3C Recommendation, World Wide Web Consortium, 2006.

[BRJ99] Grady Booch, James Rumbaugh, und Ivar Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, 1999.

[Bro04] Wayne Brown. Enterprise resource planning (ERP) implementation planning and structure: a recipe for ERP success. In *SIGUCCS '04: Proceedings of the 32nd annual ACM SIGUCCS conference on User services*, Seiten 82–86, New York, NY, USA, 2004. ACM Press.

[BW95] Paulo Barthelmess und Jacques Wainer. WorkFlow systems: a few definitions and a few suggestions. In *COCS '95: Proceedings of Conference on Organizational Computing Systems*, Seiten 138–147, New York, NY, USA, 1995. ACM Press.

[CBMT96] Bernadette Charron-Bost, Friedemann Mattern, und Gerard Tel. Synchronous, Asynchronous, and Causally Ordered Communication. *Distributed Computing*, 9(4):173–191, 1996.

[CDK01] George Coularis, Jean Dollimore, und Tim Kindberg. *Distributed Systems – Concepts and Design*. Addison Wesley, 3. Edition, 2001.

[CHB+05] Carmen Constantinescu, Uwe Heinkel, Jan Le Blond, Stephan Schreiber, Bernhard Mitschang, und Engelbert Westkämper. Flexible Integration of Layout Planning and Adaptive Assembly Systems in Digital Enterprises. In *Proceedings of the 38th CIRP International Seminar on Manufacturing Systems (CIRP ISMS)*, Mai 2005.

[CHM02] Carmen Constantinescu, Uwe Heinkel, und Holger Meinecke. A Data Change Propagation System for Enterprise Application Integration. In Waleed W. Smari, Nordine Melab, und Shu-Ching Chen, Editoren, *The 2nd International Conference on Information Systems and Engineering (ISE 2002)*, Seiten 129–134. San Diego: The Society for Modeling and Simulation International, Juli 2002.

[CHR98] Andrzej Cichocki, Abdelsalam Helal, und Marek Rusinkiewicz. *Workflow and Process Automation – Concepts and Technology*. Kluwer Academic Publishers, 1998.

[CHRM01] Carmen Constantinescu, Uwe Heinkel, Ralf Rantzau, und Bernhard Mitschang. SIES - An Approach for a Federated Information System in Manufacturing. In *Proceedings of the International Symposium on Information Systems and Engineering (ISE); Las Vegas, Nevada, USA, June 2001*, Seiten 269–275. CSREA Press, Juni 2001.

[CHRM02] Carmen Constantinescu, Uwe Heinkel, Ralf Rantzau, und Bernhard Mitschang. A System for Data Change Propagation in Heterogeneous

Information Systems. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS), Volume I, Cuidad Real, Spain, April 2002*, Seiten 73–80. ICEIS Press/Escola Superior de Technologia de Setubal, Portugal, April 2002.

[CHRM03] Carmen Constantinescu, Uwe Heinkel, Ralf Rantzau, und Bernhard Mitschang. *A System For Data Change Propagation In Heterogeneous Information Systems*, Seiten 51–59. Enterprise Information Systems IV. Dordrecht, Netherlands: Kluwer Academic Publishers, Januar 2003. ISBN: 1-4020-1086-9.

[Con97] Stefan Conrad. *Föderierte Datenbanksysteme – Konzepte der Datenintegration*. Springer Verlag, 1997.

[CS99] Peter Checkland und Jim Scholes. *Soft System Methodology in Action*. John Wiley & Sons Ltd., Chichester, 1999.

[Cum02] Fred A. Cummins. *Enterprise Integration – An Architecture for Enterprise Application and System Integration*. OMG Press - John Wiley Computer Publishing, 2002.

[Dat00] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 2000.

[Dau03] Berthold Daum. *Modeling Business Objects with XML Schema*. Morgan Kaufmann Publisher and dpunkt.Verlag, 2003.

[dFRH98] Fernando de Ferreira Rezende und Klaudia Hergula. The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, Seiten 146–157, New York, NY, USA, 1998. Morgan Kaufmann Publishers Inc.

[DR02] Hong Hai Do und Erhard Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *VLDB*, Seiten 610–621, 2002.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, und Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.

[EM02] Andrew Eisenberg und Jim Melton. SQL/XML is making good progress. *ACM SIGMOD Record*, 31(2):101–108, 2002.

[Fay02] Mohamed Fayad. Accomplishing software stability. *Communications of ACM*, 45(1):111–115, 2002.

[FW04] David C. Fallside und Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation, World Wide Web Consortium, 2004.

[GHI+01] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, und D. Suciu. What can databases do for peer-to-peer. June, 2001.

[GHM+03a] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, und Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, World Wide Web Consortium, 2003.

[GHM+03b] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, und Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation, World Wide Web Consortium, 2003.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, und Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Seiten 173–182, New York, NY, USA, 1996. ACM Press.

[Gro04a] Object Management Group. Event Service Specification – Version 1.2. Specification, Object Management Group, 2004.

[Gro04b] Object Management Group. Notification Service Specification – Version 1.1. Specification, Object Management Group, 2004.

[Har01a] Christoph Hartwich. N-Tier Enterprise-Applikation. In Martin Endig und Thomas Herstel, Editoren, *13. GI-Workshop Grundlagen von Datenbanken*, Gommern, Sachsen-Anhalt, Germany, 2001.

[Har01b] Christoph Hartwich. Why It Is So Difficult to Build N-Tiered Enterprise Applications. Technical Report B 01-05., Institute of Computer Science, Freie Universität Berlin, 2001.

[Has00] Wilhelm Hasselbring. Information system integration. *Communications of ACM*, 43(6):32–38, 2000.

[Hau99] Manfred Hauswirth. *Internet-Scale Push Systems for Information Distribution – Architecture, Components, and Communication*. Dissertation, Technisch-Naturwissenschaftliche Fakultät, Technischen Universität Wien, 1999.

[HBS+02a] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, und Kate Stout. Java Message Service. Specification, SUN Microsystems, 2002.

[HBS+02b] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, und Kate Stout. Java Message Service – Version 1.1 April 12, 2002. Spezifikation, Sun Microsystems, Inc., 2002.

[HC94] Michael Hammer und James Champy. *Business Reengineering – Die Radikalkur für das Unternehmen*. Campus Verlag, 1994.

[HCM05] Uwe Heinkel, Carmen Constantinescu, und Bernhard Mitschang. Integrating Data Changes with Data from Data Service Providers. In *Proceedings of the 18th International Conference on Computer Applications in Industry and Engineering (CAINE 2005)*, Seiten 146–151. ICSA, November 2005.

[Hei00] Uwe Heinkel. Informationsmodelle für wandlungsfähige Produktionssysteme. Diplomarbeit, Universität Stuttgart, 2000.

[Her03] Klaudia Hergula. *Daten- und Funktionsintegration durch Föderierte Datenbanksysteme*. Dissertation, Technische Universität Kaiserslautern, 2003.

[HHW+04] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, und Steve Byrne. Document Object Model (DOM) Level 3 Core Specification – Version 1.0. W3C Recommendation, World Wide Web Consortium, 2004.

[HIM+04] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, und Igor Tatarinov. The Piazza Peer Data Management System. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[Hol04] David Hollingsworth. *The Workflow Handbook 2004*, Kapitel The Workflow Reference Model 10 Years on, Seiten 295–312. Future Strategies Inc., 2004.

[IBM08a] IBM. WebSphere Message Broker – ESQL. Developer's Guide, IBM Corp., 2008.

[IBM08b] IBM. WebSphere Message Broker – Introduction. Developer's Guide, IBM Corp., 2008.

[IBM08c] IBM. WebSphere Message Broker – Message Flows. Developer's Guide, IBM Corp., 2008.

[JBS97] Stefan Jablonski, Markus Böhm, und Wolfgang Schulze, Editoren. *Workflow-Management: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologie*. DPunkt Verlag, 1997.

[JGJ97] Ivar Jacobson, Martin Griss, und Patrik Jonsson. *Software Reuse – Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[JLM+05] Stefan Jablonski, Rainer Lay, Christian Meiler, Sascha Müller, und Wolfgang Hümmer. Data logistics as a means of integration in healthcare applications. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, Seiten 236–241, New York, NY, USA, 2005. ACM Press.

[JM90] S. Jajodia und David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[JWP00] P. Johannesson, B. Wrangler, und P.Jayaweera. Application and Process Integration – Concepts, Issues, and Research Directions. In *Information Systems Engineering Symposium–CAiSE 2000*, Chigago, USA, 2000.

[KA00] Bettina Kemme und Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[Kay07] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Recommendation, World Wide Web Consortium, 2007.

[Kel02] Wolfgang Keller. *Enterprise Application Integration – Erfahrung aus der Praxis*. DPunkt Verlag, 2002.

[Kep02] Stephan Kepser. A Proof of the Turing-completeness of XSLT and XQuery. Technischer Bericht, SFB 441, Universität Tübingen, Mai 2002.

[Kep04] Stephan Kepser. A Simple Proof of the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.

[Ker01] Oliver Kersten. Konzeption eines Propagationsmanagers. Diplomarbeit, IPVS – Universität Stuttgart, 2001.

[KKL+04] Matthias Kloppmann, Dieter König, Frank Leymann, Gerhard Pfau, und Dieter Roller. Business process choreography in WebSphere: Combining the Power of BPEL and J2EE. *IBM System Journal*, 43(2):270 – 296, 2004.

[Krü84] Wilfried Krüger. *Organisation der Unternehmung*. Kohlhammer Lehrbuchreihe Betriebswirtschaft, 1984.

[Krc03] Helmut Krcmar. *Informationsmanagement*. Springer Verlag, 3. Edition, 2003.

[Kur02] Alexander Kurth. *Entwicklung agentenorienter Informationssysteme für die Fertigungsleittechnik*. Dissertation, RWTH Aachen, 2002.

[KvH00] Kuldeep Kumar und Jos van Hillegersberg. Enterprise resource planning: introduction. *Communications of ACM*, 43(4):22–26, 2000.

[KZ02] Akhil Kumar und J. Leon Zhao. Workflow support for electronic commerce applications. *Elsevier Decision Support Systems*, 32(3):265–272, 2002.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, 1978.

[Ley96] Frank Leymann. Transaktionskonzepte für Workflow-Management-Systeme. In Gottfried Vossen und Jörg Becker, Editoren, *Geschäftsprozessmodellierung und Workflow-Management*, Seiten 335–351. Thomson Publishing, 1996.

[Ley99] Frank Leymann. A practitioners approach to database federation. In *Proceedings of 4th Workshop on Federated Databases - Integration of Heterogeneous Information Sources*, Berlin, Deutschland, 1999.

[LHM+86] Bruce Lindsay, Laura Haas, C. Mohan, Hamid Pirahesh, und Paul Wilms. A snapshot differential refresh algorithm. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, Seiten 53–60, New York, NY, USA, 1986. ACM Press.

[Li03] Qiang Li. Entwicklung einer graphischen Eingabemöglichkeit für Propagationsskripte. Studienarbeit, IPVS – Universität Stuttgart, 2003.

[Lin00] Davis S. Linthicum. *Enterprise Application Integration.* Addison-Wesley, 2000.

[LJdP97] Soon Huat Lim, Neal Juster, und Alan de Pennington. The seven major aspects of enterprise modelling and integration: a position paper. *SIGGROUP Bullutin*, 18(1):71–75, 1997.

[LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, und Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, Seiten 419–430, New York, NY, USA, 2005. ACM Press.

[LR00] Frank Leymann und Dieter Roller. *Production Workflow – Concepts and Techniques.* Prentice Hall, 2000.

[LR02] Framk Leymann und Dieter Roller. Using flows in information integration. *IBM Systems Journal*, 41(4):732 – 742, 2002.

[LSH03] Jinyoul Lee, Keng Siau, und Soongoo Hong. Enterprise integration with ERP and EAI. *Communications of ACM*, 46(2):54–60, 2003.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.

[MBR01] Jayant Madhavan, Philip A. Bernstein, und Erhard Rahm. Generic Schema Matching with Cupid. In *VLDB*, Seiten 49–58, 2001.

[McC93] Carma McClure. *Software-Automatisierung: reengineering – repository – Wiederverwendbarkeit.* Prentice-Hall International, 1993.

[MFJPPMK04] Jesús M. Milan-Franco, Ricardo Jiménez-Peris, Marta Patiño-Martínez, und Bettina Kemme. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Seiten 175–194, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[Mic01] Sun Microsystems. Jini Technology Core Platform Specification – Version 1.2. Specification, Sun Microsystems, 2001.

[Mic08] Microsoft. Microsoft BizTalk Server. Webseite (Zugegriffen am 21.08.2008), Microsoft Corporation, 2008. Verfügbar bei http://www.microsoft.com.

[Mit03] Nilo Mitra. SOAP Version 1.2 Part 0: Primer. W3C Recommendation, World Wide Web Consortium, 2003.

[MJHM09] Jorge Mínguez, Mihaly Jakob, Uwe Heinkel, und Bernhard Mitschang. A SOA-based Approach for the Integration of a Data Propagation System. In *Proceedings IEEE International Conference on Information Reuse. Integration IRI '09*, Seiten 47–52, New York, NY, USA, 10-12 August 2009.

[Mül05] Joachim Müller. *Workflow-based Integration – Grundlagen, Technologien, Management.* Springer, 2005.

[MR95] Stefan Morschheuser und Heinz Raufer. Integrated document and workflow management applied to the offer processing of a machine tool company. In *COCS '95: Proceedings of Conference on Organizational Computing Systems*, Seiten 106–115, New York, NY, USA, 1995. ACM Press.

[MRB03] Sergey Melnik, Erhard Rahm, und Philip A. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, Seiten 193–204, New York, NY, USA, 2003. ACM Press.

[MS00] Zakaria Maamar und Jeff Sutherland. Toward intelligent business objects. *Communications of ACM*, 43(10):99–101, 2000.

[MTvF00] M. Lynne Markus, Cornelis Tanis, und Paul C. van Fenema. Enterprise resource planning: multisite ERP implementations. *Communications of ACM*, 43(4):42–46, 2000.

[Oas07] Oasis. Web Services Business Process Execution Language Version 2.0. Oasis Standard, Sun Microsystems, Inc., 2007.

[Ora06] Oracle. Oracle Enterprise Service Bus. Data Sheet, Oracle Corp., 2006.

[Ora07] Oracle. Oracle Enterprise Service Bus. Developer's Guide, Oracle Corp., 2007.

[Ora08] Oracle. Oracle SOA Suite. Data Sheet, Oracle Corp., 2008.

[Pap06] Christian Pape. Enterprise Application Integration – Integrationsarchitekturen. Vorlesungsscript, Hochschule Karlsruhe – Technik und Wirtschaft, 2006.

[Pau93] G. N. Paulley. Engineering an SQL gateway to IMS. In *CASCON '93: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, Seiten 789–803. IBM Press, 1993.

[PB03] Rachel Pottinger und Philip A. Bernstein. Merging Models Based on Given Correspondences. In *VLDB*, Seiten 826–873, 2003.

[PG00] Robin Poston und Severin Grabski. The impact of enterprise resource planning systems on firm performance. In *ICIS '00: Proceedings of the twenty first international conference on information systems*, Seiten 479–493, Atlanta, GA, USA, 2000. Association for Information Systems.

[PMJPKA05] Marta Patiño-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, und Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[PS00] Esther Pacitti und Eric Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[Rah94] Erhard Rahm. *Mehrrechner-Datenbanksysteme.* Addison-Wesley, 1994.

[RB01] Erhard Rahm und Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[RCHM02] Ralf Rantzau, Carmen Constantinescu, Uwe Heinkel, und Holger Meinecke. Champagne: Data Change Propagation for Heterogeneous Information Systems. In *Proceedings of the International Conference on Very Large Databases (VLDB); Demonstration Paper; Hong Kong, August 20-23, 2002*. Morgan Kaufmann, August 2002.

[RMB01] William A. Ruh, Francis X. Maginnis, und William J. Brown. *Enterprise Application Integration – A Wiley Tech Brief*. John Wiley Computer Publisching, 2001.

[SAP07] SAP AG. SAP Bibliothek. Online Hilfe, SAP AG, http://help.sap.com (Zugriff 30.10.2007), 2007.

[SBB+99] David Shutt, Philip A. Bernstein, Thomas Bergstraesser, Jason Carlson, Shankar Pal, und Paul Sanders. Microsoft repository version 2 and the open information model. *Information Systems*, 24(2):71–98, 1999.

[Sch98] August-Whilhelm Scheer. *ARIS-Vom Geschäftsprozeß zum Anwendungssystem*. Springer Verlag, 3. Edition, 1998.

[Süd05] Süddeutsche Zeitung. BMW hängt Mercedes ab. Webseite (Zugegriffen am 22.07.2008), Süddeutsche Zeitung, 2005.

[SE98] Oliver Sims und Peter Eeles. *Building Business Objects*. John Wiley & Sons, Inc., 1998.

[SGB02] Rainer A. Sommer, Thomas R. Gulledge, und David Bailey. The n-tier hub technology. *SIGMOD Record*, 31(1):18–23, 2002.

[SH01] Michael Stonebraker und Joseph M. Hellerstein. Content integration for e-business. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Seiten 552–560, New York, NY, USA, 2001. ACM Press.

[SKTY00] Christina Soh, Sia Siew Kien, und Joanne Tay-Yap. Enterprise resource planning: cultural fits and misfits: is ERP a universal solution? *Communications of ACM*, 43(4):47–51, 2000.

[SL90] Amit P. Sheth und James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Survey*, 22(3):183–236, 1990.

[Son99] Sonderforschungsbereich 467. Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienfertigung. Finanzierungsantrag, Universität Stuttgart, 1999.

[Spr05] Sven Sprandel. Entwicklung eines Quelladapters für relationale Datenbanken. Diplomarbeit, IPVS, Universität Stuttgart, 2005.

[SRL00] Budi Surjanto, Norbert Ritter, und Henrik Loeser. XML Content Management Based on Object-Relational Database Technology. In *Web Information Systems Engineering*, Seiten 70–79, 2000.

[SS05] Yasushi Saito und Marc Shapiro. Optimistic replication. *ACM Computing Survey*, 37(1):42–81, 2005.

[Ste02] Michael Stender. *Eine komponentenorientierte Softwarearchitektur für Informationssysteme im Investitionsgütermarketing.* Dissertation, Universität Stuttgart, 2002.

[STSB02] Siew Kien Sia, May Tang, Christina Soh, und Wai Fong Boh. Enterprise resource planning (ERP) systems as a technology of power: empowerment or panoptic control? *SIGMIS Database*, 33(1):23–37, 2002.

[Stu10] Boris Stumm. *Änderungsmanagement in großen Informationssystemen.* Dissertation, Technische Universität Kaiserslauten, 2010.

[Tat01] Satish Tatte. XLANG – Web Services For Business Process Design. Specification, Microsoft Corporation, 2001. Verfügbar bei: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm [27.01.2004].

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, und Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. W3C Recommendation, World Wide Web Consortium, 2004.

[TIM+03] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin (Luna) Dong, Yana Kadiyska, Gerome Miklau, und Peter Mork. The Piazza peer data management project. *SIGMOD Record*, 32(3):47–52, 2003.

[TRA96] Francisco J. Torres-Rojas und Mustaque Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *10th Workshop on Distributed Algorithms (WDAG)*, Seiten 71–88, 1996.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, und C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, Seiten 172–182, New York, NY, USA, 1995. ACM Press.

[vdAK03] W. M. P. van der Aalst und Akhil Kumar. XML-Based Schema Definition for Support of Interorganizational Workflow. *Information Systems Research*, 14(1):23–46, 2003.

[vdAvH02] Wil van der Aalst und Kees van Hee. *Workflow Management – Models, Methods, and Systems.* MIT Press, 2002.

[vdAVK01] W. M. P. van der Aalst, H. M. W. Verbeek, und A. Kumar. Verification of XRL: An XML-based Workflow Language. In *Proceedings of the 6th International Conference on CSCW in Design*, Seiten 427–432. NRC Research Press, Ottawa, Canada, 2001.

[Vet90] Max Vetter. *Aufbau betrieblicher Informationssysteme – mittels konzeptioneller Datenmodellierung.* B.G. Teubner Stuttgart, 6. Edition, 1990.

[Vet94] Max Vetter. *Informationssysteme in der Unternehmung: eine Ein führung in die Datenmodellierung und Anwendungsentwicklung.* B.G. Teubner Stuttgart, 2. Edition, 1994.

[VHvdA02] H. M. W. Verbeek, A. Hirnschall, und W. M. P. van der Aalst. XRL/Flower: Supporting Interorganizational Workflows using XRL/Petri-net Technology. In *Lecture Notes in Computer Science: Web Services, E-Business, and the Semantic Web, CAiSE 2002 International Workshop (WES 2002)*, Seiten 93–108. Springer Verlag, Berlin, 2002.

[Vu05] Tien Minh Vu. Entwicklung eines Zieladapters für relationale Datenbanken. Diplomarbeit, IPVS – Universität Stuttgart, 2005.

[Wes06] Engelbert Westkämper. Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienproduktion – Sonderforschungsbereich 467. Abschlussbericht, Universität Stuttgart, 2006.

[WH99] Seth White und Mark Hapner. JDBC 2.1 API. Spezifikation, Sun Microsystems, Inc., 1999.

[WML+05] Scott Woodgate, Stephan Mohr, Brian Loesgen, Susie Adams, Alex Cobb, Benjamin Goeltz, Brandon Gross, Chris Whytock, Erik Leaseburg, Gavin Islip, Imran Aziz, Kevin Smith, Michael Roze, Naveen Goli, Puru Amradkar, und Stephen Roger. *Microsoft BizTalk Server 2004 – Unleashed.* Sams Publishing, 2005.

[Wor05a] Workflow Management Coalition. Reference Model. http://www.wfmc.org/standards/model2.htm [30.09.2005], 2005.

[Wor05b] Workflow Management Coalition (WfMC). Process Definition Interface – XML Process Definition Language (Version 1.09). Specification, Workflow Management Coalition (WfMC), 2005. Verfügbar bei: http://www.wfmc.org [01.07.05].

[Wor07] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C Recommendation 26 June 2007, W3C, 2007.

[WvB01] Engelbert Westkämper und Ralf von Briel. Continious improvement and participative factory planing by computer systems. In *Proceedings of the 51st General Assembly of CIRP*, Seiten 347–352, Nancy, France, 2001.

[WW02] Engelbert Westkämper und Ralph Winkler. The Use of System Modelling for the Intelligent Planning, Scheduling and Control of Agile Manufacturing. In *Proceedings of the 35th CIRP ISMS "Manufacturing technology in the information age"*, Seiten 644–655, Seoul, South Korea, 2002.

[WZ09] Engelbert Westkämper und Erich Zahn, Editoren. *Wandlungsfähige Unternehmensstrukturen – Das Stuttgarter Unternehmensmodell.* Springer Verlag, 2009.

[Yan04] Yingwei Yang. Distribution of the Propagation System. Master thesis, IPVS – Universität Stuttgart, 2004.

[Zah99] Ron Zahavi. *Enterprise Application Integration with CORBA – Component and Web-Based Solutions.* OMG Press - John Wiley Computer Publishing, 1999.