A SYSTEM FOR DATA CHANGE PROPAGATION IN HETEROGENEOUS INFORMATION SYSTEMS

Carmen Constantinescu*, Uwe Heinkel*, Ralf Rantzau, Bernhard Mitschang
Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
Email: {Carmen.Constantinescu, Uwe.Heinkel, Ralf.Rantzau, Bernhard.Mitschang}@informatik.uni-stuttgart.de

Keywords: Information systems, integration of heterogeneous data sources, XML technology

Abstract: Today, it is common that enterprises manage several mostly heterogeneous information systems to supply their production and business processes with data. There is a need to exchange data between the information systems while preserving system autonomy. Hence, an integration approach that relies on a single global enterprise data schema is ruled out. This is also due to the widespread usage of legacy systems. We propose a system, called Propagation Manager, which manages dependencies between data objects stored in different information systems. A script specifying complex data transformations and other sophisticated activities, like the execution of external programs, is associated with each dependency. For example, an object update in a source system can trigger data transformations of the given source data for each destination system that depends on the object. Our system is implemented using current XML technologies. We present the architecture and processing model of our system and demonstrate the benefit of our approach by illustrating an extensive example scenario.

1. INTRODUCTION

Corporate IT infrastructures typically comprise distributed heterogeneous information systems. In this paper, we focus on the management of data stored in these systems. In particular, we present a mechanism to transform and deliver data from one system to the needs of the receiving systems. The mechanism developed generates output data based on a collection of data dependencies between the distributed information systems and the transformation specifications associated with each dependency. This approach increases system autonomy, because specific system features are treated within a dependency only, and the usage of global structures is abandoned.

In this section, we give some background information on data change management for enterprise information systems and the motivation for developing our system. Then, we briefly introduce the main concepts used in our approach, which are further detailed in Section 2.

1.1 The Problem of Data Change Management

Most enterprises are forced to innovate their products and services and to improve their business processes ever more quickly to remain competitive. This is due to the turbulences observed on the markets: the supply of and demand for resources change more radically and abruptly than in the past. The ability to adapt to market changes is a key success factor.

1.1.1 Data Producers and Consumers

An enterprise can be viewed as a set of business units that are connected by information flows and resource flows. A business unit, characterized by the autonomy to manage its own resources, is responsible to deliver services or goods, which are further processed by other business units. Furthermore, business units frequently run their own information systems to manage the data tailored for the respective business processes.

Business units act as producers and consumers of information. Hence, change propagation between

* This work was partially supported by the German Research Society (DFG/SFB 467)

business units means, among other things, the propagation of data between the corresponding information systems. However, due to the autonomy of business units, not all information systems store information using the same data representation. Furthermore, some data is relevant only to a single information system, while other data is relevant to several systems and is stored locally, possibly using different representations. Thus, data changes in one system have different effects on data stored in other systems. Consider, for example, a manufacturing enterprise with several business units. Let us focus on the two service-oriented business units *Order Management* (OM) and *Facility Layout Design* (FLD). Suppose that a customer orders a large number of certain products. The OM business unit is responsible to generate optimal plans using production planning and control methods (Wiendahl & Westkämper, 2001). This may lead to a capacity increase of machines. The FLD business unit is responsible to generate factory plans that are used to adapt the real factory layout (Westkämper & von Briel, 2001).

Because of the need for additional machines, the FLD may have to position them in the factory layout. This causes a change of the material flow, affecting the processing of other orders as well. The information about new transportation times has an effect on OM, thus establishing a mutual information dependency between OM and FLD. Data changes in one of the information systems have to be reflected in the data of the other system. In general, an information system of a business unit produces data which is consumed by one or more other information systems, i.e., there is a 1-to-N data dependency. In our example, the OM business unit changes or creates data describing production plans. The FLD business unit then receives data on new production plans and adapts its data describing the factory layout. This example is further detailed in Section 3, showing technical aspects of the internal processing of our data change management system.

1.1.2 System Heterogeneity

Due to the autonomy of business units and the heterogeneity of their IT infrastructures, it is often infeasible or too expensive to manage a single, integrated enterprise information system that feeds all business units with their data. Hence, enterprise change management should be supported by a generic approach to solve data change management. Such an approach should be able to manage data dependencies and help to transform data stored in a source information system into data stored in the depending information systems.

1.2 A Generic Approach for Data Change Propagation

We first briefly discuss our approach using general terms and concepts that we introduce in this section. In Section 2, we present the architecture and details on the technologies employed in the proposed system.

A *data model* is a description of data structures. For example, in the object-oriented model, data structures are classes, and relations in the relational model. Classes also specify *methods* but we focus on the data aspect, only. Therefore, we are only interested in the *attributes* of the classes.

A *data schema* is an instance of a data model, i.e., the specification of data structures for an application. For example, in the relational model, an employee can be represented by a relation with the attributes (and types) *empno* (integer), *name* (string), *birth* (date), etc.

We define a *system* as an application or an information system. A system can act as a producer and/or consumer of data.

A directed relationship from a single data producer, named *source system*, to a single data consumer, named *destination system*, is called *dependency*. We say that the destination system is *dependent* on the source system.

A data change occurring in one system has an impact on all dependent systems. For example, two systems store data objects that represent the same information but possibly using different data models and data schemas. Thus, a change of such data in one system requires a change in the other, sometimes involving complex adaptations.

In this paper, we merely consider updates of instances of a data schema but we do not consider changes of the schema itself. For example, in the relational data model, our approach manages the update of attribute values in tuples of a relation but not the manipulation of attribute data types, the addition of attributes, etc. The management of schema (meta data) changes remains as our future work.

A *change propagation* is the process of forwarding a data change from a source system to a dependent system. The process of change propagation includes transformations and filtering of the changed data that is delivered to the dependent system.

A *transformation* is an operation that maps given input data into output data according to a specification. The specification defines how the contents and schema of the input data have to be adapted to represent valid target data.

Filtering is an operation that accepts or rejects the execution of a transformation according to a constraint, which is an expression (a set of condi-

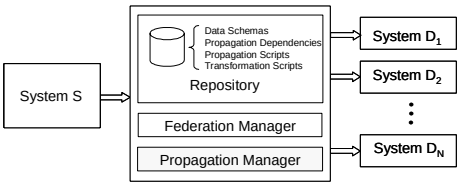

Figure 1: SIES, a data change management system

tions) over the contents of the input data of a transformation.

1.3 Overview of a Data Change Management System

We develop a data change management system, called Stuttgart Information and Exploration System (SIES) (Constantinescu, Heinkel, Rantzau & Mitschang, 2001), illustrated in Figure 1, which manages the change propagation between several systems. It consists of two main components, called *Propagation Manager* and *Federation Manager*. We use the term *propagation dependency* to describe a dependency as defined in Section 1.2. A propagation dependency specifies a source system, a single dependent system, and a *propagation script*. The propagation script contains information required for transformations (defined by *transformation scripts*), filtering, and routing. The Propagation Manager transforms, filters, and routes changed data using the dependencies that have been created, stored, and updated by the Federation Manager. The *Repository* stores propagation dependencies, schema descriptions of the systems, and propagation and transformation scripts.

We designed our system to have the following features. It must be possible to *add new systems* without much effort, i.e., SIES has to provide software, called wrappers, that is able to track data updates in a system and to map between its own data model and the Propagation Manager data model.

Another requirement is to support the *addition*, *removal*, and *update* of *dependencies* and *transformation specifications*. SIES needs to provide a user-friendly graphical user interface that allows to manipulate the dependencies and the associated scripts and to register them in a repository. Furthermore, SIES should offer a library of simple and frequently used transformations that can be called within newly created propagation scripts, tailored for a new system. Of course, the user interface should also allow to adapt given transformations, or to create new ones and to add them to the library for future reuse.

SIES has to provide a high degree of *reliability* and *availability*. For example, failures in participating systems have to be handled in a way that other systems are unaffected.

Finally, SIES has to be *scalable*, i.e., the resources (time and memory) required to manage dependencies should grow (linearly) in proportion to the number of dependencies and, of course, also in proportion to the amount and size of data to be transformed and propagated.

In this paper, we focus on one component of SIES, the Propagation Manager.

2. THE PROPAGATION MANAGER

The central component of our data change management approach is the Propagation Manager. This section describes this component regarding its functionality, architecture, and processing model.

2.1 Functionalities

The task of the Propagation Manager is to transform a source data object into a destination data object. Such a transformation has to be triggered automatically whenever the value of the source data object changes. Thus, the Propagation Manager combines two functionalities: a) the processing of a transformation and b) the propagation of updated data object values from one system to several dependent systems. The second functionality can be seen as a way to provide (some degree of) consistency between data stored in different systems.

2.2 Architecture

In this section, we describe the architecture of the Propagation Manager, which is responsible for change propagation. The Propagation Manager consists of several sub-components, sketched in Figure 2. One of them is the *Transformation Manager*, which itself is composed of the *Script Engine*, the *Mapper*, and the *Filter*. The Script Engine interprets a propagation script, associated with a propagation dependency. The script specifies several operations for transforming, filtering and routing the changed data. The Script Engine calls the Mapper to perform the transformation of a source data object (based on a change) into a destination data object. If the destination system requires special constraints, specified in the propagation script, the Script Engine invokes the Filter. The Filter then informs the Propagation Manager if a data change should be ignored or if the destination system has to be updated. The Queue Manager is used to exchange messages between the involved systems. It manages the Propagation Man-

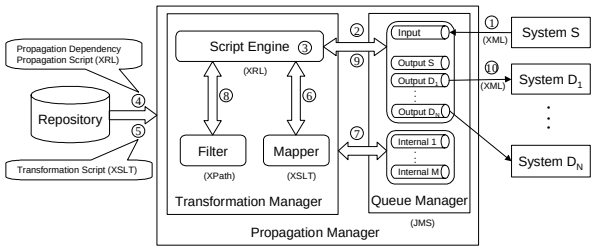

Figure 2: Propagation Manager architecture. The technologies employed are shown in brackets.

ager's input queue, the systems' output queues and several internal queues of the Transformation Manager.

2.3 Processing Model

In the following, we describe the internal flow of information needed to propagate a change, illustrated in Figure 2. The figure shows a sequence of processing steps which belong to four main stages:

1. Fetch a new message from the input queue (steps 1 and 2),
2. select applicable propagation dependencies (3, 4, 5),
3. execute corresponding propagation scripts (6, 7, 8), and
4. put processed messages into the output queues (9, 10).

In the following subsections, we explain these stages in detail.

2.3.1 Message In- and Output

The Queue Manager has a single input queue and for each connected system one output queue. It also manages the internal queues of the Transformation Manager to store intermediate transformation results. Whenever a system connects to the Propagation Manager, it is assigned an output queue where it can fetch messages from. We assume that an average system stays connected for a relatively long time.

All systems connected to the Propagation Manager have to be extended by wrappers. The wrapper is responsible to detect a change in the source system, called System *S* in Figure 2, to create a message according to a given format, and to send it to the input queue of the Propagation Manager (step 1 in Figure 2). A dependent system, named System D_i, is responsible for fetching the propagated message from its queue (10).

2.3.2 Dependency Selection

The dependency selection is initiated by the Queue Manager, which notifies the Script Engine about a new message in the input queue (2). Then, the Script Engine fetches the new message and interacts with the Mapper and the Filter components to transform and filter the given source data object according to each propagation script. The repository stores, among other things, propagation dependency specifications. They have the following structure: (*SourceSystem*, *PropagationScript*, *DestinationSystem*). The parameter *PropagationScript* denotes the name of a propagation script stored in and managed by the repository. The propagation script contains information regarding the transformation scripts and the output queue of the destination. The Script Engine extracts from the message header the name of the source system that has sent the message (3). Based on this information, the Script Engine retrieves all propagation dependencies matching the message's source from the repository (4). A matching dependency contains information about the propagation script and indicates the destination output queue.

A propagation script is a *template*, i.e., it contains the code to be interpreted by the Script Engine and placeholders that are replaced (using a simple parser) by textual values of the current dependency, like the actual names of the source and destination system. The code also specifies the names of one or more transformation scripts, which are stored in the repository. The use of templates allows a propagation script to be employed for various dependencies that require the same transformations. Hence the code of a propagation script does not have to be replicated for each dependency. A transformation script is a piece of code that is interpreted by the Mapper. The Script Engine retrieves the transformation scripts from the repository (5).

2.3.3 Propagation Script Execution

The propagation script execution consists of transforming, filtering and routing steps. It is triggered by the Script Engine which calls the Mapper (6). The Mapper performs the transformations corresponding to the retrieved transformation scripts. All messages exchanged between the Transformation Manager's components are temporarily stored in internal queues (7).

If the Script Engine detects a filter expression (a constraint) in a transformation script, it calls the Filter (8). The Filter decides if the message is routed to

the destination system or discarded. A filter expression can appear anywhere in a transformation script because the decision whether to skip a change can be based on results of transformations. If the Filter concludes that there be an output message, the Script Engine puts the message into the output queue of the dependent system (9).

2.4 Implementation

So far, we have introduced the general concepts and the architecture of our approach. In this section, we discuss the implementation of our concepts using various state of the art technologies.

2.4.1 Messages and JMS

The sources and destinations of data dependencies may have different data models and data schemas. We use XML to specify the schemas of the connected systems because of its flexibility to adapt it to ones needs. Suppose that a source system uses the relational data model to manage its data objects. For instance, consider a table (source object) consisting of many rows where only a single row has changed. We suggest using XML Fragment Interchange to send only a fragment of the original XML document, representing the changed row in the source data, together with some context information.

The communication between the Propagation Manager components as well as between the Propagation Manager and the external systems is implemented using the Java Message Service (JMS). The body of a message received from a source system contains the changed data according to the XML format. We employ for communication the message-oriented middleware which provides a means for asynchronous application-to-application communication via message queuing (Leymann, 1999).

2.4.2 Transformation Scripts and XSLT

We use an XML-based language, called eXtensible Stylesheet Language Transformations (XSLT), to transform XML documents in the Mapper component. During a complex propagation, several XSLT files can be involved. For a single transformation, an XSLT processor reads the XML document representing the source data and interprets it according to the specified transformation script. The calls of transformation scripts are embedded in propagation scripts.

2.4.3 Propagation Scripts and XRL

We use the eXchangeable Routing Language (XRL) (van der Aalst & Kumar, 2000; Kumar & Zhao, 1998) for propagating changed data between dependent systems. This is an XML dialect and can thus be processed by any XML parsers, which are widely available. XRL is aimed at asynchronous, flow-type applications and provides a mechanism to describe processes at an instance level.

We extended XRL according to our propagation purposes by three new elements: transform, propagate and message_event. A *transform* element specifies a subject message, transformed using a given transformation script, named *template*, and a *result*. The routing schema manages a processed message using an internal name. The *propagate* element puts a subject message into a specified destination queue. A *message_event* element defines an event. This allows to fire an action if a certain message arrives in an input queue. An example XRL file that contains these new elements is presented in Section 3.2 and illustrated in Figure 4.

2.4.4 Repository and UML

We specify the systems' data schemas using the Unified Modeling Language (UML), which it is becoming a de-facto standard for object-oriented modeling. We use a repository system as a common place to store systems' data schemas and the propagation and transformation scripts because of the following advantages: a) a repository is a shared database with value-added services like versioning, configuration management and context-management, and b) it is a valuable basis for the reuse of the stored objects.

3. EXAMPLE SCENARIO

In the following sections we describe an extensive example that illustrates the usage of our technologies introduced before. It highlights our main concepts for data change management: the transformation, filtering and routing of data changes.

3.1 Data Change Management in Manufacturing

Our scenario refers to a manufacturing enterprise that receives a new order. At least two systems of the enterprise are affected by the new order: OM and FLD, which store, among other things, data about the resources needed to manufacture products, like human resources and machines.

As a result of a new order from a customer, the OM System, the source system, creates a new, optimized production order. This describes the manufacturing process plan in terms of transportation and

```
<?xml version="1.0" encoding="ISO8859-2"?>
<production_order>
  <operation>
    <transport start="M002" dest="M001" ual="A12">
      <part no="P001" amount="5000"/>
    </transport>
  </operation>
  <operation>
    <drilling unit="M001" cost_center="DM1">
      <part no="P001" amount="5000"/>
      <manufacturing_lead_time days="0" hours="3"
                                        mins="40"/>
    </drilling>
  </operation>
  <operation>
    <transport start="M001" dest="M003" qual="C13">
      <part no="P001" amount="5000"/>
    </transport>
  </operation>
  <operation>
    <milling unit="M003" cost_center="BM2" >
      <part no="P001" amount="5000"/>
      <manufacturing_lead_time days="0" hours="1"
                                        mins="13"/>
    </milling>
  </operation>
</production_order>
```

Figure 3: Changed production order (om.xml)

manufacturing operations containing data regarding parts, machines, transportation and manufacturing lead times. The FLD System manages a database of the geometry (width, length, height) of machines as well as the current layout of the factory, i.e., the positions of the facilities. Several dependencies are defined between these systems. In order to create a new facility layout, based on simulation and optimization techniques, the FLD System processes the changed order data, potentially enriched by additional data from other systems. For example, if the changed production order involves a new machine, the FLD System will retrieve from its local data the corresponding geometry/size of the machine and an optimal position has to be computed.

In our scenario, the additional data represents the qualification and the hourly wage of the employee involved in operating a new machine, stored in a third system, the Strategic Management System (SM). In our approach, the additional data is not part of a propagation dependency. As a result, our scenario involves three systems: the OM and FLD Systems are dependent on each other, while the SM System has the role of a supplier of additional data.

We assume that the Federation Manager has already created the schema descriptions of these three systems and the propagation dependencies between them, including the propagation and transformation scripts. All this information is stored in the repository. The source system wrapper has to notify SIES about the change of some source data as well as to provide the changed data itself. The Propagation Manager then triggers the change propagation, based on the stored dependencies among the systems. The propagated change, enriched by the additional data, is then sent to the destination system, FLD, which computes a new facility layout.

```
<?xml version="1.0" encoding="ISO8859-2"?>
<xrl:ROUTE id="ex" created_by="script_designer"
                   creation_date="29-10-2001">
  <xrl:SEQUENCE>
    <xrl:WAIT sync="1">
      <xrl:MESSAGE_EVENT source="OM"
                         type="production_order" name="om"/>
    </xrl:WAIT>
    <xrl:TRANSFORM subject="om" template="om2fld.xslt"
                                name="t1"/>
    <xrl:PROPAGATE subject="t1" destination="FLDQ"/>
  </xrl:SEQUENCE>
</xrl:ROUTE>
```

Figure 4: Propagation script (om2fld.xrl)

3.2 Change Propagation

Based on this overall view of the scenario, we now present details of the scripts used to process our example change propagation.

We assume that the OM System changes data related to production orders. A production order consists of transportation and manufacturing operations applied to a part. A part is specified by its number and the ordered quantity. A manufacturing operation selects the machine corresponding to a specified processing step, i.e., drilling, milling, and the corresponding manufacturing lead time. A transportation operation specifies a start and a destination machine.

A change in the OM System is specified as an XML message that has to be propagated to the dependent system, FLD. Figure 3 presents the XML document (om.xml) representing the changed production order, extended by a drilling and a milling manufacturing process. As a result, two new machines are added, machines M001 and M003, and the associated manufacturing lead times and transportation operations are specified.

This message has to be transformed, filtered and routed. First, the message is put into the input queue of the Queue Manager. The Script Engine fetches the message and retrieves the propagation dependencies that match the specified OM source in the message header. One XRL propagation script (om2fld.xrl), is retrieved from the repository (Figure 4). It consists of the following routing schema:

1. Wait for an event representing a message from the OM System. It contains the changed production order (om.xml), managed internally under a specified name, *om*.
2. Transform the received message *om* using the transformation script (om2fld.xslt), specified as a template. The resulting message is assigned an internal name, here *t1*.
3. Propagate the subject *t1* to the specified destination output queue, here FLD queue.

```
<?xml version="1.0" encoding="ISO8859-2"?>
<xslt:transform version="1.0"
  xmlns:xslt="http://www.w3.org/1999/XSL/Transform">
  <xslt:output method="xml" encoding="ISO8859-2"
                                    indent="yes"/>

  <xslt:template match="/production_order">
    <material_flow>
      <xslt:apply-templates select="."/>
    </material_flow>
  </xslt:template>

  <xslt:template match="//transport">
    <xslt:element name="transport">
      <xslt:attribute name="start">
        <xslt:value-of select="@start"/>
      </xslt:attribute>
      <xslt:attribute name="dest"
        <xslt:value-of select="@dest"/>
      </xslt:attribute>
      <xslt:element name="part">
        <xslt:attribute name="no">
          <xslt:value-of select="part@no"/>
        </xslt:attribute>
        <xslt:attribute name="amount"
          <xslt:value-of select="part@amount"/>
        </xslt:attribute>
      </xslt:element>
      <xsl:variable name="qual" select="@qual"/>
      <xslt:element name="employee">
        <xslt:attribute name="qualification">
          <xslt:value-of select="@qual"/>
        </xslt:attribute>
        <xslt:attribute name="hourly_wage"
          <xslt:value-of select=
          "document('http://www.SM.com/employee.xml')
                  //employee[qual@q=$qual]h_wage@h_w"/>
        </xslt:attribute>
      </xslt:element>
    </xslt:element>
  </xslt:template>
</xslt:transform>
```

Figure 5: Transformation script (om2fld.xslt)

```
...
<material_flow>
  <transport start="M002" dest="M001">
    <part no="P001" amount="5000"/>
    <employee qualification="A12" hourly_wage="129"/>
  </transport>
  <transport start="M001" dest="M003">
    <part no="P001" amount="5000"/>
    <employee qualification="C13" hourly_wage="45"/>
  </transport>
</material_flow>
...
```

Figure 6: Propagated production order (fld.xml)

The Script Engine interprets the propagation script and retrieves the specified transformation script (om2fld.xslt) from the repository (Figure 5). This matches the *transport* node of the source document (om.xml) and retrieves its corresponding attributes (start, destination, qualification, part number, and part amount). These are used to define in the result document (fld.xml) (Figure 6) the associated *transport* node with the attributes *start* and *destination*, and the elements *part* and *employee*. The attributes of the *part* element (*number* and *amount*), are selected from the source document. The attributes of the *employee* element (*qualification* and *hourly_wage*) represent the required additional data from the SM System, specified by the URL of the document, http://www.SM.com/employee.xml.

The Mapper performs the transformation, illustrated before, using an XSLT processor, and sends the result to the Script Engine. In our example, no filter expression is involved. The resulting XML file is put into the body of a message that is delivered to the FLD System's output queue. It contains the following data: a) the transportation operations from machine M002 to M001 and from M001 to M003, respectively and b) the employees' qualifications and hourly wages corresponding to the manufacturing operations.

4. RELATED WORK

Our approach is designed to propagate changed data between dependent autonomous and heterogeneous systems. One of the main objectives is to provide support for routing of changed data between dependent systems. We envision two directions that achieve this objective. One is to use a language comprising routing constructs required to design a routing schema. This approach belongs to the field of *workflow definition* and *workflow management systems* (van der Aalst & Kumar, 2000; Kumar & Zhao, 1998). Our Propagation Manager implementation is based on XRL, a simple routing language, which is less powerful than a workflow definition language.

The second strategy is to build *(heterogeneous) agent systems* that can communicate intelligently with one another (Nwana, 1996). Heterogeneous agent systems are represented by the integration of two or more agents which belong to two or more different agent classes (collaborative, interface, mobile, information/Internet, reactive, hybrid and smart agents).

Another field related to our approach is *replication*, which is the process of maintaining several copies of the same data in several systems. The objective is to access data locally in order to improve response times and to reduce the communication overhead with other systems.

There are two main techniques to update replicated tables: table snapshots (Adiba & Linfsay, 1980) and triggers (Hanson & Widom, 1993). Table snapshots allow an asynchronous update of replicas, while the processing of triggers is synchronous. The trigger approach is more flexible than table snapshots since it allows defining actions being applied to replicas that are different from the update operations applied to the original copy. Furthermore, other objects/tables besides the replicas can be manipulated as well. The concept of triggers is a characteristic of *active database systems*. Triggers are closer to our approach than table snapshots due to their flexibility for transformations. Furthermore, our approach is designed to preserve the local autonomy of the data sources as much as possible. For example, we require the data sources to provide a mechanism to detect changes. If a data source happens to be a database system then triggers could be used to notify

the local wrapper about the data change, which is then responsible to generate an input message for the Propagation Manager.

5. CONCLUSION AND FUTURE WORK

Current IT infrastructures need a flexible, loosely coupled approach to propagate data changes between enterprise information systems while preserving their data management autonomy. We suggest a software component, called Propagation Manager, that manages dependencies between data stored in potentially different schemas and models. The data is not replicated in the Propagation Manager but only the data schemas of the systems connected to the Propagation Manager are stored in a repository. The Propagation Manager transforms an XML input message into an XML output message based on a transformation specification that has been defined for a data dependency. Such a transformation can be composed of several smaller transformation activities (XSLT scripts). By employing script templates, our approach allows for a reuse of transformation code for many dependencies.

In some situations it is useful to prevent data propagation. This can be achieved by applying constraints (XPath expressions) on the message input or on the intermediate results of transformations.

Our current propagation approach and system implementation manages 1-to-1 data dependencies. We currently investigate how to manage 1-to-N and M-to-N dependencies using a single XRL script instead of employing several 1-to-1 XRL scripts. This may have positive performance effects for the Propagation Manager because only one script instead of several would have to be processed. We intend to employ the XQuery language to query the repository. Furthermore, we plan to integrate XQuery statements into the transformation scripts to offer even more flexibility for transformations. This would add another XML technology to our system, thus making it an XML-dominated middleware component.

Meta data changes do not occur as frequently as changes of operational data. However, it is important to support the change of data schemas in our approach. As we have mentioned in Section 1.2, the Propagation Manager does not cover such changes but we intend to extend our system into this direction.

REFERENCES

van der Aalst, W., & Kumar, A. (2000). XML Based Schema Definition for Support of Inter-organizational Workflow. In *Proceedings of 21st International Conference on Application and Theory of Petri Nets* (ICATPN 2000), Aarhus, Denmark.

Adiba, M., & Lindsay, B. (1980). Database Snapshots. In *VLDB*, Montreal, Quebec, Canada, pp. 86-91.

Constantinescu, C., Heinkel, U., Rantzau, R., & Mitschang, B. (2001). SIES – An Approach for a Federated Information System in Manufacturing. In *Proceedings of the International Symposium on Information Systems and Engineering* (ISE 2001), Las Vegas, Nevada, USA, pp. 269-275.

Gray, J., Helland, G., O'Neill, P., & Sasha, D. (1996). The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, pp. 173-182.

Hanson, E., & Widom, J. (1993). An Overview of Production Rules in Database Systems. In *The Knowledge Engineering Review*, Vol.8, No.2.

Kumar, A., & Zhao, Z. (1998). Workflow Support for Electronic Commerce Applications. In *International Conference on Telecommunications and Electronic Commerce*, Nashville, TN, USA.

Leymann, F. (1999). A Practitioners Approach to Data Federation. In *4. Workshop Föderierte Datenbanken*, Berlin, Germany, CEUR-WS/Vol. 25.

Nwana, H. (1996). Software Agents: An Overview. In *Knowledge Engineering Review*, Vol. 11, No. 3, pp. 205-244.

Westkämper, E., & von Briel, R. (2001). Continuous Improvement and Participative Factory Planning by Computer Systems. In *CIRP Annals Manufacturing Technology*, Nancy, France, pp. 347-352.

Wiendahl, H.-H., & Westkämper, E. (2001). Situation-Based Selection of PPC Methods: Fundamentals and Approaches. In *CIRP 34th International Seminar on Manufacturing Systems*, Athens, Greece, pp. 241-246.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., & Alonso, G. (2000). Understanding Replication in Databases and Distributed Systems. In *Proceedings of 20th International Conference on Distributed Computing Systems* (ICDS 2000), Taipei, Taiwan, Republic of China, pp. 264-274.