A Data Change Propagation System for Enterprise Application Integration

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

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

Abstract
Most enterprises have a diverse environment of heterogeneous and autonomous information systems. If the same data is relevant for several information systems, then data changes in one supplier system affect data stored in other demander systems. The process of exchanging changed data between systems, named change propagation, is based on dependencies established between these systems. The management of a single, integrated enterprise information system is often infeasible or too expensive, due to the autonomy of business units and the heterogeneity of their IT infrastructures. The solution is to support the enterprise by a generic approach able to manage data dependencies and to transform data stored in a source information system into data stored in the dependent information systems. We propose a loosely coupled system, called Stuttgart Information and Exploration System. Our prototype mainly consists of a data dependency specification tool, a propagation engine and a repository that stores all relevant objects for these components.

1 INTRODUCTION

1.1 The Problem of Enterprise Application Integration

It is an understatement by far to say that the new information economy is changing the way enterprises do business. More specifically, it is changing the way enterprises manage the information they need for their production and business processes. Valuable information once produced and then stored in corporate IT infrastructures is distributed to all new groups of information consumers: remote employees, business partners and customers. As a result, the enterprise remains competitive and achieves huge benefits in terms of cost savings and improved customer relationship. In order to fulfill these objectives, the enterprise leverages their existing business applications by integrating them into new applications, a practice commonly referred to as *Enterprise Application Integration* or *EAI*. In this paper, we offer a solution to EAI involving heterogeneous information systems. The problem of integrating heterogeneous and autonomous information systems is approached from the perspective of managing data stored in these systems.

Our approach presents a system, called *Stuttgart Information and Exploration System* (SIES), which manages the exchange of data between the involved information systems. This section presents the background of our approach and the motivation of our work. Section 2 introduces our system, SIES, by its main concepts, architecture and implementation details. The propagation manager, the dependency manager, the systems' adaptors and the repository are presented in Sections 3, 4, 5 and 6. The last two sections present the related work and the conclusions.

1.2 Motivation Scenario

The motivation of our approach is the manufacturing enterprise, which manages several autonomous, and heterogeneous information systems and legacy systems. Our scenario focuses on two IT-oriented business units *Order Management* (OM) and *Facility Layout Design* (FLD) involved in processing a considerable increase in product demand.

The OM business unit is responsible to generate optimal plans using production planning and control methods (Wiendahl & Westkämper, 2001). As a possible result, the enterprise has to increase its capacity by purchasing new equipment. This is reflected in the FLD business unit, which is concerned with generating factory plans that are used to adapt the real factory layout (Westkämper & von Briel, 2001). The newly produced factory layouts involve changes of the material flow, which affect the transportation times and the processing of other orders by the OM as well. Thus, a bi-directional information dependency between OM and FLD business units is established. Data changes in one information system have to be reflected in the data of other systems. In general, an information system of a business unit pro-

* This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG/SFB 467)

duces/supplies data, which is consumed/demanded by one or more 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 received by the FLD business unit, which adapts its data describing the factory layout.

2 A DATA CHANGE MANAGEMENT SYSTEM

2.1 Basic Concepts

Our approach is based on several concepts that we introduce in this section:

We define an *information system* as any software system that offers access to data.

A *data model* is a description of data structures. For example in the object-oriented model, data structures are classes, and in the relational model, data structures are relations. Classes can also specify *methods* (the behavior of objects). As we focus on the data aspect, 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 information system. 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.

A 1-to-N relationship between a source system/schema combination and one or more destination system/schema combinations is called a *dependency*. We say that a 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 dependent system, sometimes involving complex adaptations of the data. In our approach, 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.

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

A *transformation* is an operation that maps given input data to output data according to a specification which defines how the contents and schema of the input data has to be adapted to represent valid destination data. A *filter* is an operation that applies a boolean expression to input data. If the condition is true, the result represents the input data; otherwise, no output data is returned. A system which takes benefit of change propagation offered by our data change management system is named *coupled* or *connected system*.

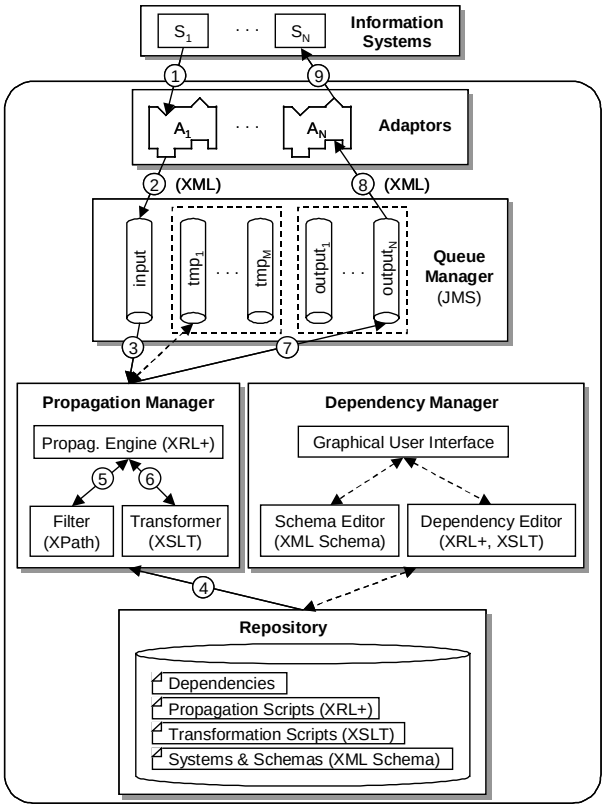

Figure 1. SIES, a data change management system

2.2 Stuttgart Information and Exploration System

2.2.1. Architecture

We develop a data change management system, called *Stuttgart Information and Exploration System* (SIES) (Constantinescu, Heinkel, Rantzau & Mitschang, 2001), which manages the change propagation between any connected systems. In Figure 1, we present the architecture of SIES as well as the control flow during the processing of data change propagation between the source system S_1 and the dependent system S_N.

Our prototype consists of the following main components, the *propagation manager*, the *dependency manager*, the *adaptors* and the *repository*. An adaptor maps data between the local schemas of the connected systems and the corresponding XML schema representation. The propagation manager transforms, filters, and routes changed data using the dependencies that have been created and stored by the dependency manager. The repository stores the XML sche-

mas of the systems, the dependencies and their associated propagation and transformation scripts, adding flexibility and extensibility to the entire system.

2.2.2. Implementation

In this section, we discuss the implementation of our concepts using various state of the art technologies.

The objective of our work is to enable interoperability between heterogeneous information systems. We address two fundamental interoperability issues. The first issue, semantic interoperability, is to define and agree on the semantics of the content and logical structures of data. The second issue, syntactic interoperability, is to define a platform-independent data structure that can represent data corresponding to the system model. We use the Extensible Markup Language (XML) as data interchange format because of its simplicity and extensive tool support. We specify the data schemas of the connected systems according to their XML schemas.

For communication, we employ message-oriented middleware, which provides a means for asynchronous application-to-application communication via message queuing (Leymann, 1999). We implement the communication between SIES components as well as between SIES and the connected systems using the Java Message Service (JMS).

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 based-language and can therefore be processed by any XML parser, which are widely available. Our approach implements a propagation script as an XRL file. XRL provides a mechanism to describe processes at an instance level and is thus mainly used in asynchronous, flow-type applications.

We employ the XML-based language eXtensible Stylesheet Language Transformations (XSLT) to transform XML documents in the propagation manager. We implement a transformation script as an XSLT file. Several XSLT files can be involved in a complex propagation. 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.

2.2.3. Features

We designed our system to have several features. The requirement to *connect* (*add*) *new systems* to SIES is provided by *adaptors*, components which can track data updates in a system and map between the data model of SIES and the connected systems.

SIES supports the *addition, removal,* and *update* of *dependencies* and *transformation specifications* by providing a user-friendly graphical interface. It facilitates the management of dependencies and associated scripts and increases the reusability by offering built-in reusable transformation scripts, which are stored in the repository.

Our system has to manage the failures in connected systems in order to protect other systems, providing a high degree of *reliability* and *availability*. As a final requirement, 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.

3 PROPAGATION MANAGER

The central component of our approach is the propagation manager. This section presents its functionalities and architecture, the propagation and transformation scripts as a base for the processing model.

3.1 Functionalities and Architecture

The propagation manager combines two functionalities: the transformation of a source data object into a destination data object and 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 consistency between data stored in different systems. The propagation manager consists of several sub-components, presented in Figure 1: the *propagation engine*, the *transformer*, and the *filter*. The propagation engine interprets a propagation script, associated with a dependency. The script specifies several operations for transforming, filtering and routing the changed data. The propagation engine calls the transformer to perform the transformation of a source data object into a destination data object. If the destination system requires special constraints, specified in the propagation script, the propagation engine invokes the filter. The filter then informs the propagation engine 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 connected systems. It manages the propagation manager's input queue, the adaptors' output queues and several internal queues of the propagation engine.

3.2 Propagation and Transformation Scripts

The processing of an input XML document is specified as a *propagation script.* This conforms to the *eXchangeable Routing Language* (XRL). We extended the language, called *XRL+*, with constructs for parallel and sequential execution, propagations, transformations, filtering, as well as event wait conditions:

TRANSFORM (xml_in, xml_out, xslt)
FILTER (xml_in, xml_out, xpath),
MESSAGE_EVENT (system, schema, xml_out) and
PROPAGATE (system, schema, xml).

The attributes *xml*, *xml_in*, and *xml_out* are IDs of XML documents, *xslt* is the ID of a transformation script, *xpath* is an XML Path Language (Xpath) expression, and *system* and *schema* are the IDs of the source and destination system and schema used.

Our XRL+ engine interprets a propagation script. It delivers a transformation script to the transformer, an *XSLT* processor, whenever it encounters an XRL+ *TRANSFORM* element. If the engine processes a *FILTER* element, the filter's Xpath expression is evaluated with an XML document as input. The *MESSAGE_EVENT* element waits for the arrival of a specific message, belonging to a given system/schema combination, and fetches it from the input queue. The *PROPAGATE* element sends a message containing a data change to a destination system's output queue.

A transformation is specified in an XSLT file, called *transformation script*. Such a script consumes and produces an XML document, offering powerful mapping operations. The input/output document conforms to an input/output XML schema that is defined in the repository. An extended example scenario including these scripts is presented in Constantinescu, Heinkel, Rantzau, & Mitschang (2002).

3.3 Processing Model

In the following, we describe the internal flow of information needed to propagate a change, illustrated in Figure 1. The figure shows a sequence of processing steps, which belong to four main stages: fetch a new message from the input queue (steps 1 and 2), select applicable propagation dependencies (3, 4), execute corresponding propagation scripts (5, 6, 7), and put processed messages into the output queues (8, 9).

The queue manager administrates a single input queue for the propagation manager, one output queue for each adaptor corresponding to a connected system and several internal queues necessary to store intermediate transformation results. Whenever a system connects to the propagation manager, it is assigned an output queue from where its adaptor can fetch messages. A message consists of three parts: header, properties and body. The message properties specify the action that has occurred in the data source (update, insertion, deletion). The message body includes the XML representation of the changed data object: the values of a newly inserted object, the new values of an updated object, or the values of a deleted object.

The communication between the connected systems and the propagation manager is facilitated by adaptors, presented in Section 5. First, the adaptor A_1 of system S_1 either detects a changed data object or is notified by S_1 (1). Then, A_1 maps the changed data object into an XML representation conforming to an XML schema, which has been stored in the repository using the dependency manager's schema editor. Next, the adaptor puts a message containing the XML representation of the changed object into the input queue of the

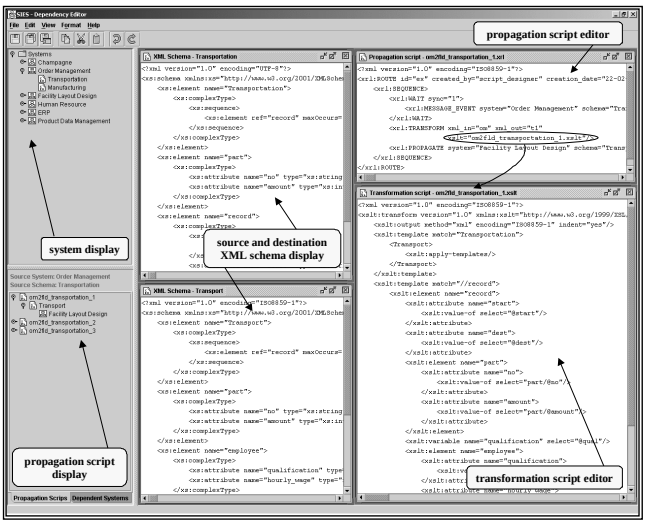

Figure 2. The graphical user interface of the dependency editor

queue manager (2). The adaptor of each destination system fetches the message from its output queue (8), maps the XML representation of the data object into its local representation, and performs or triggers the operations (insertion/update/deletion), which are specified in the properties of the output message (9).

The dependency selection is initiated by the queue manager, which notifies the propagation engine about a new message in the input queue (2). Then, the propagation engine fetches the new message from the input queue (3) and retrieves all dependencies from the repository where the source matches the specified system/schema combination (4).

The propagation script associated with each dependency is interpreted by the propagation engine, which interacts with the filter and the transformer components to transform and filter the given source data object according to specifications in the propagation script (5, 6). During the processing, intermediate transformation results are stored in temporary queues. The final data objects that result from the transformations are put into the output queues of each dependent destination system (7).

4 DEPENDENCY MANAGER

Another component of our data change management system is the dependency manager. The main task of the dependency manager is to register and update the content of the repository, i.e., the XML schemas of the systems, the dependencies, and their associated propagation and transformation scripts. In this section, we present the functionalities of the dependency manager and detail its main component, the dependency editor.

4.1 Functionalities and Components

The dependency manager is the design-time component of our approach, which brings together two functionalities. The first one is to create the XML schemas of the connected systems, and to store and update them in the repository. This is achieved by the *schema editor*. The *dependency editor* accomplishes the second functionality, the creation and management of the dependencies and their associated scripts.

4.2 Dependency Editor

The dependency editor provides a user-friendly graphical user interface that allows to manipulate the dependencies and their associated scripts and to register them in the repository. Furthermore, it offers a library of simple and frequently used transformations that can be called within newly created propagation scripts, tailored for a new system. The user interface also allows to adapt given transformations, to create new ones and to add them to the library for future reuse. A dependency is created through a sequence of steps mainly representing two stages: select the source system and access its XML schema, and create or update the propagation script and the associated transformation scripts.

We introduce these stages by using the graphical user interface presented in Figure 2. This illustrates the dependency created for our example systems, the source OM and the destination FLD.

The dependency editor performs source selection in several steps. First, selects the source system and one of its schemas (system display panel in Figure 2). The retrieved XML schema from the repository is displayed in the source XML schema display window. In our example, OM represents the source system and Transportation the selected and displayed schema.

This second activity deals with propagation and transformation scripts creation or update. The scripts are created as new ones or by using the reusable built-in scripts registered in the repository. The first step of this stage retrieves all dependencies (propagation scripts) for the specified source system and schema. Then, the propagation script names and the schema names of the destination systems are displayed in the propagation script display panel. The selected propagation script and its associated destination schemas and transformation scripts are retrieved from the repository and displayed in their related windows. There is a window for each destination schema and an editor for each transformation script referenced in the propagation script. In Figure 2, the retrieved and displayed propagation script names are *om2fld_transportation_1.xrl*, *om2fld_transportation_2.xrl* and *om2fld_transportation_3.xrl*. The XML schemas Transportation and Transport are displayed and the selected propagation script, *om2fld_transportation_1.xrl*, and its associated transformation script, *om2fld_transportation_1.xslt*, are displayed for editing purposes.

By accepting the referenced transformation script or editing it according to the desired transformation, the dependency creation process continues. In case of an update, a new transformation script is created and stored into the scripts library, in the repository. The last action updates the current propagation script as well, and stores it in the repository.

5 ADAPTORS

The communication between the connected systems and the propagation manager is facilitated through a set of software components, named *adaptors* in Figure 1. We define an adaptor as software that handles the communication between the connected systems and the propagation manager. It provides a bi-directional translation between a local data representation and a representation that conforms to an XML schema. An adaptor is responsible for putting a message containing changed data objects of its associated information system into the input queue of propagation manager, and or fetching a message for its system from the corresponding output queue. Suppose, for example, that the information system is a relational database system. If a table of a database is the *source* of a dependency then triggers can be employed to deliver the changed data objects (set of records) to the adaptor. If the database system is the *destination* of a dependency, then the adaptor may use SQL DML statements to update/insert/delete the affected records in the database. We differentiate two types of adaptors, depending on the requirements of the dependency and the characteristics of the data source: active adaptors that are able detect a change and passive adaptors that are notified by the data source.

6 REPOSITORY

Our system uses a repository, based on an object-relational database system, as a common place for all objects that need to be stored persistently: information system ID and names and their associated XML schemas, dependencies, consisting of the IDs of the source and destination systems and schemas, as well as the ID of the propagation script, propagation scripts involved in any dependency, transformation scripts referenced in any propagation script, and authentication information needed when systems connect to SIES. We intend to store even more data in our repository, for example the time when a system has connected to or disconnected from SIES, or a log of (a subset of) the messages exchanged. Such a data collection will then be subject to analysis with business intelligence tools and it may reveal a potential for communication optimizations. A possible result may illustrate that currently distributed data should better be integrated into a single schema because of a considerable communication overhead observed for propagations.

7 RELATED WORK

Our system propagates changed data between dependent autonomous and heterogeneous systems. We envision two directions on providing support for routing changed data. The first is related to the field of *workflow definition* and *workflow management systems* (van der Aalst & Kumar, 2000; Kumar & Zhao, 1998). Our implementation uses XRL+, a simple workflow definition language which offers routing constructs required to design a propagation schema. The second strategy is to build *(heterogeneous) agent systems* that can intelligently communicate with one another (Nwana, 1996).

Related to our approach is the field of *replication.* Between the two main techniques to update replicated tables, table snapshots (Adiba & Lindsay, 1980) and triggers (Hanson & Widom, 1993), triggers are closer to our approach than table snapshots due to their flexibility for transformations. Table snapshots allow an asynchronous update of replicas, while the processing of triggers is synchronous. Furthermore, our approach is designed to preserve the local autonomy of the data sources as much as possible by providing a mechanism to detect changes in a data source. We have to mention here two other approaches: IBM DB2 DataPropagator, a solution designed for database replication, and Microsoft BizTalk Server, which offers transformation facilities for business data. Our system, positioned between IBM DB2 DataPropagator and BizTalk Server, creates the XML schemas of the connected systems and offers a more flexible specification of dependencies.

8 CONCLUSION AND FUTURE WORK

Corporate IT infrastructures consist of distributed heterogeneous information systems. In this paper, we focus on the management of data stored in these systems and propose a method to transform and deliver changed data from one system to the needs of the receiving systems. We present a loosely coupled approach to propagate data changes between enterprise information systems while preserving their autonomy. By generating output data based on a collection of data dependencies between the involved information systems and on the transformation specifications associated with each dependency, our solution retains system autonomy.

We offer essential data interchange and interoperability features by employing XML as the interchange format and data schema. The propagation manager administrates dependencies between data stored according different schemas and models. We do not replicate the data; we maintain data schemas of the involved information systems in a repository system, adding flexibility and extensibility to the entire system. The propagation manager transforms an XML input message into XML output messages based on the transformation specifications (XSLT scripts) defined for a dependency by the dependency manager. In order to prevent, in some cases, the data propagation, we apply filter constraints (XPath expressions) on initial message input or on intermediate transformation results.

We orient our future work into the direction of optimizing the access of the objects stored in the repository. For querying the repository, we intend to employ the XQuery language. Furthermore, we plan to use XQuery instead of the XSLT scripts, thus offering more flexibility to the transformation processes.

In the real life of an information system, the meta data changes do not occur as frequently as changes of operational data. However, our system has to offer support to the change of data schemas of the involved systems. While our current prototype does not implement such changes, we plan to direct our work into this direction, as well.

9 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)*, Las Vegas, Nevada, USA, pp. 269-275.

Constantinescu, C., Heinkel, U., Rantzau, R., & Mitschang, B. (2002). A System for Data Change Propagation in Heterogeneous Information Systems. In *Proceedings of the 4th International Conference on Enterprise Information Systems*, Ciudad Real, Spain, Vol 1, pp. 73-80.

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.