Second International Workshop on Evolution and Change in Data Management (ECDM 2002)

Tampere, Finland, October 11, 2002

Held in conjunction with the 21st International Conference on Conceptual Modelling, ER 2002

Fabio Grandi
John F. Roddick
(Co-Chairs)
Change is a fundamental but sometimes neglected aspect of information and database systems. The management of evolution and change and the ability for database, information and knowledge-based systems to deal with change is an essential component in developing and maintaining truly useful systems. Many approaches to handling evolution and change have been proposed in various areas of data management and this forum seeks to bring together researchers and practitioners from both more established areas and those from emerging areas to look at this issue. The second ECDM workshop (the first ECDM workshop was held with ER’99 in Paris and its report can be found in SIGMOD Record 29(1):21-25, March 2000) deals with the manner in which change can be handled, and the semantics of evolving data and data structure in computer based systems. Workshop topics include:

- Semantics of Change in Time and Space
- Modelling and Management of Time-Varying Data, Temporal Databases
- Handling Changes and Versioning of Semi-structured Data
- Handling Changes of Metadata, Schema Evolution and Versioning
- Change Detection, Monitoring and Mining
- Evolution and Change in Internet-based Information Systems
- Evolution and Change in e-services and e-world Systems
- Induction of Cause and Effect, Logics for Evolution
- Maintenance of Views, Summaries, Dictionaries and Warehouses
- Managing Evolution of Sources in Information Integration

With respect to the main ER conference, the ECDM workshop aims at stressing the evolutionary aspects involved in conceptual modelling and in development and implementation of systems, ranging from the modelling of information dynamics to the dynamics of the modelling process itself. Another explicit aim of ECDM 2002 (as it was for ECDM ’99) it to bring together scientists and practitioners interested in evolution and change aspects in different research fields and, thus, often belonging to completely separate communities. It is our opinion that such an interaction can be tighter and cross-fertilization more useful in the context of a collaborative workshop like ECDM than in the context of the main conference sessions. Moreover, since the emphasis is on the evolutionary dimension, a special insight is sought upon this specific aspect, that hardly could find an appropriately broad coverage in the scope of the main ER conference.

Following the acceptance of the workshop proposal by the ER 2002 organizing committee, an international and highly qualified program committee was assembled from research centers worldwide. As a result of the call for papers, the program committee received 19 submissions from 15 countries and after rigorous refereeing 10 high quality papers were eventually chosen for presentation at the workshop which appear in these proceedings.

We would like to thank both the program committee members and the additional external referees for their timely expertise in reviewing the papers. We would also like
to thank all authors for submitting their papers to this workshop. Last but not least, we would like to thank the ER 2002 organizers for their support and in particular the workshop co-chairs, Antoni Olivé, Eric Yu and Masatoshi Yoshikawa.

September 2002

Fabio Grandi, John Roddick

Workshop Website

http://kdm.first.flinders.edu.au/events/ECDM02.html

Workshop Organizers

ECDM 2002 Co-Chair Fabio Grandi (University of Bologna, Italy)
ECDM 2002 Co-Chair John Roddick (Flinders University, South Australia)

Program Committee

Michael Böhlen Aalborg University, Denmark
Curtis Dyreson Washington State University, USA
Ramez Elmasri University of Texas, USA
Enrico Franconi University of Manchester, UK
Kathleen Hornsby University of Maine, USA
Richard McClatchey University of the West of England, UK
Erik Proper University of Nijmegen, The Netherlands
Chris Rainsford DSTO, Australia
Sudha Ram University of Arizona, USA
Elke Rundensteiner Worcester Polytechnic Institute, USA
Maria Rita Scalas University of Bologna, Italy
Myra Spiliopoulou Leipzig Graduate School of Management, Germany
Babis Theodoulidis UMIST, UK
Carlo Zaniolo UCLA, USA
# Table of Contents

## Management of Time and Changes in Information Systems

Change Management for a Temporal Versioned Object-Oriented Database

*Renata de Matos Galante, Nina Edelweiss, Clesio Saraiva dos Santos*

Towards Temporal Information in Workflow Systems

*Carlo Combi, Giuseppe Pozzi*

Preserving and Querying Histories of XML-Published Relational Databases

*Fusheng Wang, Carlo Zaniolo*

A Lightweight Constraint Check Framework for Consistently Updating XML Documents

*Hong Su, Bintou Kane, Victor Chen, Cuong Diep, De Ming Guan, Jennifer Look, Elke A. Rundensteiner*

A Framework for Change Discovery in Ontology-based Systems

*Alexander Maedche, Ljiljana Stojanovic, Nenad Stojanovic*

## Architectures, Models and Tools for Systems Evolution

An Architecture for Managing Database Evolution

*Eladio Dominguez, Jorge Lloret, Maria Antonia Zapata*

Reifying Design Patterns to Facilitate Systems Evolution

*Florida Estrella, Zsolt Kovacs, Jean-Marie Le Goff, Richard McClatchey, Tony Solomonides, Norbert Toth*

Managing Configuration with Evolving Constraints in Design Databases

*Tom W. Carnduff, Jeevani S. Goonetillake*

Intentions of Operations Characterization and Preservation

*Mira Balaban, Steffen Jurk*

An Active Approach to Model Management for Evolving Information Systems

*Henrik Gustavsson, Brian Lings, Bjorn Lundell*

## Author Index

*Author Index*
Change Management for a Temporal Versioned Object-Oriented Database *

Renata de Matos Galante, Nina Edelweiss, and Clesio Saraiva dos Santos

Federal University of Rio Grande do Sul, Cx Postal 15064, 91501-970 Porto Alegre - RS, Brazil
{galante, nina, clesio}@inf.ufrgs.br

Abstract. In this paper, we propose a schema versioning mechanism to manage the schema evolution in temporal object-oriented databases. The schema evolution management uses an object-oriented data model that supports temporal features and versions definition - the Temporal Versions Model - TVM. One interesting feature of our proposal is that TVM is used to control not only the schema versioning, but also the storage of extensional database and propagation of the changes performed on the objects. The extensional data level supports integration with the existing database, allowing the maintenance of conventional and temporal versioned objects. The instance propagation approach is proposed through the specification of propagation and conversion functions. These functions assure the correct instance propagation and allow the user to handle all instances consistently in both backward and forward schema versions. Finally, the initial requirements concerning data management in the temporal versioning environment, during schema evolution, are presented.

1 Introduction

Object-oriented databases offer powerful modelling concepts as those required by advanced application domains as CAD and Case tools. Typical applications handle large and complex structured objects, which frequently change their value and structure. As the structure is described in the schema of the database, support to schema evolution is a highly required feature. In this context, the version concept has been applied to maintain all the history of the database evolution.

Schema evolution and schema versioning are two techniques that allow schema modifications while consistency is maintained between a schema and its data. According to accepted terminology [1], a database supports schema evolution if it allows schema changes without losing extentional data. In addition, the schema versioning support allows not only the maintenance of data, but also the access of all data through schema versions.

* This work has been partially supported by Capes and CNPq.
However, the representation of the temporal dimension is essential to keep the whole evolution history. This feature is necessary in many computer applications, as medical control, geographical information systems and flight reservation [2]. The schema versioning with temporal features has been studied extensively in relational environments [2–4]. Bitemporal schema versioning [3] enabling retro and pro-active schema changes to produce past, present and future schema versions has been specially analyzed.

The majority of the proposals found in the literature approached schema versioning through temporal features [5–7] or version control mechanisms [8]. The main contribution of this paper is to present a schema evolution mechanism which integrates features of both versions and time.

The main attempt of this paper is to present a schema versioning mechanism to manage the schema evolution in a temporal object-oriented database. The schema evolution management uses an object-oriented data model that supports temporal features and versions definition - the Temporal Versions Model - TVM [9]. Thus, the extensional data level supports integration with an existing database, allowing the maintenance of conventional and temporal versioned objects. In order to fulfill the schema evolution requirements, the instance propagation approach is also presented. Concerning physical implementation, propagation and conversion functions are specified to assure the correct instance propagation and allow the user to handle all instances consistently in both backward and forward schema versions. The initial requirements concerning data management in a temporal versioned environment, during schema evolution, are also exposed.

This paper is structured as follows. Section 2 briefly presents the Temporal Version Model. Section 3 shows an overview of the schema evolution architecture proposal. In section 4, the schema versioning approach is presented. The instance propagation approach is presented in Section 5. The main requirements concerning data management during schema evolution are exposed in section 6. Section 7 cites some related works. Section 8 summarizes the main ideas of this paper and proposes some topics for future works.

2 Temporal Versions Model - TVM

Temporal Versions Model - TVM [9] is an object-oriented data model supporting time and versions concepts. TVM allows storing object versions, objects lifetime, and keeps all the changes of data dynamic values. Only the main features of TVM are presented here, which are necessary to understand the schema evolution process proposed in this paper.

Time is associated with objects, versions, attributes and relationships. An object has a time line for each of its versions. In this way, TVM supports two different temporal orders: branching time for a versioned object, and linear time within a version. Time variation is discrete, and is represented in the model through intervals as timestamps. These timestamps are bitemporal: valid time
(when a fact becomes true in reality), transaction time (when a new value is posted to the database) and implicit [1].

Class attributes and relationships can be defined as static (past values are not stored in case of modification) or temporal (when all the defined values are stored in the database, composing the history). The user is responsible for classifying attributes and relationships as static or temporal, during the application specification. A class may present attributes and relationships of both kinds.

Figure 1-a shows the state diagram of a version, showing the possible status, as well as the transitions among states, and the events that cause these transitions. Depending on the status, operations can or cannot be applied to them (Figure 1-b).

Fig. 1. State diagram of a version

### 3 An Overview of the Schema Evolution Architecture

This section presents a proposal to manage the schema evolution. Figure 2 shows the architecture proposed to manage the schema versioning mechanism, which is split into six distinct parts:

- **Schema versioning manager** - Implements the schema and object versioning through TVM, the basis on which our proposal is founded.
- **Schema evolution manager** - Controls the schema evolution management (change manager) and accomplishes instance change propagation (propagation manager) in order to assure consistency between the schema versions and their corresponding data.
- **Data Management** - Controls the schema changes, change propagation and data manipulation through the transaction processing. These operations can be handled inside the same transaction.
- **Metadata structure** - Keeps evolutionary information concerning schemas and their corresponding classes, attributes, methods and relationships.
- **Data storage** - A database that stores the user applications and their associated data.
The schema evolution management uses TVM to control not only the schema versioning, but also the storage of extensional database and the propagation of the changes to the objects. Concerning the extensional data level, it supports integration with the existing database, allowing the maintenance of conventional and temporal versioned objects.

The next sections present our proposal on how the requirements explained here can be achieved.

4 The Schema Versioning Approach

This section explains the schema versioning approach using TVM. A metadata structure is defined to store information concerning the evolving schema states, as well as their classes, attributes and relationships. In addition, two alternatives are defined for the database extension management: multi-pool for schema versioning and single-pool for class versioning. Concerning the physical representation, both approaches can be used in the same application.

4.1 Schema Versioning using the Temporal Version Model

In order to complete the requirements to temporal management of applications, the occurrence of schema versions besides data evolution shall also be supported. TVM already presents versions and temporal features on the data level, as presented in the preceding section. This section adds to TVM the support to schema versioning.

Similar to data versions, the schema versions also have states that represent the development phases. Figure 3 (a) illustrates the states that a schema version can go through, and the actions that can cause the transitions. A schema version is initially created in the working status. In this status the version represents a schema that is in a definition phase (all changes cause corrections). Differently from data versions, in this state the schema version does not have time associated to itself, cannot be instantiated nor referred, but can be physically removed.
A schema version in the stable status has the features of a data version in the same status. A schema version in the instantiable state has a data extension associated and new instances can be created. The consolidated status is also similar to consolidated status of data versions. However, a schema version in this stage can be returned to the instantiable state. The frozen status (which corresponds to the logic exclusion) is similar to the deactivated status of data versions, but schema versions in this status can be only used to solve queries. Figure 3 (b) presents a summary of the actions that can be performed in each of the schema version states.

4.2 Meta Schema

A metadata structure was specified in [10], and it is modelled through classes which maintain evolutionary information concerning the schemas, their corresponding classes, methods, attributes and relationships. Figure 4 illustrates the specification of a metadata structure using a UML class diagram.
The metadata is modelled through the following main classes: *versioned schema, schema, class, attribute, relationship* and *method*. The schema versioned class keeps control of information about the schema versioning storage, and schema class stores data concerning the schema versions. Each schema version can have classes that maintain temporal information about their corresponding classes, methods, attributes and relationships. The metadata structure is completely specified in [10]. The specification of class operations must be reported in future papers since it was not necessary in this work.

Since bitemporal database is adopted to model the schema versioning mechanism, all temporal labels must include the following attributes: initial transaction time ($t_{Time_i}$), final transaction time ($t_{Time_f}$), initial valid time ($v_{Time_i}$) and final valid time ($v_{Time_f}$).

### 4.3 Intention and Extension Data Management

The management of temporal schema evolution is done through *schema versioning* and *multiple repositories*. Any schema change leads to a derivation of the new schema version and a new data repository is created. In addition, data of the previous repository is copied to the new repository, updated according to the new schema.

Figure 5 illustrates the inclusion of a new class, named *Subject*, in the first schema version (E1: Schema1). A new schema version is derived (E2: Schema2), whose data are stored in the intention database. A new data repository containing the changes performed is created (Repository,2).

**Fig. 5. Schema Versioning**

**Single Repository Storage.** The schema versioning approach always handles versions of the complete schema, which can cause a high number of version derivations. In this case, the quantity of data repositories can be greatly increased. Another implementation proposal is defined here in order to improve the performance of the system, avoid frequent interruption during data propagation and assure the integrity of stored data.

In this context, a *class derivation mechanism* [11] is proposed. This approach has been applied to adapt the versioning concept in class level. Here, every class
is seen as an independent versionable object. If the class description is changed, a new class version is stored. The propagation of changes is performed in the same data repository in order to decrease data duplication, and all the history of the schema evolution is kept.

Figure 6 illustrates the inclusion of the address attribute in the Student class, in the first schema version (E1: Schema1). A new schema version is derived (E2: Schema2), whose data are stored in the intention database. But, the instance propagation is stored in the already existing data repository (Repository,1).

The meta schema must be implicitly updated in order to keep the performed schema changes. The versioned schema class stores control data about schema versioning. In addition, each schema version stores temporal data in the Class, Attribute, Method, and Relationship classes. On the same way, each versioned class must be associated with a versioned schema, which also stores the versioning class data control.

5 The Instance Propagation Approach

When a schema version is modified, extensional data needs to be adapted to assure consistency between the new schema version and data. This can be accomplished through the following steps: (i) identify the objects which should be updated; and (ii) propagate the changes to the database instances.

There are two important approaches for implementing the instance propagation mechanism [12]: immediate and deferred propagation. Our proposal presents a hybrid method, which allows the use of both approaches, depending on the application necessities. A schema change causes an immediate update in those objects that are being used by applications. That means that only objects that have active valid time must be updated. Therefore, an object can be required by an application, and then updated through the deferred mechanism, following the same rules of immediate propagation. Figure 7 shows the access scope of the schema versions and their corresponding database instances.

Concerning physical implementation, two kinds of function have been defined: propagation and conversion. These functions have been specified to assure
the correct instance propagation and to allow the user to handle all instances consistently in both backward and forward schema versions.

5.1 Propagation Functions

Propagation functions must be specified in order to update the instance value required by schema versions. Two kinds of functions must be defined for each schema structure modification: forward and backward. The former describes the instance behavior when it is required by the new schema version and must be associated with the previous one. In an opposite direction, the later describes the behavior of the instance required by the previous schema version, but being associated with the new schema version. The conversion functions are reversible, which means the instances can be converted freely among several schema versions.

Figure 8 shows the algorithms which specify both propagation functions, forward and backward, applied to the schema modification (add class subject) represented previously in figure 5. The former defines the operations involved in this change: (i) create the object with its values, and (ii) specify the relationship (study) with Student class. The later excludes subject object and study relationship.

5.2 Conversion Functions

Conversion functions must be specified in order to update the instance values required by class versions. Two kinds of functions must be defined for each class structure modification: forward and backward. The former describes the instance behavior when it is required by the new class version and must be associated with the previous one. In an opposite direction, the later describes the behavior
of the instance required by the previous class version, but being associated with the new class version. The conversion functions are reversible, that means the instances can be converted freely among several class versions.

Figure 9 illustrates both conversion functions, forward and backward, applied to class modification (add address attribute in class Student) represented previously in figure 6. Through the former function, the address attribute is excluded. Through the later, address attribute is included and it receives null value.

5.3 Remarks about Propagation and Conversion Functions

When the object is temporal, besides updating the value the transaction and valid time must be recorded. In this case, transaction time must be updated to present time (now). The valid time must be defined by the user, being limited to present and future. The temporal pertinence must be verified, and the time instances must be included in the time of the schema versions.

For further research, we are analyzing the possibility of specifying a process that integrates (merge operation) different schema versions and their associated instances.

6 Data Management during Schema Evolution

The schema evolution management involves not only the schema change and data propagation but also data manipulation operations. For instance, object
handle through *insert*, *update* and *delete* operations could be made while a schema change operation is performed. Many of the conventional techniques employ lock mechanisms to manage schema evolution, restricting the concurrence. Thus, there is a direct relationship between schema evolution and transaction processing.

Since transactions are constructs that change the database state, we are analysing their effects during the schema evolution and the instance propagation approaches. The result will be a mechanism to control schema changes, instance propagation and data manipulation, operations which can be specified inside the same transaction. Then, the first requirements are the following:

- **Transaction specification** - Specification of how operations must be handled inside transactions and definition the effects caused in the TVM environment, considering schema, instances and version derivation.
- **Data manipulation rules** - Specification of rules to assure consistency of data manipulation during schema evolution. The temporal label integrity must be maintained by rules defined in [13], and these rules must be specified according to TVM.
- **Integrity rules** - Specification of rules to assure that the database outcome is always correct and consistent. Otherwise, the transaction must be aborted.
- **Concurrency mechanisms** - Specification of techniques for the performance of transactions processing with interleave operations, including schema change, data propagation and data manipulation.

Since our work in this area is quite recent, more concrete results will be reported in future papers.

### 7 Related Work

The appropriate treatment of performed changes in a schema during its lifetime has been the objective of much research. Consequently, a great number of proposals, which aim at dealing properly with this subject, have emerged.

Historically, three fundamental techniques have been employed for modifying the conceptual structure of an object-oriented database. First, schema evolution [14], in which the database has one logical schema and change operations are applied to the class definition and class hierarchy. Second, schema versioning [8], in which several versions of one logical schema can be created and manipulated independently. Third, class versioning [15], which specifies type through the versions maintenance of each type and instance binds. The Sades Evolution System [16] proposes the use of schema evolution for class versioning and views for conceptual database evolution.

In the temporal database context, [7] defines formally a model that uses concepts of time and version in an object-oriented environment. The model admits only transaction time. It defines axioms in order to ensure the schema consistency, but it does not define rules that guarantee the validity of the changed objects. Another formal model for schema version management is defined in [6].
This research defines an object-oriented data model for schema versioning, which considers valid time and transaction time. [17] proposes the use of a temporal object model for management of schema changes and adaptation of instances. For any process change, the object adaptation is crucial, but only the affected part of the object has to be changed. In [5], a complete collection of schema change operations is proposed. This proposal also uses the bitemporal versioning. The used versioning granularity is the temporal version, which is defined as a union of the schema version with the correspondent instances.

Temporal concepts and versions modelling are mostly treated individually in literature. In contrast to the approaches above our proposal presents a schema evolution management approach, which is based on TVM, thus adding time and version features to the class, schema and instance levels. Finally, the main achievement of this paper in relation to [10] is that TVM is used as the basis for intentional and extensional database, allowing the management of the instance propagation process. Furthermore, we have adopted branched time, which provides more than one active schema version at the same instant of time.

8 Summary and Concluding Remarks

This paper proposes a schema evolution management approach, which is based on the Temporal Versions Model, adding time and version features to the schema, class and instance levels.

The union of version and time concepts keeps different schema versions as well as stores the schema changes performed. It also allows both retroactive and proactive updates and queries in the schema and database. The use of bitemporal schema versioning increases the flexibility of our proposal, allowing access to past, present and future schema versions.

The instance propagation approach is also proposed through the specification of propagation and conversion functions. These functions assure the correct instance propagation and allow the user to handle all instances consistently in both backward and forward schema versions. One interesting feature is that the extensional data level supports integration with existing database, maintaining conventional and temporal versioned objects. Thus, it represents the main achievement of this paper in relation to [10], since the management of the instance propagation process is made through TVM.

Currently we are working on an extension of the present approach in two directions. First, we are specifying a mechanism to control concurrently schema changes, instance change propagation and data manipulation through the transaction processing. Second, a TVM object definition language is being specified as extension of the TVQL [18] in order to fulfill the schema evolution management and data manipulation. TVQL (Temporal Versioned Query Language) is a query language proposed to TVM. As a final result, we intend to propose a complete schema evolution model with temporal and version features. In addition, this schema evolution model will be implemented on top of a commercial database, whose main attempt is to simulate the feasibility of the proposed model.
References

Towards Temporal Information in Workflow Systems

Carlo Combi¹ and Giuseppe Pozzi²

¹ Università di Verona, strada le Grazie 15 I-37134 Verona -Italy-
combi@sci.univr.it,
² Politecnico di Milano, P.za L. da Vinci 32 I-20133 Milano -Italy-
giuseppe.pozzi@polimi.it

Abstract. A workflow management system (WfMS) is a software system that supports the coordinated execution of different simple activities, assigning them to human or automatic executors, to achieve a common goal defined for a business process. Temporal aspects of stored information cannot be neglected and the adoption of a temporal database management system (TDBMS) could benefit. By this paper we scratch the surface of the topic related to the use of a TDBMS in a WfMS, identifying some advantages in managing temporal aspects by a TDBMS inside some of the components of a WfMS. E.g., queries to reconstruct the schema of the business process or to assign activities to executors balancing their workload over time, or the definition of constraints among tasks can benefit from the use of a TDBMS.

1 Introduction

Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities. A worktask (or task) defines some work to be done by a person, by a software system or by both of them. Specifying a workflow involves describing those aspects of its component tasks (and the processing entities that execute them) that are relevant to control and coordinate their execution, as well as the relationships between the tasks themselves.

Information needed to run a workflow are stored by a database management system (DBMS). In most cases, the adopted DBMS is a traditional, off-the-shelf, relational DBMS which does not provide any facility to manage temporal aspects of stored information, forcing the designer to explicitly define and manage those aspects. Temporal information to be managed in a workflow involves several aspects, such as the starting and ending timestamps of a task, the deadline for completing an activity, the validity time of a data item, the time interval between the execution of two different activities, to mention few of them.

With regards to these temporal aspects, the adoption of a temporal DBMS (TDBMS), which is based on a data model where time and time dimensions are suitably represented and managed [12], could easily improve the development of a system devoted to the automatic management of workflows (workflow management system: WfMS). Indeed, the adoption of a temporal data model and of a related temporal query language, could help to:
manage in a homogeneous way information related to different aspects of a
workflow system, providing a uniform data model to all the software com-
ponents of a workflow system;
express in a general way those temporal aspects that are not application-
dependent, thus allowing the focus on specific issues of the application when
designing a workflow;
allow a powerful specification of workflows, where application-dependent
temporal aspects of real world situations can be simply expressed.

This paper provides a first analysis of specific temporal issues of workflow
systems which could be suitably faced by the adoption of a TDBMS; more specif-
cally, we first consider temporalities of information modelled in workflow sys-
tems and then examine the temporal issues related to the management of this
information during the automatic execution of workflows. In this paper we shall
adopt a generic temporal relational data model and the widely known temporal
query language TSQL2 [13], to show how to model and manage different tempo-
ralities in WfMSs. Throughout the paper, we shall mainly consider the two basic
temporal dimensions, namely valid and transaction times, which have been ex-
tensively considered in the temporal database literature [12]; more recent topics
in the temporal database area will be mentioned for specific needs of WfMSs.

In the following, Section 2 provides some basic concepts on workflow systems,
a motivating example, and a description of used models. Section 3 discusses the
workflow models which require temporal dimensions, and describes the compo-
nents of a workflow system that could greatly benefit from the adoption of a
TDBMS, mentioning also relevant related work. Finally, Section 4 sketches out
some conclusions and general remarks.

2   Workflow Management Systems - WfMSs

Workflow Management Systems (WfMSs) are software systems that support the
specification of business processes (described by their process model or schema),
the execution of process instances (cases) described in terms of activities (tasks)
and of dependencies among tasks. A business process may refer to the man-
agement of a car rental company, of a travel agency, of an assurance company
or even a bank loan. WfMSs control process executions by scheduling activities
and by assigning them to executing agents. WfMSs are very complex systems
and include many different components, such as: the interpreter of the process
definition language (PDL) used to formally model the business process; the pro-
cess model designer, which helps the user to suitably define a process model
according to the supported PDL; the resource management unit, to assign tasks
to executing agents; the database connectivity unit, to access data stored into a
DBMS; the transaction manager; and the e-mail feeder.

2.1   A Motivating Example

Throughout the paper, as motivating example of a business process that can be
managed by a WfMS, we shall refer to a car rental company: Figure 1 depicts
the car rental example. Although the semantics of the graphical representation is very intuitive, we refer to the conceptual model described in [5].

A new case is started as the car rental company receives a reservation request. The task GetRentalData collects customer’s data and pick-up and return date and place. Next, ChooseCar specifies the type of car the customer prefers (Ferrari, Cadillac, ...). The task CheckCarAvailability queries the database and verifies whether the specified car is available by defining a value for the variable Available. According to the value of Available observed by routing task R1, the outgoing left arch may be followed, leading to the task RejectReservation which informs the customer about failure in reserving the car: otherwise, if the car is available, the task MakeReservation performs the reservation and informs the customer, while SendConfirmation sends the customer a mail with reservation details.

2.2 Workflow Models

A WfMS stores all the information it needs inside a database managed by a DBMS. Several models can be defined for a WfMS considering the organization where the process is enacted and managed, the formal description of the process model, the data used by the process, the exception model to manage expected
events which may force a deviation from the main flow [6], the transaction model to manage aspects of concurrent access to shared data [5]. In the following, we categorize some of the models we shall refer to, as defined in [5].

**The Organizational Model**
The organizational model formally depicts the organization of the company where the process is being enacted. Information mainly relates to agents, owned skills and hierarchy inside the company: all this information is stored by suitable database tables, whose structure is independent both from the organization and from the application domain. Every agent belongs to a group: a group collects agents either involved in the same project or working in the same geographical area. Every agent has a function (or role) specifying the skills the agents owns and the tasks he/she can be assigned, and a supervisor. A generic description of the organizational model includes several tables: however, for clarity reasons, we shall assume that all the information are stored into one unique table, namely *Agent*, storing the name and the Id of the agent, the e-mail address, the role owned, the group the agent belongs to, and the Id of the manager for that agent. The structure of the table, with some tuples for the *CarRental* process, is the following:

<table>
<thead>
<tr>
<th>AgentName</th>
<th>AgentId</th>
<th>E-mail</th>
<th>Role</th>
<th>Group</th>
<th>ManagedBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hernest</td>
<td>H03</td>
<td>hernest@mailer</td>
<td>PhoneOperator</td>
<td>Boston</td>
<td>B08</td>
</tr>
<tr>
<td>Rudy</td>
<td>R01</td>
<td>rudy@mailer</td>
<td>PhoneOperator</td>
<td>Boston</td>
<td>B08</td>
</tr>
<tr>
<td>Laura</td>
<td>L05</td>
<td>laura@mailer</td>
<td>PhoneOperator</td>
<td>Harvard</td>
<td>B08</td>
</tr>
<tr>
<td>Ken</td>
<td>K13</td>
<td>ken@mailer</td>
<td>PhoneOperator</td>
<td>Harvard</td>
<td>B08</td>
</tr>
<tr>
<td>Doug</td>
<td>D10</td>
<td>douglas@mailer</td>
<td>CarExpert</td>
<td>Syracuse</td>
<td>B08</td>
</tr>
<tr>
<td>Bob</td>
<td>B08</td>
<td>mass_dealer@mailer</td>
<td>LocalDealer</td>
<td>Boston</td>
<td>B01</td>
</tr>
<tr>
<td>Luke</td>
<td>B01</td>
<td>boss@mailer</td>
<td>CEO</td>
<td>Arlington</td>
<td>B01</td>
</tr>
</tbody>
</table>

**The Process Model**
The process model describes the schema of the process; all the information is stored inside suitable database tables whose structure is statically determined and independent from the application domain. At process model design time, the process engineer specifies which is the first task to be executed, the subsequent tasks, the conditions to be evaluated to select different outgoing arcs, the criteria used to assign tasks to agents.

Namely, the tables of the process model are: *Workflow*, storing the names of the process models registered into the WfMS with their respective first task to be activated; *Worktask*, storing the names of the tasks of all the process models and the required role for the executing agent; *RoutingTask*, describing the type of all the routing tasks of any process model; *Next*, storing the successor task for any task of any process model; *AfterFork*, storing the criteria adopted to activate arcs outgoing from a routing task of any process model. Their structures with some tuples related to the process model *CarReservation* are the following:

<table>
<thead>
<tr>
<th>Workflow</th>
<th>SchemaName</th>
<th>StartTask</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarReservation</td>
<td>GetRentalData</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WorkTask</th>
<th>SchemaName</th>
<th>TaskName</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarReservation</td>
<td>GetRentalData</td>
<td>PhoneOperator</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>ChooseCar</td>
<td>CarExpert</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>CheckCarAvailability</td>
<td>InformationSystem</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>RejectReservation</td>
<td>PhoneOperator</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>MakeReservation</td>
<td>PhoneOperator</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>SendConfirmation</td>
<td>InformationSystem</td>
<td></td>
</tr>
</tbody>
</table>
The Information Model

The information model describes the application domain data the process has to manage and the history of different cases.

**Process specific data.** The structure of process-specific tables is defined at schema design time and tables are automatically derived. Even though in most cases several tables are used, for clarity reasons in the following we assume that all the data about the cases of the process model 

| CarReservation | R1 | mutualex_fork |

<table>
<thead>
<tr>
<th>SchemaName</th>
<th>TaskName</th>
<th>NextTask</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarReservation</td>
<td>GetRentalData</td>
<td>ChooseCar</td>
</tr>
<tr>
<td>CarReservation</td>
<td>ChooseCar</td>
<td>CheckCarAvailability</td>
</tr>
<tr>
<td>CarReservation</td>
<td>CheckCarAvailability</td>
<td>R1</td>
</tr>
<tr>
<td>CarReservation</td>
<td>MakeReservation</td>
<td>SendConfirmation</td>
</tr>
<tr>
<td>CarReservation</td>
<td>SendConfirmation</td>
<td>end_flow</td>
</tr>
<tr>
<td>CarReservation</td>
<td>RejectReservation</td>
<td>end_flow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AfterFork</th>
<th>SchemaName</th>
<th>ForkTask</th>
<th>NextTask</th>
<th>Cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarReservation</td>
<td>R1</td>
<td>RejectReservation</td>
<td>Available = no</td>
<td></td>
</tr>
<tr>
<td>CarReservation</td>
<td>R1</td>
<td>MakeReservation</td>
<td>Available = yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CaseHistory</th>
<th>CaseId</th>
<th>SchemaName</th>
<th>Responsible</th>
<th>StartTime</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>CarReservation</td>
<td>H03</td>
<td>01-03-09 10:03:10</td>
<td>8 min 40 sec</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>CarReservation</td>
<td>L05</td>
<td>01-07-11 09:00:10</td>
<td>4 min 49 sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RentalData</th>
<th>CaseId</th>
<th>CName</th>
<th>Car</th>
<th>ReservationDate</th>
<th>RentalDate</th>
<th>ReturnDate</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Marple J.</td>
<td>Cadillac</td>
<td>01-03-09</td>
<td>01-12-01</td>
<td>02-01-06</td>
<td>&quot;yes&quot;</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>Wallace E.</td>
<td>Lamborghini</td>
<td>01-07-11</td>
<td>02-02-01</td>
<td>02-02-02</td>
<td>&quot;no&quot;</td>
<td></td>
</tr>
</tbody>
</table>

**Historical data.** Historical data contains information related to the execution of tasks and cases and is independent from the application domain. A generic description of historical data is obtained by the tables CaseHistory (storing the identifier of all the executed cases, the name of their respective process model, the responsible agent, which in most cases is the starting agent, and temporal information about the start time and the duration of the entire case) and TaskHistory (storing the name of the task, the identifier of the case the task belonged to, the name of the executing agent, the final status of the task and the time of start and end of the task itself). Their structures with some tuples from the process model CarReservation are the following:

<table>
<thead>
<tr>
<th>CaseHistory</th>
<th>CaseId</th>
<th>SchemaName</th>
<th>Responsible</th>
<th>StartTime</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>CarReservation</td>
<td>H03</td>
<td>01-03-09 10:03:10</td>
<td>8 min 40 sec</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>CarReservation</td>
<td>L05</td>
<td>01-07-11 09:00:10</td>
<td>4 min 49 sec</td>
<td></td>
</tr>
</tbody>
</table>
3 Temporal Aspects in WfMS

In this section, we show that several aspects managed by a WfMS have relevant temporal dimensions. We first introduce some examples of temporalities we need to consider when representing and storing information related to the different models of Section 2.2; then, we show how this information should be managed by the WfMS through a TDBMS.

3.1 Temporalities of the Models

Several temporal aspects of data could be managed by a TDBMS, providing a unified support for all the models the WfMS needs. In the following we shall focus on the application to workflow models of those temporal dimensions which are widely recognized as the fundamental temporal dimensions of any fact relevant to a database [12]: valid and transaction times. The valid time (VT) of a fact is the time when the fact is true in the considered domain, while the transaction time of a fact is the time when the fact is current in the database.

**Temporality in the Organizational Model** In defining the organizational model, temporal aspects have to be considered: indeed, we need to represent when agents are available to perform any task suitable for their roles. Let us consider the case of human agents in the CarReservation process: each agent is working only during certain hours of the day and only during some days of the week; furthermore, each agent spends some holidays out of work during the year. All these aspects have to be managed in any case by the WfMS. The adoption of an organizational model which explicitly considers these temporal features can improve the usability of the system by the workflow designer, which can suitably represent the temporal constraints of the organization under consideration. As an example, in the following table we describe when a given agent is available.

<table>
<thead>
<tr>
<th>AgentId</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>H03</td>
<td>working days in [00-06-09 ÷ +∞]</td>
</tr>
<tr>
<td>H01</td>
<td>working days in [00-05-09 ÷ +∞]</td>
</tr>
<tr>
<td>L05</td>
<td>working days in [00-05-07 ÷ +∞]</td>
</tr>
<tr>
<td>L04</td>
<td>working days in [01-01-20 ÷ +∞]</td>
</tr>
<tr>
<td>K15</td>
<td>working days in [01-01-19 ÷ +∞]</td>
</tr>
<tr>
<td>B08</td>
<td>working days in [01-01-11 ÷ +∞]</td>
</tr>
<tr>
<td>B01</td>
<td>all days in [00-01-03 ÷ +∞]</td>
</tr>
</tbody>
</table>

---

**AgentAvailability**

<table>
<thead>
<tr>
<th>AgentId</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>GetRentalData 01-03-09 10:03:10 01-03-09 10:05:50 Completed H03</td>
</tr>
<tr>
<td>47</td>
<td>ChooseCar 01-03-09 10:06:10 01-03-09 10:08:30 Completed D10</td>
</tr>
<tr>
<td>47</td>
<td>CheckCarAvailability 01-03-09 10:08:51 01-03-09 10:08:52 Completed auto</td>
</tr>
<tr>
<td>47</td>
<td>SendConfirmation 01-03-09 10:09:10 01-03-09 10:11:39 Completed R01</td>
</tr>
<tr>
<td>48</td>
<td>GetRentalData 01-07-11 9:00:10 01-07-11 9:01:43 Completed L05</td>
</tr>
<tr>
<td>48</td>
<td>ChooseCar 01-07-11 9:01:45 01-07-11 9:02:01 Completed D10</td>
</tr>
<tr>
<td>48</td>
<td>CheckCarAvailability 01-07-11 9:02:03 01-07-11 9:02:04 Completed auto</td>
</tr>
<tr>
<td>48</td>
<td>RejectReservation 01-07-11 9:02:10 01-07-11 9:03:59 Completed K13</td>
</tr>
</tbody>
</table>

---

18
Temporality in the Process Model

In the previous example we assume we are able to express valid time according to different granularities, suitably defined and managed by the TDBMS. Several works on time granularity have been done within the temporal database community, which should be deeply considered in modelling this kind of temporal data for a WfMS [1, 8, 13, 10].

Versioning. A process can have different versions, which result from the modification of the process schema due to the evolution of the application needs: for example, we can imagine that the schema of the process CarReservation, defined on January 5th, 2000, is that one depicted in Figure 1 until October 12th 2001, when it is modified as depicted in Figure 2. As a difference from the previous version of the schema, if the reservation succeeds, the customer is no longer sent a confirmation: if the reservation fails, the customer is informed and an apologizing mail is sent along with some advertising flyers (task SendApologizes).

Each version of the process model has, thus, a valid time, which identifies the temporal interval during which the given version was used. Tables WorkTask and Next, which are now valid time relations, are suitably updated according to the changes to the schema as follows:
Temporal constraints. A process model includes the definition of temporal constraints among the different tasks and their durations [1]. For instance, the column labelled \( T_{\text{cons}} \) in table Next defines a constraint for the scheduler of the WfMS, specifying that the successor task is to be scheduled, after the current task finished, within the minimum and maximum time stated by \( T_{\text{cons}} \). Similarly, also time constraints for the maximum allowable task duration can be defined for every task, serving as temporal constraints on the execution time. TDBMS should be able to represent temporal constraints even with different temporal granularities, as discussed in [1, 2].

**Temporal in the Information Model** Several different temporal dimensions can be considered for data inside the information model; in this context, valid and transaction times are very important, managing both the history of workflow data and of possible changes on entered data. As an example, the table RentalData, managed by a bitemporal database system [12], allows the WfMS to discover changes both in the customer preferences and in entered data.

The table RentalData presents two situations. In the first one, customer Smith for case 47 modified his reservation. In fact, the customer named Smith on November 12th 2001 made the reservation for a car to be returned on January 6th 2002 (see the first tuple with CaseId=47). The valid time for the case started on November 12th 2001. On November 21st 2001 he changed the return date to January 7th 2002 - thus forcing the same case to be still opened and closing the first transaction for case 47 - and asked for a new car type FIAT.

In the second situation, data originally entered were wrong. Customer Jones did not change anything about his reservation; there has been only an error in data insertion (collection). Indeed, in this case, there is only one tuple about Jones in the current database state (i.e., the state consisting of tuples having the transaction time with +\( \infty \) as upper bound): valid time of this tuple confirms that the reservation of Jones was valid from its definition up to now.

The table RentalData is the following:
As for historical data of the information model, by a TDBMS we are able to manage in a homogeneous way temporal data already present into the database and explicitly managed by the WfMS as user-defined times. Indeed, tables CaseHistory and TaskHistory depicted in Section 2.2 represent the valid time of stored information by attributes StartTime, Duration and StartTime, EndTime, respectively; being these attributes user-defined times [12], the WfMS has to deal with them explicitly. Using a TDBMS, instead, these two tables would be simply valid time relations, directly and homogeneously managed by the temporal database system. Instances of tables CaseHistory and TaskHistory, when represented in a temporal database system as valid time relations, are depicted in the following:

<table>
<thead>
<tr>
<th>CaseId</th>
<th>CaseName</th>
<th>Car</th>
<th>RentDate</th>
<th>RetDate</th>
<th>Avlb</th>
<th>VT</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>Smith J.</td>
<td>Ferrari</td>
<td>01-12-01</td>
<td>02-01-06</td>
<td>yes</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
</tr>
<tr>
<td>48</td>
<td>Jones J.</td>
<td>Lotus</td>
<td>01-12-01</td>
<td>02-01-06</td>
<td>no</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:08:59]</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:08:59]</td>
</tr>
</tbody>
</table>

As for historical data of the information model, by a TDBMS we are able to manage in a homogeneous way temporal data already present into the database and explicitly managed by the WfMS as user-defined times. Indeed, tables CaseHistory and TaskHistory depicted in Section 2.2 represent the valid time of stored information by attributes StartTime, Duration and StartTime, EndTime, respectively; being these attributes user-defined times [12], the WfMS has to deal with them explicitly. Using a TDBMS, instead, these two tables would be simply valid time relations, directly and homogeneously managed by the temporal database system. Instances of tables CaseHistory and TaskHistory, when represented in a temporal database system as valid time relations, are depicted in the following:

<table>
<thead>
<tr>
<th>CaseHistory</th>
<th>CaseId</th>
<th>SchemaName</th>
<th>Responsible</th>
<th>VT</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>CarReservation</td>
<td>HO3</td>
<td>H03</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
</tr>
<tr>
<td>48</td>
<td>CarReservation</td>
<td>LO5</td>
<td>L05</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:08:59]</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:08:59]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TaskHistory</th>
<th>CaseId</th>
<th>TaskName</th>
<th>FinalState</th>
<th>Agent</th>
<th>VT</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>GetRentalData</td>
<td>Completed</td>
<td>HO3</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
<td>[01-11-12 10:03:10 ÷ 01-11-12 10:05:50]</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>ChooseCar</td>
<td>Completed</td>
<td>HO3</td>
<td>[01-11-12 10:06:00 ÷ 01-11-12 10:08:50]</td>
<td>[01-11-12 10:06:00 ÷ 01-11-12 10:08:50]</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>CheckCarAvailability</td>
<td>Completed</td>
<td>auto</td>
<td>[01-11-12 10:08:51 ÷ 01-11-12 10:08:52]</td>
<td>[01-11-12 10:08:51 ÷ 01-11-12 10:08:52]</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>MakeReservation</td>
<td>Completed</td>
<td>HO1</td>
<td>[01-11-12 10:08:53 ÷ 01-11-12 10:08:59]</td>
<td>[01-11-12 10:08:53 ÷ 01-11-12 10:08:59]</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>GetRentalData</td>
<td>Completed</td>
<td>LO5</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:01:43]</td>
<td>[01-08-02 9:00:10 ÷ 01-08-02 9:01:43]</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>ChooseCar</td>
<td>Completed</td>
<td>LO5</td>
<td>[01-08-02 9:01:45 ÷ 01-08-02 9:02:43]</td>
<td>[01-08-02 9:01:45 ÷ 01-08-02 9:02:43]</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>CheckCarAvailability</td>
<td>Completed</td>
<td>auto</td>
<td>[01-08-02 9:02:45 ÷ 01-08-02 9:03:29]</td>
<td>[01-08-02 9:02:45 ÷ 01-08-02 9:03:29]</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>RejectReservation</td>
<td>Completed</td>
<td>L13</td>
<td>[01-08-02 9:03:29 ÷ 01-08-02 9:03:59]</td>
<td>[01-08-02 9:03:29 ÷ 01-08-02 9:03:59]</td>
<td></td>
</tr>
</tbody>
</table>

Observation. To reconstruct the complete history of a case, it could be useful to distinguish at the modelling level both the time during which the task has been executed, i.e., the valid time of the task, and the time at which the task has been assigned to the agent. This latter temporal dimension could be suitably modelled by the concept of event time, which is the occurrence time of a real-world event that either initiates or terminates the validity interval of the modelled fact [7].

3.2 Managing Temporal Aspects

All the components of a WfMS heavily interact with the DBMS to store and reference all the data from the different models (organizational, process and information models). If the adopted DBMS does not feature the management of temporal information, all the components of the WfMS have to explicitly manage those temporal aspects of data: on the other hand, if a TDBMS is adopted, temporal aspects of data are directly managed by the DBMS itself. Without loss of generality, in the following we shall use the widely known temporal query
language TSQL2 [13], to show how the interaction of the WfMS with a TDBMS could be powerfully performed by a temporal query language. In the following, we briefly outline some of the components of a WfMS that could take the greatest advantages from the adoption of a TDBMS.

**Process instancer**: this component defines the proper process model to be adopted for all the running cases and also for completed ones. Obviously, if no change has never been applied to the model of a given process, there is no need to manage temporal aspects related to schema migration of the process model itself; however, a very very small percentage of process models (a good reasonable estimation is less than 1%) does not need for a change during their life, not even for adaptive, perfective or corrective maintenance [4]. On the other hand, if the process model has been changed, one may want to reconstruct the exact process model defined for all the cases, no matter if completed or not. For completed cases, the table CaseHistory defines only the executed tasks, while the table does not allow one to completely reconstruct the schema: in fact, if in front of a routing operator with three or more outgoing arcs one of them only was followed, the table CaseHistory does not specify which outgoing arcs were available but only the followed outgoing arcs. The complete reconstruction of the process model is thus not feasible by the workflow history only. Assuming that a case evolves according to the process model valid at the time of its initiation [4], the temporal query in TSQL2 [13] to reconstruct the process model for any case should sound like:

```
SELECT CaseId, TaskName, NextTask
FROM Next N, CaseHistory C
WHERE C.SchemaName = N.SchemaName AND
  VALID(N) OVERLAPS BEGIN(VALID(C))
```

On the other hand, whenever a new case is started, the process instancer has to define which is the current process model to be followed for the starting case. The TSQL2 query to reconstruct the current process model for CarReservation, if many different versions are defined, should sound like:

```
SELECT TaskName, NextTask
FROM Next N
WHERE N.SchemaName = "CarReservation" AND
  VALID(N) OVERLAPS CURRENT_DATE
```

In both the described situations, the reconstruction of the correct process model would be completed by considering the information about routing tasks stored in the table AfterFork, with the same approach previously described.

**Scheduler of the workflow engine**: this component assigns tasks to agents for execution based on their skills. The component has to mainly consider working time and work load of agents, which are very critical and may vary continuously. **Working time**: the scheduler should not assign to an agent a task whose expected completion, computed as the maximum task duration started from the initiation of that task, will fall into or will include a holiday period for that agent. Additionally, holiday time for agents is in most situations made of a set of intervals, e.g., agent Bob will be on holiday on 02-04-04÷02-04-11 and on
02-05-01±02-05-06. If the task is expected to start on 02-04-02 and the expected
duration is 3 days, the agent will suspend the execution of the task on 02-04-03
(beginning of holiday) and will resume its execution at the end of the holiday
on 02-04-11, resulting in a duration of the execution of that task of approxima-
tively 10 days, much beyond the expectancy of 3 days. The scheduler, thus, has
to compare intervals of holidays of agents and expected execution time of tasks
by suitable techniques, to be able to manage comparisons between intervals at
different levels of granularities and/or with indeterminacy [1, 8]. In this direction,
Bettini and colleagues in [1] study the problem of specifying, even with different
granularities, time constraints for tasks and for delays between tasks; they also
propose a general algorithm for finding free schedules, i.e. schedules which do
not impose further constraints on the task durations, for a workflow.

Work load: the scheduler has to assign tasks to agents balancing the load of
work among them. When evaluating the history of tasks executed by agents, the
scheduler could consider, for example, whether there is some agent which is not
working when some other agent (with the same role) is working. Let us consider
the following TSQL2 query:

```sql
SELECT A.AgentId, TaskName
FROM AgentAvailability A, TaskHistory T, Agent G1, Agent G2
WHERE A.AgentId <> T.Agent AND A.AgentId = G1.AgentId AND
    T.AgentId = G2.AgentId AND G1.Role = G2.Role AND
    NOT EXISTS (SELECT *
    FROM TaskHistory T1
    WHERE T1.Agent=A.AgentId AND VALID(T1) OVERLAPS VALID(T))
```

Through this query, by using the default semantics of TSQL2 in the FROM clause,
tuples of tables AgentAvailability, TaskHistory, and Agent are associated
only if their valid times intersect (in this case, tuples of atemporal tables are
considered as always valid); the condition expressed in the WHERE clause is then
evaluated on these tuples. Valid times of tuples in the result are the intersection
of the tuples considered in the evaluation of the query.

Process model designer tool: the process model engineer uses a tool, namely
workflow designer, to formally define the schema inside the WfMS. The process
engineer has to define for every task an expected task duration (i.e., the average
duration for a task) and the maximum allowable duration (i.e., a deadline for
the completion of the task): if the execution of a task exceeds the maximum
duration, the agent has to be urged to complete the task and if the delay per-
sists, a compensation action could possibly be executed. The tool also has to
enable the engineer in defining temporal constraint (e.g., through suitable val-
ues of the attribute Tcons in table Next of Section 3.1) over dependencies among
tasks, identifying the maximum allowable interval between the completion of a
predecessor task and the activation of its successor(s): these constraints must be
observed by the workflow scheduler.

Exception designer tool: the process model engineer, after having defined
the process model, has to define expected exceptions [6, 11], i.e., those anomalous situations that are part of the semantics of the process and that are known
in advance at workflow design time. Exceptions are an important component of workflow models and permit the representation of behaviors that, although abnormal, occur with high frequency (sometimes even more than 10% of the times). Examples of expected exceptions are the violation of either constraints over data or temporal conditions, a car accident in a car rental process, as well as a change in the RentalDate or in the ReturnDate: if the customer postponed the RentalDate, some compensation actions must be performed, e.g., re-arranging the subsequent rentals of the same car to other customers. Exception management modules in a WfMS are periodically invoked, sometimes with a frequency ranging from tens of minutes to several hours: such a frequency considers that business processes are long-running activities [9] and in most cases there is no need for an immediate management of exceptions. These modules manage exceptional situations that may have occurred during the process enactment since the previous activation of the modules themselves [6] and consider for execution the several exceptions occurred to different cases, grouping them by exception type: e.g., all the task instances that for any reason exceeded their maximum allowable task durations are processed altogether. In order to select all the exceptions to be processed, a temporal query must be performed to select all the exceptions that actually occurred since the last execution of the exception manager: the use of a TDBMS could easily help. Additionally, a TDBMS can help in retrieving values of process specific data at given instants. Let us assume that we need an exception detecting whether the customer switched from a car to another car. The exception mechanism has to compare the currently booked car with the previously booked one. The event of the exception is a modify of an attribute, namely Car: the condition part has to monitor the value of Car with its previous value. By a TDBMS the query to reconstruct the changes to the Car attribute sounds like the following:

```
SELECT SNAPSHOT R1.Car, R2.Car
FROM RentalData R1, RentalData R2
    VALID(R1) BEFORE VALID(R2) AND
    NOT EXISTS (SELECT *
                 FROM RentalData R
                 WHERE R.CaseId = R1.CaseId AND VALID(R) AFTER VALID(R1)
                 AND VALID(R) BEFORE VALID(R2))
```

It can be easily observed that if the adopted DBMS is not a temporal one, the query would result in a more specific expression: indeed, the attributes representing the valid time would be treated as application-dependent ones. Furthermore, the query would be executed without any special support by the DBMS.

4 Conclusions

In this paper, we showed how temporal database systems applied to WfMS could greatly improve performances, e.g. by allowing one to define different versions of the same process model, picking the last one for new cases to be started and
using the original one valid at the time the already running case was started, and ease-of-usage of WfMSs, e.g. by storing the subsequent changes to process-specific data. We also showed how to model and query in a homogeneous way temporal information concerning different models needed by WfMSs.

The management of temporal information in WfMSs is an underestimated research topic, where research results from the temporal database area could be extensively applied and suitably extended. To this regard, we are going to consider the following possible future research directions:

**Design of visual tools for process model design:** the process engineer can define real world temporal constraints among tasks and on task/case durations, possibly by a powerful and easy-to-use tool;

**Exception modelling:** the process engineer can define suitable compensation actions if deadlines and/or temporal constraints are violated. Compensation actions, on the other hand, can be immediately executed, delayed, or temporally grouped, depending on the violated constraints and on the specific process.

**References**

Preserving and Querying Histories of XML-Published Relational Databases

Fusheng Wang and Carlo Zaniolo

Department of Computer Science, University of California, Los Angeles, Los Angeles, CA 90095, USA
{wangfsh, zaniolo}@cs.ucla.edu

Abstract. There is much current interest in publishing and viewing database-resident data as XML documents. In fact, such XML views of the database can be easily visualized on web browsers and processed by web languages, including powerful query languages such as XQuery. As the database is updated, its external XML view also evolves. In this paper, we investigate the problem of representing the evolution history of such a view as yet another XML document, whereby the complete history of the database can also be visualized on web browsers, processed by web languages, and queried using powerful query languages such as XQuery. We investigate various approaches used for publishing relational data, and identify and select those which are best for representing and querying database histories. We show that the selected representations make it easy to formulate in XQuery temporal queries that are difficult to express using SQL on database relations. Finally, we discuss briefly the storage organization that can be used to support these queries efficiently.

1 Introduction

There is a much current interest in publishing database-resident data as (concrete or dynamic) XML documents, which can then be viewed on web browsers, and processed by various web-based applications, including queries written in languages such as XPath and XQuery [4]. As the underlying database is updated, its external XML view also changes (continuously for dynamic documents and at refresh time for concrete ones). Most users who are interested in viewing and querying the current database are also interested in viewing and querying its past snapshots and evolving history—preferably, using the same browsers and query languages. In fact, in many applications, (such as inventory control, supply chain management, surveillance, etc.) changes in the database being monitored are of critical interest. To address this need, web data warehouses have been proposed recently [25]; these detect changes in web sites of interest, preserve their past contents, and answer continuous queries for subscribing users [25]. As in the case of more traditional warehouses, changes can be monitored in two ways:

1. the site publishing the database sends to the web warehouse the log of its recent updates (either continuously or at regular intervals), or
2. the web warehouse downloads from the site frequent snapshots of the XML-published data, and then computes the delta between the new version and the previous one.

The second problem can be reduced to the first one, by computing the delta between the two versions and then deriving an edit script that shows how one version can be transformed into the other; algorithms to support this computation were proposed in [25, 17]. Since we are dealing with XML-published relational data, the order of the tuples is immaterial and we can also use the change detection algorithm for semistructured information proposed in [9]. All these algorithms represent the deltas between the documents as edit scripts and return minimum deltas that will transform the old version into the new one. As discussed in [6], for elements that are logically identified by keys, it is semantically preferable to detect changes between elements denoted by the same key. The X-Diff algorithm proposed in [33] applies in this situation; this algorithm was in fact designed for detecting changes in unordered XML documents with keys, as in the case of our XML-published relational data. By utilizing node signatures and node XHash values, the algorithm tries to find the minimum-cost matching. The algorithm can reach a high matching accuracy, and has complexity \(O(n^2)\) [33].

The additional step of computing the edit script is avoided when the publishing site communicates the changes directly to the web warehouse. Thus in the rest of the paper, we assume that the update log is given. Moreover, we will not go into details about the particular form in which the corresponding updates to the XML document are represented. While somewhat different representations have been used in the past, these differences are not significant in our study, and they are bound to disappear once a standard XML update language will emerge [30]. Moreover, the use of the database update log avoids the temporal indeterminacy problems that instead occur when the remote database is sampled at regular intervals and the edit script is reconstructed using various diff algorithms.

All the approaches previously discussed focus on the preservation and retrieval of past versions of web documents; in this paper, we instead focus on relational tables and discuss how to preserve their content and support complex historical queries via XML and XQuery. Thus, we examine alternative ways to represent the history of XML-published relational tables as XML documents, and show that some of these representations allow the expression of powerful historical queries in a natural fashion. The conceptual and practical interest of this conclusion is underscored by the fact that expressing temporal queries directly on relational databases had instead proven to be a difficult problem that required major extensions to SQL [16, 35, 36, 26]. Thus viewing the history of relational tables as XML documents could provide an appealing venue for supporting historical queries on databases. Observe that the publication of the current database as an XML document is actually not required for representing the database history as an XML document, since this can be constructed directly from the update log of the database.
2 Preserving the History of Documents

Traditional schemes for version management, such as RCS [32] and SCCS [28], are widely used in applications such as software configuration control and support for cooperative work; version-control techniques have also been proposed for databases, often in the context of O-O systems and CAD applications [24]. The emergence of web information systems and many new web-based applications has generated a flurry of interest and research activities, at first focusing on semistructured information [9], and now on XML [13, 25, 14, 6]. This interest is due to the fact that (i) traditional version management applications are now migrating to a web-based environment [3], (ii) there is an increasing realization that e-permanence must be achieved and the broken link problem must be fixed [23], and (iii) very interesting queries can now be answered (using XQuery or XPath) on the preserved history of multiversion documents.

The e-permanence problem has motivated a significant amount of previous work. In particular the Wayback machine crawls the whole web [23], preserving the past content, but without much support for queries (temporal or otherwise). Transaction-time web servers were instead proposed in [18] to archive previous versions of web resources to support transaction timeslice requests by remote browsers. As further enhancement was proposed in [19], where it was shown that the XPath data model and query language can be naturally extended to support transaction time semantics.

The problem of efficiently storing and querying the history of versioned XML documents was discussed in [12, 13, 25, 14]. The reference-based model proposed in [13] unifies the logical representation and the physical one, but can only handle simple queries; in fact, different storage representations are needed for more complex queries [14]. An extension of the SCCS scheme [28] was recently used for representing versions of hierarchically structured documents [6]. Here, we will use a similar version scheme to represent and query the history of XML-published databases at the logical level. Since many different XML-based representations can be used for publishing [29, 30] the same database tables we will also study alternative representations and determines which are most suitable for supporting temporal queries. We will also show that still different representations are needed at the physical level.

3 Publishing Relational Data as XML Documents

Table 1 and 2 describe the history of employees and departments. These transaction-time tables are shown here for illustration and they are not stored in the actual database. Instead, our database only contains the evolving snapshots of these relations—e.g., a single tuple for the employee in the example.

Therefore, we propose to represent and preserve the evolving history of these database relations by means of the XML documents shown in Figure 1 and Figure 2. We will call these H-documents. Each element in a H-document is assigned two attributes tstart and tend, which represent the inclusive time-interval of the
element. The value of \( t_{end} \) can be set to \( now \), to denote the ever-increasing current time.

Our H-documents use a temporally grouped data model [16]. Clifford, et al. [16] show that temporally-grouped models are more natural and powerful than temporarily-ungrouped ones. Temporal groups are however difficult to support in the framework of flat relations and SQL. Thus, many approaches proposed in the past instead timestamp the tuples of relational tables. These approaches incur into several problems, including the coalescing problem [35]. TSQL2’s approach [35] attempts to achieve a compromise between these two [16], and is based on an implicit temporal model, which is not without its own problems [10].

Our model supports temporal grouping by taking advantage of the richer structure of XML documents, and the expressive power of XQuery. An advantage of our approach is that powerful temporal queries can be expressed in XQuery without requiring the introduction of new constructs in the language. We next show how to express temporal projections, snapshot queries, joins and historical queries on employees and departments. These queries were tested with Quip [2] (SoftwarAG’s implementation of XQuery) and can be downloaded from http://wis.cs.ucla.edu/~wangfsh/ecdm02/.

### Table 1. The snapshot history of employees

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Title</th>
<th>Dept</th>
<th>DOB</th>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Engineer</td>
<td>QA</td>
<td>1945-04-09</td>
<td>1995-06-01</td>
<td>1995-09-30</td>
</tr>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Sr Engineer</td>
<td>RD</td>
<td>1945-04-09</td>
<td>1995-10-01</td>
<td>1996-01-31</td>
</tr>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Tech Leader</td>
<td>RD</td>
<td>1945-04-09</td>
<td>1996-02-01</td>
<td>1996-12-31</td>
</tr>
</tbody>
</table>

### Table 2. The snapshot history of departments

<table>
<thead>
<tr>
<th>Name</th>
<th>Manager</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA</td>
<td>Johnson</td>
<td>1994-01-01</td>
<td>1998-12-31</td>
</tr>
<tr>
<td>RD</td>
<td>Joe</td>
<td>1992-01-01</td>
<td>1996-12-31</td>
</tr>
<tr>
<td>RD</td>
<td>Peter</td>
<td>1997-01-01</td>
<td>1998-12-31</td>
</tr>
<tr>
<td>Sales</td>
<td>Frank</td>
<td>1993-01-01</td>
<td>1997-12-31</td>
</tr>
</tbody>
</table>

#### 3.1 Each Table as an XML Document: Columns as Elements

A natural way of publishing relational data is to publish each table as an XML document by converting relational columns into XML elements [29]. Figure 1 shows the history of the table employee and Figure 2 shows the history of the dept table. Thus the history of each relation is published as a separate H-document.
Fig. 1. The history of the employee table is published as employees.xml

Fig. 2. The history of the dept table is published as dept.xml

Based on the published documents, we can specify a variety of queries in XQuery:

QUERY 1: Temporal projection: retrieve the salary history of employee “Bob”:

```xquery
element salary_history{
  for $s in document("employees.xml")/employees/employee
    [name='Bob']/salary
  return $s }
```

QUERY 2: Snapshot queries: retrieve the departments on 1996-01-31:

```xquery
for $d in document("depts.xml")/depts/dept
  [@tstart <= "1996-01-31" and @tend >= "1996-01-31"]
let $n := $d/name[@tstart<="1996-01-31" and @tend="1996-01-31"]
let $m := $d/manager[@tstart<="1996-01-31" and @tend="1996-01-31"]
return( element dept{$n,$m } )
```

QUERY 3: Find employees history from 1995-05-01 to 1996-04-30:

```xquery
for $e in document("employees.xml")/employees/employee
  [tstart='1995-01-01' and tend='1996-12-31']
let $e := $e/
```
for $e$ in document("employees.xml")/employees/employee
let $ol:= overlap($e/@tstart, $e/@tend, "1995-05-01","1996-4-30")
where not (empty($ol))
return ( $e/name, $ol )

Here overlap($v1s, $v1e, $v2s, $v2e) is a user-defined function that returns an element overlap with overlapped interval as attributes (tstart, tend). If there is no overlap, then no element is returned which satisfies the XQuery built-in function empty(). The next query is a containment query:

QUERY 4: Find employee(s) who worked in the “QA” department through the history of that department:
for $d$ in document("depts.xml")/depts/dept[name='QA']
for $e$ in document("employees.xml")/employees/employee[dept='QA']
where $e/@tstart = $d/name/@tstart and $e/@tend = $d/name/@tend
return $e/name

QUERY 5: Find the manager of each employee:
for $e$ in document("employees.xml")/employees/employee
for $d$ in document("depts.xml")/depts/dept[name=$e/dept]
for $m$ in $d/manager
let $ol :=overlap($m/@tstart,$m/@tend,$e/@tstart,$e/@tend)
where not (empty($ol))
return ( $e/name, $m, $ol )

This query will join employees.xml and depts.xml by dept, and the overlap() function will return only managers that overlap with the employee with the overlapped version timestamp intervals.

QUERY 6: Find the history of employees in each dept:
element depts{
for $d$ in document("depts.xml")/depts/dept
return
  element dept { $d/@*, $d/*,
      element employees {
        for $e$ in document("employees.xml")/employees/employee
        where $e/dept = $d/name and
        not(empty(overlap($e/@tstart, $e/@tend, $d/@tstart,$d/@tend)))
        return ( $e/name, $e/dept,
            overlap($e/@tstart, $e/@tend, $d/@tstart,$d/@tend) )
      }
  }
}

This query will join depts and employees document and generate a hierarchical XML document grouped by dept(Figure 5).

3.2 Multiple Tables as a Single XML Document: Flat Structure

Another way to publish relational data is to publish multiple relational tables into a single XML document(Figure 3), but still with the flat structure as shown
Fig. 3. The history of the employee and dept tables is published as company.xml in Figure 2. Essentially there is not much difference between this approach and the previous one.

 Queries on this representation are similar to those described in the last section.

3.3 Multiple Tables as an XML Document: Flat Structure with IDs

To facilitate query processing, when multiple relational tables are published as XML document, tuples can be assigned IDs, which can be referred by IDREF from other elements. For example, in Figure 4, the IDs assigned to dept element, are referred to from employee.

Fig. 4. The history of the employee and dept tables is published as company2.xml

This representation supports queries similar to those discussed in the previous sections, but simplifies joins:

QUERY 7: Retrieve the dept Bob worked on 1995-10-15:

```
return document(“company2.xml”)/company/employees/
```
3.4 Multiple Tables as a Single XML Document: Hierarchical Structure

Another approach is to generate a hierarchical XML document from multiple relational tables (Figure 5). This approach is also taken by XPERANTO [8] through grouping in XML views and SQLX [21] through extended aggregate functions.

```xml
<depts tstart="1991-01-01" tend="1998-12-31">
  <dept tstart="1994-01-01" tend="1998-12-31">
    <name tstart="1994-01-01" tend="1998-12-31">QA</name>
    <manager tstart="1994-01-01" tend="1998-12-31">Johnson</manager>
    <employees tstart="1994-01-01" tend="1998-12-31">
      <employee tstart="1995-01-01" tend="1995-09-30">
        <name tstart="1995-01-01" tend="1995-09-30">Bob</name>
        <salary tstart="1995-01-01" tend="1995-05-31">60000</salary>
        <salary tstart="1995-06-01" tend="1995-09-30">70000</salary>
        <title tstart="1995-01-01" tend="1995-09-30">Engineer</title>
        <DOB tstart="1995-01-01" tend="1995-09-30">1945-04-09</DOB>
      </employee>
    </employees>
  </dept>
  <dept tstart="1991-01-01" tend="1998-12-31">
    <name tstart="1991-01-01" tend="1998-12-31">RD</name>
    <manager tstart="1991-01-01" tend="1996-12-31">Joe</manager>
    <manager tstart="1997-01-01" tend="1998-12-31">Peter</manager>
    <employees tstart="1991-01-01" tend="1998-12-31">
      <employee tstart="1995-10-01" tend="1996-12-31">
        <name tstart="1995-10-01" tend="1996-12-31">Bob</name>
        <salary tstart="1995-10-01" tend="1996-12-31">70000</salary>
        <title tstart="1995-10-01" tend="1996-01-31">Sr Engineer</title>
        <title tstart="1996-02-01" tend="1996-12-31">Tech Leader</title>
        <DOB tstart="1995-10-01" tend="1996-12-31">1945-04-09</DOB>
      </employee>
    </employees>
  </dept>
</depts>
```

Fig. 5. The history of employee and dept is published as depts3.xml

This approach simplifies some queries but complicates others. For example, if we want to retrieve employees in each department (containment query), we can simply write:
QUERY 8: Find employee(s) who worked in the QA department through the dept history:

```xquery
for $d in document("depts3.xml")/depts/dept[name='QA']
let $e := $d/employees/employee
let $e_all := document("depts3.xml")/depts/dept
          /employees/employee[name=$e/name]
where count ($e_all) = 1
    and $e/@tstart = $d/name/@tstart and $e/@tend = $d/name/@tend
return $e/name
```

However, coalescing is needed for other queries in the hierarchical representation.

QUERY 9: Find the salary history of employee 'Bob' in the company:

```xquery
for $s in document("depts3.xml")/depts/dept/
     employees/employee[name='Bob']/salary
return coalesce($s)
```

Here we rely on a user-defined function `coalesce()` (Figure 6) to coalesce the employees. This function can also be defined in standard XQuery, as follows:

```xquery
define function coalesce(xs:AnyType $e) returns xs:AnyType {
  if (count($e) =1) then $e
  else
    if( $e[1]/text() != coalesce(subsequence($e,2)) [1]/text() )
      then ($e[1], coalesce(subsequence($e,2)) )
    else
      if( string($e[1]/@tend) <
            string(coalesce( subsequence($e,2) )[1]/@tstart) )
        then $e
      else ( element {name($e[1])} {$e[1]/@tstart, coalesce( subsequence($e,2)[1]/@tend ),
                                        $e[1]/text()},
                                        subsequence( coalesce( subsequence($e,2) ), 2 ) )
```

Fig. 6. A coalescing function defined in XQuery

3.5 Relational Tables as XML Document: Columns as Attributes

A relational table can also be published as XML document as attributes (Figure 7), e.g., the FOR XML statement in Microsoft SQL Server 2000 [7]. The published XML document is essentially a flat structure that corresponds to the tuple snapshots.

This approach is similar to that of timestamping the whole tuple in the relation. Temporal queries tend to be more complex and most queries require coalescing. Thus, in general, we recommend against this approach when publishing the history of relational tables.

In summary, XML representations that map columns as elements are preferable, and hierarchical representation can only be justified for special cases.
<employees>
  <employee name="Bob" salary="60000" title="Engineer" dept="QA"
    DOB="1945-04-09" tstart="1995-01-01" tend="1995-05-31"/>
  <employee name="Bob" salary="70000" title="Engineer" dept="QA"
    DOB="1945-04-09" tstart="1995-06-01" tend="1995-09-30"/>
  <employee name="Bob" salary="70000" title="Sr Engineer" dept="RD"
    DOB="1945-04-09" tstart="1995-10-01" tend="1996-01-31"/>
  <employee name="Bob" salary="70000" title="Tech Leader" dept="RD"
    DOB="1945-04-09" tstart="1996-02-01" tend="1996-12-31"/>
</employees>

Fig. 7. History of employee published as employees2.xml by mapping the table columns into attributes

4 Efficient Implementation

In the previous sections, we have shown how it is possible to preserve the history of XML-published data as XML documents, and to express complex queries on such documents using XQuery. However, the design of an efficient archival and querying system for such documents present many difficult challenges, due to the need to satisfy multiple competing performance requirements. In fact the design must achieve good performance on

- storage utilization,
- maintaining the archive (i.e. storing the latest changes),
- querying the archive (e.g., to reconstruct past snapshots of a database table, or the salary history of an employee)

For instance, the approach based on the SCSS [28] and recently used in [6] incurs in excessive costs when retrieving a snapshot of a database table—as needed to, e.g., support a query such as ‘find the count of employees in each department on 1999-01-01’. In fact, in the SCSS storage scheme, the successive elements of the snapshot table tend to be scattered uniformly throughout the document history. Thus retrieval of a snapshot normally requires reading the whole document history. When the number of pages in the snapshot grows larger than the number of elements in the document, a temporal index can be used to identify which pages contain elements for a given snapshot. Even so, the number of page reads can be equal to the number of document elements, whereas page reads can be significantly reduced using temporal clustering schemes such as those proposed in [13, 14].

In the archival scheme used in RCS [32], the changes to the document are appended at the end of the current history. Thus the cost of maintaining the archive is minimal with this scheme; the reconstruction of a snapshot, however, can require the traversal of the whole document history. This situation can be greatly improved (at the cost of some additional storage) with the usefulness based clustering approach discussed in [11, 14], which is briefly discussed next.
Usefulness-based Clustering. The usefulness-based clustering scheme (UBCC) [11, 14] clusters the objects of a version into a new page if the percentage of valid objects in a page (i.e., its usefulness) falls below a threshold. When a page’s usefulness falls below a minimum, all the valid records in that page are copied to a new page. Since the records for a given version are clustered, reconstructing the document at a version only requires to retrieve the pages that were useful at that version [15]. The usefulness-based clustering techniques can also play an important role in managing XML-published database histories.

Document Shredding. This technique is often used to manage efficiently XML documents stored in relational databases. Basically, the original XML document is decomposed into pieces that because of their more regular and simpler structure can be efficiently supported with current database engines. Each document piece is identified by an unique ID that then facilitate the reconstruction of the original document through various joins and outer-joins queries [31, 14]. A natural way to shred XML published documents, is to decompose them along the attribute of the original relation—thus, e.g. the history of the employee relation might be shredded into a salary table, a position table, and a department table. No special new ID is here needed, since the relation key or the tuple ID can be used in this role.

Support for Complex Queries Efficient indexing schemes [15] and query processing algorithms [14] can be used to support complex queries on multiversion documents. For instance, multiversion B-Trees (MVBT) [5] indexing is used to support complex queries. The MVBT is a directed graph with multiple roots, and each root is associated with a consecutive version interval.

Finally, while complex operators such as coalesce can be expressed directly in XQuery, much faster execution can be achieved by their direct implementation as built-in primitives.

5 Conclusions

In this paper, we have shown that XML-based representations and query languages provide effective ways for representing and querying the database history. In particular, we have concentrated on a situation where relational data is published using XML: we have shown that the history of the database can be represented as an XML document and queried using XQuery. The resulting XML representation is quite natural, and similar to the temporally grouped data models that were proposed in the past as the most natural approach to dealing with historical data [16]—but one that is difficult to realize in the context of the flat relational data model. In this paper, we studied alternative XML representations and identify those that best support temporal queries. We have shown that XQuery without any modification can express complex temporal queries on such representations. We have briefly discussed the physical representations and indices that are needed to ensure a more efficient execution of these queries.

The historical representations and queries discussed here find applications in data warehouses that archive and collect data from sites of interest to assure the
e-permanence [1] of critical information and support complex queries on changes [25]. Efficient support for archiving warehouse data is already supported in some commercial systems, and various techniques have been proposed for supporting complex queries on such historical data warehouses [34, 27]. Many of the problems that considered in this paper are similar to those that occur in the context of transaction-time web servers and XPath extensions along the transaction time axis [18, 19]. An integration of the web-server and web-warehouse functions on the historical axis is possible and desirable and represents an interesting topic for future investigations.

In this paper, we have focused on how to preserve through XML the change history of the database. But similar representations and queries could, respectively, be used to capture valid-time information in XML documents, and to support temporal queries on such documents. This is a very interesting problem [22] that can be expected to become the focus of much future research. Various techniques developed in the valid-time context [20] can also be effective for dealing with the temporal indeterminacy problems that occurs in warehouses that periodically crawl remote web sites.

References

4. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/
A Lightweight XML Constraint Check and Update Framework

Hong Su, Bintou Kane, Victor Chen, Cuong Diep, De Ming Guan, Jennifer Look, and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609-2280
{suhong|bkane|vchen|cdiep|deguan|jlook|rundenst}@cs.wpi.edu

Abstract. Support for updating XML documents has recently attracted interest. When an XML document is to conform to a given schema, the problem of structural consistency arises during updating, i.e., how to incrementally guarantee that the modified XML document continues to conform to the given XML Schema. To achieve this following the traditional database approach, the XML Schema would first have to be analyzed to construct a structured repository and the XML documents would have to be loaded into this repository before any update could be checked for possible schema constraint violation. Due to the very nature of XML being lightweight and freely shared over the Web, we instead propose a novel approach towards incremental constraint checking that follows the loosely-coupled web paradigm. Namely, we propose to rewrite an XML update query into a safe XML update query by extending the original query with appropriate constraint checking subqueries. This enhanced XML update query can then safely be executed using any existing XQuery engine that supports updates. In order to verify the feasibility of our approach, we have implemented a prototype, SAXE, that implements the above techniques by extending the Kweelt XML query engine by University of Pennsylvania with both XML update support as well as incremental constraint support.

Keywords: XML Update, XQuery, XML Schema, Structural Consistency.

1 Introduction

1.1 Motivation

Change is a fundamental aspect of persistent information and data-centric systems. Information over a period of time often needs to be modified to reflect perhaps a change in the real world, a change in the user’s requirements, mistakes in the initial design or to allow for incremental maintenance.

However, change support for XML in current XML data management systems is only in its infancy. First of all, practically all change support is tightly
tied to the underlying storage system of the XML data. For example, both in IBM DB2 XML Extender [IBM00b] and Oracle 9i XSU [Ora02] who support decomposition of XML data into relational storage or object-relational storage respectively, the user would then need to work with the relational data representing the original XML document as with any other relational data. In particular, any update on the XML data has to be specified using SQL and then will be executed on the underlying relational data. This requires users to be aware of not only the underlying storage system but also the particular mapping chosen between the XML model and the storage model. In other words, there is a lack of abstraction for specifying native updates to XML data independent of the different underlying storage models.

As the first step for native XML update support, a native language for updating XML must be proposed. In this paper, since no World Wide Web Consortium proposal on XML updating has emerged to date, we utilize an extension of XQuery [W3C01b] to support powerful XML updates [TIHW01]. We will further discuss choice of XML update language in Section 2.

An indispensable next step towards supporting updates is to provide a mechanism for maintaining the structural consistency of the XML documents with all associated XML schemata (if any) during the course of the update. Structural consistency is a desired property in database systems since they require that the data must always conform to its schema. An update is considered to be safe only if it will not result in any data violating the associated schema. For example, in a relational database, if an attribute is defined as NOT NULL in the schema, an insertion of a tuple with a NULL value for this attribute will be regarded as an unsafe operation and thus would be refused by the system. Though it is not required that XML documents must always have associated schemata due to their “self-describing” nature, many application domains tend to use some schema specification in either DTD [W3C98] or XML Schema [W3C01a] format to enforce the structure of the XML documents. Whenever XML schemata are associated with the XML data, then structural consistency should also be taken care of during update processing. No work has been done to date to address this issue for native XML.

1.2 Illustrating Example

For example, Figures 1 and 2 show an XML schema juicers.xsd and an XML document juicers.xml conforming to the schema respectively. Suppose the user specifies to remove the cost of the juicer with name “Champion Juicer” (the first juicer in juicers.xml). This operation will render the Champion juicer to no longer have a cost subelement. Such an updated XML document is inconsistent with the schema juicers.xsd since a juicer element is required to have at least one cost subelement, indicated as `<xsd: element ref = cost minOccurs = 1 maxOccurs = unbounded/>` in juicer.xsd. This update would however have been allowed for the second juicer (i.e., Omega Juicer). Some mechanisms must be developed to prevent such violation of structural consistency.
1.3 Desiderata of Preserving Structural Consistency

In our current work, we assume the schema is the first-class citizen. In this sense, an update to an XML document is only allowed when the update is safe, i.e., the updated data would still conform to the given XML schemata. In this section, we discuss the desiderata of the mechanism for checking the safety of XML data updates.

Native XML Support. There have been some techniques proposed for translating constraints in XML to constraints in other data models, say the relational model [KKRSR00] or the object model [BGH00]. Once the mapping is set up, XML constraint checking would be achieved by the constraint enforcement mechanism supported in the other underlying models. However we prefer a native XML support for several reasons. Primarily, we want to avoid the overhead of a load into a database management system (DBMS) as well as the dependency of XML updates on some specific alternate representation.

Loosely-Coupled Constraint Checking Support. Following the traditional database approach shown in Figure 3, the XML Schema would first be analyzed to construct a fixed structure and XML documents could then be loaded into a repository in order to allow updates on the document to be checked. It is preferable to have the validity checking a lightweight standing-alone module rather than being tightly coupled to an XML DBMS. Ideally the constraint checking tool should be a middleware service so that it is general and portable over all XML data management systems.

Incremental Constraint Checking. A naive approach to ensuring the safety of data updates is to do a validation from scratch (shown in Figure 4), namely, to first execute the updates, then run a validating parser\(^1\) on the updated XML document.

---

\(^1\) Most XML document parsers [IBM00a] support validating the XML document against the given DTD or XML Schema.
document, and lastly decide whether to roll back to the original XML document based on the validation result. Such an approach is inefficient since it involves redundant checking on those unchanged XML fragments. It is preferable to have an incremental checking mechanism where only the modified XML fragments rather than the complete XML document are checked. Moreover when the validating parser is run on an XML document modified by a batch of updates and any inconsistency is detected, the parser is unable to tell which updates have caused the inconsistency. Hence, this would make a roll back of only the unsafe updates (but not the changes made by safe updates) impossible.

![Fig. 3. Tightly-Coupled Approach](image)

![Fig. 4. From Scratch Validation Approach](image)

1.4 Our Approach and Contributions

In this paper we introduce a native, incremental and lightweight framework for checking the validity of data updates specified in an XQuery update language Update-XQuery [TIHW01]. The key concept we exploit is the capacity of the XQuery query language to not only query XML data but also XML Schema. This allows us to rewrite Update-XQuery statements by extending them with appropriate XML constraint checking sub-queries.

In summary, we make the following contributions in this work:

1. We identify the issue of preserving structural consistency during the update of XML documents in a loosely-coupled native XML context.
2. We propose a general constraint checking framework that provides native, incremental and lightweight XML constraint checking support.
3. We describe the prototype system SAXE we have implemented. We verified the feasibility of this proposed approach by comparing its performance against that of current state-of-the-art solutions.

2 Related Work

Several XML update languages have been proposed [IBM00b] [Obj99] [SKC+00] [TIHW01]. The expressive power of the language concerns two capabilities: i.e., the power to specify (1) what nodes to update (i.e., querying over the data to select target nodes) and (2) what actions to take on the selected nodes.

[IBM00b] provide their own language for native XML update support in DB2 XML Extender. The expressive power of the update language is limited. XML
Extender allows to specify target nodes in the XML document using XPath expressions [W3C99]. XPath is a subset of XQuery, e.g., in particular, XPath does not support variable bindings. Moreover Extender only allows in-place content update on the selected nodes without other basic support such as inserting new nodes or removing existing nodes. Excelon [Ob99] offers an update language. The disadvantage of this language is that it uses its own proprietary query specification which detracts from its compatibility with the standard XML query language. An XML working group XML:DB [XML02] proposes XUpdate [XUp02] which also has the expressive power limitation in that it uses XPath as the query specification. [TIHW01] stands out among the XML update languages in terms of its expressive power and compatibility with XQuery. It is a natural extension of XQuery that supports the application of a set of update operations including insertion of new data, removal or in-place modification of existing data on bound variables.

None of the above work deals with the problem of incremental validation after the updates. To the best of our knowledge, our earlier work on XEM [SKC+00] is one of the first efforts addressing this problem. XEM proposes a set of update primitives each of which is associated with semantics ensuring the safety of the operation. In XEM, a data update primitive on the other hand is only executed when it passes the validity check. The main limitations of XEM are: (1) the data update primitives in XEM can be only performed on one single element selected by an XPath expressions; (2) XEM is a tightly-couple approach, namely, we implemented an engine on top of PSE (a lightweight object database), mapped the DTD to a fixed schema and loaded the data into object instances. Such a paradigm requires schema evolution support from PSE and specialized constraint enforcement has to be hardcoded into the PSE system.

3 XML Query and Update Language

3.1 XML Query Language: XQuery

XQuery [W3C01b] is an XML query language proposed by W3C. An XQuery statement is composed of several expressions. An important expression in XQuery is the FLWR expression constructed from \texttt{FOR}, \texttt{LET}, \texttt{WHERE} and \texttt{RETURN} clauses.

1. \texttt{FOR} and \texttt{LET} clauses bind values or expressions to one or more variables.
2. \texttt{WHERE} clause (optional) filters the bindings generated by \texttt{FOR} and \texttt{LET} clauses by any specified predicates.
3. \texttt{RETURN} clause constructs an output XML document.

We give an example XQuery over the XML document in Figure 2:

\begin{verbatim}
For $p$ in document(``juicers.xml'')/juicer, $c$ in $p$/cost[1]
Return $c$.
\end{verbatim}

The variable $p$ is bound to iterate over each element node satisfying the expression \texttt{document(``juicers.xml'')/juicer}. For each identified binding of $p$, $c$ is bound to the first \texttt{cost} child node of $p$ and returned.
3.2 XML Update Language: Update-XQuery

[TIHW01] proposes a set of update operations and embeds them into the XQuery language syntax. Each update operation is performed on a target object indicated as target. Table 1 gives the set of update operations and their semantics.

<table>
<thead>
<tr>
<th>Update Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete child</td>
<td>Remove child from children list of target</td>
</tr>
<tr>
<td>Rename child To n</td>
<td>Rename child to name n</td>
</tr>
<tr>
<td>Insert new_attr(n, v)</td>
<td>Insert new attribute with name n and value v to target</td>
</tr>
<tr>
<td>Insert c Before/After child</td>
<td>Insert XML fragment with content of c directly before/after child</td>
</tr>
<tr>
<td>Replace child With new_attr(n, v)</td>
<td>Replace child with attribute with name n and value v</td>
</tr>
<tr>
<td>Replace child With c</td>
<td>Replace child with XML fragment with content c</td>
</tr>
</tbody>
</table>

Table 1. Taxonomy of Update Operations

[TIHW01] extends XQuery’s original FLWR expressions to accommodate the update operations by introducing FOR... LET... WHERE... UPDATE..., i.e., FLWU expressions. We will refer to this extension of XQuery now as Update-XQuery. The BNF of FLWU expression syntax is shown in Figure 5 while the BNF for the UPDATE clause (subOp in Figure 5) in particular is shown in Figure 6.

The semantics of FOR, LET and WHERE clauses are exactly the same as that in a FLWR expression as briefly described in Section 3.1. The UPDATE clause specifies the target node to be updated and a sequence of update operations or FLWU expressions to be applied on the target node.

Figure 7 shows a sample Update-XQuery on the XML document in Figure 2. The variable $p$ is bound to iterate over each element node satisfying the expression document(‘‘juicers.xml’’)/juicer (line 1). For each identified binding of $p$, $c$ is bound to the first cost child nodes of $p$ (line 2) and $p$ is updated by deleting its child node, i.e., the binding of $c$ (line 4).
4 XML Framework For Safe Updates

Our Overall Approach. In order to allow only consistent updates to be processed on XML documents, we aim to develop a loosely-coupled update strategy that supports incremental schema constraint checking that accesses only updated parts of the XML document. The key idea is to first generate a safe Update-XQuery statement from a given input Update-XQuery statement. This generated safe Update-XQuery statement, still conforming to the standard Update-XQuery BNF, can then be safely executed on any XQuery update engine. This way we succeed in separating the concern of constraint check verification from that of developing the XML query and update engine.

For this safe query generation, we design appropriate constraint checking subqueries. The constraint checking subqueries take input parameters from the update operation and determine whether the update operation is valid or not. For this, we exploit the capacity of the XQuery query language to not only be able to query XML data but also XML Schema. This allows us to rewrite Update-XQuery statements by extending them with appropriate XML constraint check sub-queries for each update operation as in Table 1. The execution of an update operation is conditional on passing the constraint checking.

Illustrating Example. For example, Figure 8 shows the rewritten Update-XQuery from the Update-XQuery in Figure 7. There is one update operation in the query, i.e., DELETE $c in line 4. We can see that lines 3, 5 and 6 in Figure 8 have been inserted into this update operation so that this update is only executed when delElePassed(...) (line 5) returns true. delElePassed(...) is a constraint check function which determines the validity of the update DELETE $c. The subquery schemaChkDelEle(...) in line 3 is a function that provides information that is needed by delElePassed(...) to make the determination. We will further discuss the details of these two functions in Section 5.3.

SAXE Architecture. Figure 9 shows the architecture of SAXE3, the framework for generating a safe Update-XQuery statement given an input Update-XQuery. The safe Update-XQuery generator SAXE is composed of the five components described below:

1 FOR $p in document("juicers.xml")/juicer,
2 $c in $p/cost[1]
3 UPDATE $p {
4 DELETE $c
5 }  
6 Fig. 7. Sample Update-XQuery

1 FOR $p in document("juicers.xml")/juicer,
2 $c in $p/cost[1]
3 LET $constraint = schemaChkDelEle("juicers.xsd", "juicer", "cost")
4 UPDATE $p {
5 WHERE delElePassed($c,$p/cost,$constraint)
6 UPDATE $p {
7 DELETE $c
8 }
9 }

Fig. 8. Sample Safe Update-XQuery

2 line 6 is added only to meet the syntax requirement.
3 SAfe Xml Evolution.
1. **Update-XQuery Parser.** The parser takes an Update-XQuery statement and constructs a parse tree representation [ASU86] from it.

2. **Update-XQuery Analyzer.** Given a parse tree, the analyzer identifies more detailed information about types of update operations in the parse tree and derives an enhanced parse tree (refer to Section 5.2).

3. **Constraint Checking Template Library.** We generalize the constraint checking procedures by defining named parameterized XQuery functions called **constraint checking templates**. Each constraint checking template is in charge of checking constraints for one update type.

4. **Update-XQuery Rewriter.** The rewriter handles the actual generation of a safe Update-XQuery. It determines how to rewrite the original Update-XQuery statement by plugging in the appropriate constraint checking functions from the template library and correspondingly modifying the enhanced parse tree.

5. **Update-XQuery Dumper.** The dumper constructs a textual format of the modified Update-XQuery statement from the enhanced parse tree, which now is in the standard Update-XQuery syntax.

---

**5 Components of Constraint Checking Framework**

We now describe the main components of the framework shown in Figure 9.

**5.1 Update-XQuery Parser**

Given an Update-XQuery statement, the Update-XQuery parser constructs a parse tree which is composed of objects of classes that were designed to store the parsed query. For example, a class *Update* is defined to store update clauses. Subclasses of class *Update* are defined for four types of update operations, i.e., *Delete, Rename, Insert* and *Replace*, respectively.
5.2 Safe Update-XQuery Analyzer

Given an internal representation of an Update-XQuery, the analyzer will determine a more specific sub-type of an update operation. For example, the analyzer would examine the content of an object of class *Delete* to classify the update as either deleting an element or deleting an attribute. The detailed information of update types would then be embedded into the original parse tree. We call the new parse tree an *enhanced parse tree*.

5.3 Constraint Checking Template Library

The library stores templates that account for every type of update possible using our Update-XQuery language (See BNF in Figure 5). A constraint check is composed of three steps which are:

1. Query the XML schema to identify any constraints that may be violated by the specified update.
2. Query the XML document to gather information pertaining to the target elements or attributes.
3. Compare the information retrieved from the two previous steps and thus identify whether the constraints would be violated by the update.

We illustrate how this constraint check is done for a delete-element operation. The constraint checking functions *schemaChkDelEle* and *delElePassed* shown in Figures 10 and 11 jointly achieve the three steps mentioned above.

```plaintext
Function integer schemaChkDelEle($xsdName, $parentEleName, $childEleName) Return Integer
1 {  
2 For $pDef In document($xsdName)/xsd:element[@name = $parentEleName],  
3 $cRef In $pDef//xsd:element[@ref = $childEleName]  
4 Let $cRefMinOccurs := $cRef/minOccurs  
5 Return $childRefMinOccurs }
```

**Fig. 10.** Constraint Checking Function *schemaChkDelEle*

```plaintext
Function delElePassed($childBinding, $childBindingPath, $childMinOccurs) Return Boolean
1 {  
2 LET $childInstCount := count($childBindingPath)  
3 Return  
4 If ($childMinOccurs < $childInstCount - 1  
5 Then TRUE  
6 Else FALSE  
7 )
```

**Fig. 11.** Constraint Checking Function *delElePassed*

*The Constraint Checking Function* *schemaChkDelEle* queries over the schema (i.e., step 1) for the information related to the constraints that may be violated
when deleting an element. Deleting an element $e$ of element type $t$ can only violate the constraint of a required minimum occurrence of the elements of type $t$ in the content model of $e$’s parent. $\text{schemaChkDelEle}$ is to retrieve the minimum occurrence of elements of type $\$childEleName$ in the parent type $\$parentEleName$. In particular, line 2 queries the XML schema file, specified by the file name $\$xsdName$, to find the element definition $\$pDef$ for type $\$parentEleName$. The element definition of $\$parentEleName$’s subelement referring to type $\$childEleName$ is stored in $\$childRef$ in line 3. Line 4 then retrieves the minimum occurrence of element type $\$childEleName$ in $\$parentEleName$.

Constraint Checking Function $\text{delElePassed}$ checks whether the data update is safe based on the schema constraint information collected by $\text{schemaChkDelEle}$. $\text{delElePassed}$ is composed of two parts:

1. **Query over data** (i.e., step 2). Line 2 begins querying over the XML document to find the actual count of instances of type $\$childEleName$ that are subelements of the target object. These instances can be retrieved by the XPath expression $\$childBinding$. Function $\text{count}$ on the retrieved instances returns the count of these instances. Thus there would be only $\$childInstCount - 1$ instances of type $\$childEleName$ if the update is allowed to occur.

2. **Integration of query result over schema and data** (i.e., step 3). Line 4 compares the information from the XML schema and data. It compares the minimum occurrence requirement (i.e., $\$childRefMin$) and the actual occurrence if the update were indeed to proceed. In this example, this would be $\$childInstCount - 1$. If actual occurrence after the update had occurred were larger than the minimum occurrence requirement, this check is passed and the update operation is regarded as valid.

### 5.4 Safe Update-XQuery Rewriter

The Safe XQuery Rewriter traverses the enhanced parse tree. For each update operation, based on the update type, the Rewriter determines which template function should be used for checking the constraints of the update. Since the template is parameterized, the Rewriter also instantiates the parameters. Values for these parameters can be identified through the analysis of different parts of the parsed XQuery.

This can be seen in Figure 8. The $\text{delElePassed}$ template function takes in three parameters to execute its query. For this particular example, “$\text{juicer.xsd}$” (the file name of the XML Schema), “$\text{juicer}$” (the type name of the parent element of the to-be-deleted element) and “$\text{cost}$” (the type name of the to-be-deleted element) are the three instantiated parameters respectively.

Once all parameters have been assigned values, the Rewriter needs to insert the instantiated template function into the original query. The Rewriter modifies the parse tree by inserting the constraint checking function for example via a where clause after the associated update clause (line 5 in Figure 8). After all modifications have been done to the original update XQuery, the safe Update-XQuery generation is complete. Finally, a resulting safe Update-XQuery statement is produced by the query rewriter module.
6 The Saxe System

We have designed and implemented the above proposed query rewriting techniques as a prototype system, called Saxe. Our system is based on Kwelt [SD02], a query engine for the Quilt XML query language [CRF02], a precursor of the XQuery standard, developed by the University of Pennsylvania. Kwelt is composed of two parts, i.e., the language parser and language evaluator. The parser takes a Quilt statement and constructs a parse tree. The evaluator then executes the query against the data. First, we extended the Java Compiler Compiler file (JavaCC) which is a Java parser generator in Kwelt so that the Update clauses are accepted by the language parser. Second, we have extended the evaluator so that an Update-XQuery statement can be executed.

We compare Saxe against the from-scratch validation solution, which means we first perform the given update using the extended Kwelt update engine and then we run the modified XML document through an XML-Schema Validator [Tom02] to check for conformance with the given XML schemata. We added the time for regular XQuery execution to the time needed to run the update document through the validator. Generating a typical safe Update-XQuery takes about 400 - 500 milliseconds. For some cases, the safe Update-XQuery execution takes slightly longer than the from-scratch validation solution. However, this does not mean that a safe Update-XQuery is less efficient than using validators after updates are executed. The argument is that the safe Update-XQuery is a one step process where updates are only performed once the updates are deemed safe so that all the attempts for invalid updates will be prevented. On the other hand for the execution of non-safe Update-XQuery, as mentioned in Section 1.3, the system may need to iterate several times between attempting updates and then verifying if the updates leave the XML document in a consistent state.

7 Conclusion

In this paper, we propose a lightweight approach to ensure the structural consistency of XML documents after updates. More precisely, we propose that an Update-XQuery statement can be rewritten into a safe Update-XQuery statement by embedding constraint checking operations into the query. This approach is lightweight in the sense that it can be implemented as a middleware independent of any underlying system for XML data management. To ensure the structural consistency, any Update-XQuery statement is first fed to our safe Update-XQuery generator, Saxe, while the returned safe Update-XQuery statement can then be executed by any system supporting Update-XQuery.

Currently, our safety checking semantics is at the atomic level, i.e., each atomic update on a single object is allowed if this update leads to a valid XML document. As a next step, we want to explore the concept of transactional update, i.e., a batch of updates are only allowed to be executed if the overall effect of executing them leads to a valid document.
References


Change Discovery in Ontology-based Knowledge Management Systems

L. Stojanovic, N. Stojanovic, A. Maedche

FZI - Research Center for Information Technology at the University of Karlsruhe,
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{maedche, Ljiljana.Stojanovic}@fzi.de

Institute AIFB, University of Karlsruhe,
76128 Karlsruhe, Germany
nst@aifb.uni-karlsruhe.de

Abstract: In this paper, we present a novel approach for the change discovery in ontology-based knowledge management systems. It extends our previous work in the ontology evolution by taking into account the usage of an ontology in the knowledge management system. The approach is mainly based on the analysis of the user’s interaction with the system in providing annotations for knowledge resources, as well as in the process of accessing the knowledge by querying the knowledge repository. We defined several assessment criteria to estimate the quality of annotations and the user’s needs from the point of view of the knowledge management. These criteria result in the recommendations for the continual system improvement. Two evaluation studies illustrate the benefits of our approach.

1. Introduction

Knowledge management systems in general are not developed to remain stable, but are subjects to continual change, which is caused by several factors:
− The environment in which KM systems operate can change unpredictably, thereby invalidating the assumptions that were made when the system was built. For example, acquiring a new subsidiary in an enterprise adds new business areas as well as functionalities to the existing system.
− Users’ requirements often change after the system is already built, warranting system’s adaptation. For example, hiring new employees may lead to new competencies and greater diversity within the enterprise, which need to be reflected in the system.
− Some changes in the domain are implicit and can be discovered only through the analysis of user’s interaction with the system. For example, if many users are interested in two topics in conjunction (e.g. debug and java), and there is no knowledge resource matching this criterion, then an efficient knowledge management system should signal that a knowledge resource about the
combination of these topics is needed (e.g. a document on how to perform debugging of java code).

Ad hoc management of the changes in knowledge management systems may work in the short-term, but to avoid unnecessary complexity and failures in the long run, the management has to be interpreted at the conceptual level. In ontology-based knowledge management systems ontologies are used as a conceptual backbone for providing information about knowledge resources and for accessing to the knowledge resources [1]. Therefore, in ontology-based knowledge management systems the changes caused by above-mentioned factors should be applied to the ontology.

The changes can be defined explicitly by the knowledge officer or by the end user. These changes cover business strategy evolution, modification in the application domain, new user’s needs, additional functionality, etc. and they are captured in a variety of ways: direct discussion or interviews, customer specifications, surveys, observations. However, some changes may be discovered by analysing log-files tracking user’s interaction with the system. The application of these changes enables the continual improvement of the knowledge management systems according to the changes in the users’ needs. Although this facility enhances the efficiency of the system, as known to the authors there are no methods and tools that take into account change discovery.

In this paper we present an approach that supports discovery of the changes in ontology-based knowledge management systems. Change discovery is the first phase in the ontology evolution process [2] that enables the timely adaptation of an ontology, as well as the consistent management/propagation of the ontology changes to dependent elements. More details about ontology evolution can be found in [3].

Our primary goal is to suggest the knowledge engineer how to adapt the underlying ontology or annotation in order to enhance the whole system. In that sense we do not only discover the changes, but also estimate their effects on the functionality of the system and choose the most useful one.

The paper is organised as follows: In the second section we discuss the methods to discover changes in ontology-based knowledge management system. Moreover we introduce several assessment criteria to estimate the quality of annotations and the user’s needs from the point of view of knowledge management. Section 3 contains two evaluation studies. After a discussion of related work in the section 4, concluding remarks summarize the importance of the presented approach.

2. Change Discovery in Ontology-based Knowledge Management Systems

Change discovery can be defined as a process of inducing the changes from existing data. In the ontology-based knowledge management system it should consider (i) the ontology as a domain model that underpins that system, (ii) the annotations\(^\text{1}\) that are results of the knowledge providing phase and (iii) the user’s activities in the

\(^{1}\) An annotation consists of a set of ontology instances. We use term metadata as a synonym for an ontology instance.
knowledge management system. Consequently, we have identified the following ways of discovering changes [2]:

- **Structure-driven change discovery** identifies the set of heuristics to improve an ontology based on the analysis of the ontology structure. Based on our experience in the ontology development [1], the most frequently used heuristics are:
  - A concept with a single subconcept should be merged with its subconcept;
  - If all subconcepts have the same property, the property may be moved to the parent concept;
  - If there are more than a dozen subconcepts for a concept, then an additional layer in the concept hierarchy may be necessary;
  - The concept without properties is a candidate for deletion;
  - If a direct parent of a concept can be achieved through a non-direct path, then the direct link should be deleted.

- **Data-driven change discovery** detects the changes that are induced through analysis of existing instances. For example, if no instance of a concept C uses any of the properties defined for C, but only properties inherited from the parent concept, we can make an assumption that C is not necessary;

- **Usage-driven change discovery** takes into account the usage of the ontology in the knowledge management system. For example, by tracking when the concept was last retrieved by a query, it may be possible to discover that some concepts are out of date and should be deleted or updated.

### 2.1 Changes discovered from the annotations

The experience from information retrieval research\(^2\) shows that the “quality” of annotations is crucial for the retrieval of relevant knowledge resources. Our primary objective was not only to monitor the quality of annotations over time, but also to suggest how to adapt them or the underlying ontology, in order to enhance the whole system. Indeed, this type of change discovery is based on the analysis of the quality of the annotations. We define the quality of annotations according to two assessment criteria:

- **Validity** – if metadata in annotation is inconsistent with the domain ontology, then it is not treated in the knowledge sharing;
- **Optimality** - if metadata in the annotation is redundant, inaccurate or incomplete, then it can seriously damage the users’ confidence in the system;

To note that assessment is performed on the annotation level, and the ontology structure is the basis for all measures. From the point of view of the information retrieval, the analysis of the first criterion enables increasing of the recall of the system, whereas the second ensures enhancing of the precision\(^3\).

---

\(^2\) The evolution of the information retrieval systems is mainly based on the improvement in the indexing of documents

\(^3\) For the given request and the system's response, the following numbers can be determined: \(r = \) number of relevant documents in database, \(n = \) number of documents retrieved, \(nr = \) number of relevant documents retrieved. Assuming \(n, r > 0\) than recall = \(nr/r\) and precision = \(nr/n\)
2.1.1 Validity
One annotation is valid only if all its metadata are valid. One piece of metadata is valid if it satisfies all constraints implied by ontology itself. We identified several types of inconsistency: the presence of errors in format, the usage of undefined entities, the usage of unexpected values etc.

Independently of the sources of the inconsistency of the annotation and whether annotations are embedded in the knowledge resources or they are in the knowledge repository [4], they have to keep the consistency with the ontology. The role of the change discovery is not only to find all invalid annotations, but also to make recommendation for changes in the annotations, in order to achieve the syntactic and semantics consistency.

From the knowledge management point of view, the validity of annotations is a pre-request for the successful knowledge retrieval in the highly changeable business environment, since only the annotations that conform to the ontology can be taken into consideration in the knowledge searching process. By considering the knowledge repository in which the content of some resources is annotated with invalid annotations, the knowledge searching process can result not only in the low precision – missing some relevant answers, but also in the incorrectness – the delivery of wrong answers. That is the case with the query “Give me all knowledge resources that describe bonuses of our customers” for the situation presented in Figure 1.

2.1.2 Optimality
In order to emphasise the real need of the optimal annotation, we use examples from the MEDLINE database, which represents the state-of-the-art in human indexing. However, our experiments with MEDLINE show that there are many possibilities to optimise the annotations, which may trigger discovery of changes.

To estimate the optimality of an annotation, we introduce the following three criteria that are important from the knowledge management point of view:

**Compactness** – A semantic annotation is incompact or redundant if it contains more metadata than is needed and desired to express the same “idea”. In order to achieve compactness (and thus to avoid redundancy), the annotation has to comprise the minimal set of the metadata without exceeding what is necessary or useful. The repetition of the metadata or the usage of several metadata with the same meaning only complicate maintenance and decrease the system performance.

Concept hierarchy and property hierarchy from the domain ontology are used to check this criterion. For example, if the knowledge resource is annotated, after all, with the concept Person and its subconcept Female, then this annotation is incompact. When someone searches for all knowledge resources about Person, she searches for the resources about all its subconcepts (including Female) as well. Consequently, she gets this resource (minimum) twice. Moreover, such annotation introduces an ambiguity in the understanding of the content of a knowledge resource, which implies problems in knowledge sharing. Let us examine the meaning of the annotation of a medical document using the set of metadata Person, Female, Aspirin, Complications. Does it mean that the document is about complications in using

---

4 An annotation is not minimal if excluding metadata results in the same retrieval for the same query i.e. precision and recall remain the same.
aspirin only in females, or in all persons? When the second answer is the right one, then this document is also relevant for the treatment of male persons with aspirin. This implies new questions: is the annotation using metadata **Female** an error, or the metadata **Male** is missing? Anyway, there is an ambiguity in annotations, which can be detected and resolved by using our approach.

![Diagram of business policy change](image)

**Figure 1.** Invalid annotation: The concept **Customer** is divided into two subconcepts: **Privileged Customers** and **Others**. Privileged Customers are divided into **Software Customer** (only software is sold to them) and **Hardware Customer** (only hardware is sold to them). In the “old” ontology (left side), the concepts **Software Customer** and **Hardware Customer** inherit the property **bonus** from their parent. In the “new” ontology (right side), the concept **Hardware Customer** changed its parent. However, the new parent does not contain the property **bonus**. As a consequence, the annotation of the knowledge resource with ID 247 that is about the bonus for the customer who buys hardware is incorrect, while in the “new” ontology that corresponds to the new business policy, this type of the customer has no bonus privileges.

In order to prevent this, a knowledge resource should be annotated using as special metadata as possible (i.e. more specialised sub-concepts). In this way, the mentioned ambiguities are avoided. Moreover, the maintenance of the annotations is also alleviated, because the annotation is more concise and only changes linked to the concept **Female** can provoke changes in the annotation.

**Completeness** – An annotation is incomplete if it is possible to extend the annotation only by analysing existing metadata in the annotation, in order to clarify its semantic. It means that the annotation is not finished yet, and requires that some additional metadata have to be filled up.

This criterion is computed based on the structure of the domain ontology. For example, one criterion is the existence of a dependency in the domain ontology.
between the domain entities, which are already used in the annotation. If an annotation contains concepts with many relationships between them (e.g. properties “cures” and “causes” exist between concepts Therapy and Disease), then its interpretation is ambiguous, e.g. are the knowledge resources about how a disease (i) can be cured by a therapy, or (ii) caused by a therapy. In order to constrain the set of possible interpretations, the annotation has to be extended with one of these properties.

This problem is especially important when the knowledge repository contains a lot of resources annotated with the same concepts, because the search for knowledge retrieves irrelevant resources that use certain concepts in a different context. Consequently, the precision of the system is decreased.

**Aggregation** – An annotation is aggregative if it contains a set of metadata that can be replaced with semantically related metadata in order to achieve a shortened annotation, but without producing any retrieval other than the original annotation. For example, this pattern for the annotation refinement occurs when a resource is described with all subconcepts of one concept (e.g. concepts Female and Male are subconcepts of the concept Person). From the searching for knowledge point of view, it is the same whether a resource is annotated using the combination of concepts (e.g. Female and Male) or using only the parent concept (e.g. Person). It is obvious that the second case of annotation makes the management much easier. Moreover, since the standard approaches for the ranking results of querying [5] exploit conceptual hierarchies, for example in a querying for persons a resource annotated using Female and Male will be placed at the same level as a resource annotated using only one of these concepts. It has to be ranked on the top level (level of concept Person), because it covers all subtypes of concept Person.

### 2.2 Changes discovered from the end-users’ activities

The task of a knowledge management system is to deliver the right knowledge in the right moment (at the right place). Interpreted on the level of the searching for knowledge, it means that a user has the opportunity to easily find relevant knowledge resources for the topics, which are important for the problem she solves. In other words, the list of retrieved knowledge resources for a user query should not be empty, and should also contain only highly relevant sources.

It implies that the management component of a knowledge management system should track the interests of users, as well as the list of answers for the posted queries. In order to support this task, the user’s interactions with the knowledge management system have to be recorded. The prerequisite for the meaningful analysis is that this log information is properly organized and interpreted. In order to use as much as possible of the existing mechanism for storage and query, we use the log ontology that is defined in [4]. The role of the log ontology is to model what happens, and why, when, by whom, how it is performed. Each user’s activity is captured in a log file in the form instances of the log ontology.

In order to analyse users’ preferences, we define the rate of interest $IRate(E)$ of users for an ontology entity $E$ as:

$$IRate(E) = IFrequency(E) \cdot Clarity(E) \quad \quad (0 \leq IRate(E) \leq 1)$$ (1)
IFrequency(E) represents the users’ interest in ontology entity E, and it is calculated as a ratio between the numbers of the users’ interactions with the system related to the ontology entity E and the total number of the interactions. Indeed, we use the formula:

$$IFrequency(E) = \frac{Q(E)}{Q} \quad (Q \neq 0, 0 \leq IFrequency(E) \leq 1)$$

whereas Q(E) is the number of queries that contains entity E, and Q is the total number of queries, i.e. Q(E) ≤ Q. In the case that there is no query related to the entity E, the users’ interest in ontology entity E (IFrequency(E)) is 0, whereas when all queries are about entity E, then IFrequency(E) is equal to 1.

The clarity factor represents the uncertainty to determine the user’s interest in a posted query. For example, when a user makes a query using a concept Person, which contains two subconcepts Female and Male, it could be matter of discussion: whether she is interested in the concept Person or in its subconcepts, but she failed to express it in a clear manner. Our experiences show that users who are not familiar with the given ontology used to use a more general concept in searching for knowledge resources, instead of using more specific concepts. In other words, the clarity factor makes the calculation of the users’ interest more sensitive to the structure of the ontology by accounting possible “errors” in the query formulation.

The formula for the clarity factor depends on the entity type:

$$Clarity(E) = \begin{cases} 
  k(E) \cdot \frac{1}{numSubConcepts(E) + 1} & \text{if } E \text{ is a concept} \\
  \frac{1}{numSubProperties(E) + 1} \cdot \frac{1}{numDomains(E)} & \text{if } E \text{ is a property}
\end{cases}$$

0 < Clarity(E) ≤ 1,

whereas numSubConcepts(E) is the number of subconcepts of a concept E, numSubProperties(E) is the number of subproperties of a property E and numDomains(E) is the number of domains defined for the property E.

The coefficient k is introduced in order to favour the frequency of the usage. It is calculated using the following formula:

$$k(E) = \frac{1}{numLevel(E) + 1} \quad (0 < k \leq 1)$$

where numLevel(E) is the depth of the hierarchy of the entity E.

Our primary goal is to decrease the impact of the non-leaf concepts, since they represent the common view to the set of their subconcepts, as described above. The similar strategy is applied to the properties and their hierarchy. However, the unclearness of reasons for a property usage can also arise when multiple domains for a property are defined. Thus, in order to clarify the context of a property usage, we require the explicit specification of the domain of that property, or otherwise we decrease its clarity factor.

The IRate value is calculated for all entities, and two extreme cases are analysed: the frequently used and unused entities. The first extreme corresponds to the entities with the highest rates that should be considered for changes. The formula (1) expresses our experience that the frequent usage of an entity in queries can be a consequence of the bad modelling of the hierarchy of that entity, i.e. in modelling that
entity the hierarchy is not explored in details. For example, in a medical domain the concept **Person** is not split into concepts **Male** and **Female**, although there are a lot of differences between medical treatment of male and female patients. In end effect, any time the user wants to find knowledge resources related to either the male or female patients, she has to make a query with the concept **Person** and consequently the number of retrieved queries is huge. Therefore, our analysis can suggest that the concept **Person** should be divided into several subconcepts. The knowledge engineer decides whether and how to do that. If the considered concept already has a hierarchy, then its suitability (probability) for change is decreased by the clarity factor. The similar strategy is applied to the properties, too.

In the case that nobody is interested in an entity, i.e. the rate of interest for that entity is equal 0, then the entity should be considered for deleting from the ontology and consequently from annotations. However, the problem arises when the knowledge repository contains a lot of resources annotated with that entity. It can be interpreted in various ways. One interpretation might be that the topic is interesting for the community, but it is a new one and it is not used in previous projects. Other interpretation might state that employees are very familiar to this topic and therefore do not search for it.

The previous analysis takes into consideration only one entity. For the simultaneous analysis of the set of entities, the modifications of the frequency of interest $IF_{\text{Frequency}}(E_1,...,E_n)$ is straightforward. However, the calculation of the clarity factor requires further analysis. Due to the lack of space, we mention the most frequently occurring cases:

$$Clarity(C_1, C_2) = \left\{ \begin{array}{ll} \frac{Clarity(C_1) \cdot Clarity(C_2)}{\text{numProperties}(C_1, C_2) + 1} & \text{if } \text{numProperties}(C_1, C_2) \geq 1 \\ 1 & \text{else} \end{array} \right.$$  

$$Clarity(C_1, C_2, ..., C_n) = \prod_{i=1}^{n} Clarity(C_i, C_j)$$

$$Clarity(C, P) = \left\{ \begin{array}{ll} Clarity(C) \cdot Clarity(P) \cdot \text{numDomains}(P) & C \text{ is domain of } P \\ 1 & \text{else} \end{array} \right.$$  

whereas $C$, $C_1$, ..., $C_n$ are concepts and $P$ is a property. Note that the values of all clarity factors are between 0 and 1.

The high value of the rate of interest for the set of entities indicates the following changes:

If two concepts frequently occur in the same queries, it means that the users are very interested in the relationship between them. If such ontology property doesn’t exist, then the system makes the highly ranked recommendation to create it since the clarity factor is set to 1. In case that such property already exists, it is possible that it is too general, i.e. defined for the concepts on the higher level in the concept hierarchy. This is a frequently occurring example of a bad ontology modelling. The recommendation is to specialise this property by creating subproperty whose domain is one of subconcept in the considered concept hierarchy.
Similar to the previous discussion, the frequent occurrence of the ontology entities simultaneously indicates that these concepts are related. However, since the ontology properties are binary, the set of recommendation is delivered. First, a new concept should be created. Second, this concept should be related to each of the n frequently occurring concepts through the newly created property. At the end, the annotations should be extended in order to satisfy the optimality criteria mentioned in the section 2.1.

The high number of queries related to a concept and a property indicates the importance of the concept for that property. Consequently, if the concept is neither a domain nor a range of that property, it should be defined as is recommended by our system. Otherwise, we consider the possibility to specialise the property.

3. Evaluation

In order to prove the validity of our approach, we conducted two case studies, one for each of the proposed strategies for changes discovery: (i) changes discovered from the annotations and (ii) changes discovered from the end-users’ activities.

3.1 Analysis of annotations in MEDLINE

MEDLINE is one of the largest index and abstract databases of medical journal articles, which contains over 11 million references to articles from 4,600 worldwide journals in life sciences. It is maintained by the U.S. National Library of Medicine, which has developed a sophisticated controlled vocabulary called the Medical Subject Headings, used in the indexing of articles. The assignment of MeSH topics to articles, from the MEDLINE database, represents the state-of-the-art in human indexing; the professional indexers who perform this task train for at least 1 year. Ten to twelve topics in the form MainHeading/Qualifier are associated to each article, which can be interpreted as the concept-relation relationship. Although such annotations help in searching for articles, MEDLINE suffers from the overload of information. For example, searching the MEDLINE using the MeSH topic "common cold" yields over 1,400 articles written in the last 30 years. Finding a relevant article might take 20-30 minutes.

In order to prove whether our approach can discover some inconsistencies in MEDLINE annotations, which lead to the decreasing of the precision of the system, we analysed a corpus of MEDLINE articles and corresponding annotations regarding criteria we mentioned in the section 3.2.1. About 200 articles are randomly selected from the MEDLINE database and the results are presented in Table 1.

Discussion:
Validity is perfect - Since the management of MEDLINE annotations is performed very systematically, such inconsistency does not exist in the repository. Indeed, each

5 MeSH (http://www.nlm.nih.gov/pubs/factsheets/mesh.html) is so-called medical ontology
6 The example is taken from http://www.ovid.com
year in November and December MEDLINE is in irregular operation as it makes the transition to a new year of MeSH.

The rate of compactness is small - High-frequent occurrence of this inconsistence can be explained by the format of the annotations itself. Since all metadata in an annotation are assigned separately to the corresponding knowledge resource and are not grouped according to the context, the concept-subconcept pairs occur very often in the annotations (e.g. Human and Female).

Completeness is medium - A part of the problem lies in the format of annotations itself: articles are annotated using topics and not relation metadata [6]. Consequently, it is not possible to express any relationship between medical concepts. Therefore, in lots of annotations the meaning of used topics has to be specified by adding a property, or the range of the property.

Aggregation is high - The small number of cases we found are related to the explanation given for the Compactness.

Table 1. The result of the analysis of the MEDLINE annotations (to note that in some articles two or more inconsistencies were found)

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Validity</th>
<th>Optimality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Compactness</td>
</tr>
<tr>
<td>% of documents where the criterion failed</td>
<td>0</td>
<td>80</td>
</tr>
</tbody>
</table>

3.2 Analysis of users’ queries in a Semantic Portal

The Semantic Portal (SEAL) [5] is an ontology-based application, which provides a “single-click” access to the almost all information related to the organisation, people, researches and projects of our Institute. It is widely used by our research and administrative staff as well as by our students. One of the most usable features is the possibility to search for people, research areas and projects on the semantic basis, i.e. using corresponding Institute Ontology. The portal provides a very user-friendly interface, which enables formation of arbitrary queries using entities from the underlying ontology. The search is performed as an inference through metadata, which is crawled from Portal pages.

Since the installation of the new version of the portal three months ago, the information about users’ activities, regarding querying the portal, are logged in a file. The primary goal was to test the stability of the used version of inference engine. However, we reused the log file in order to evaluate our methods for discovering changes in the ontology. We set up a “what-if” experiment concerning this log file as follows:

1. We rewrote 1000 randomly selected queries under following hypothetical conditions:
   a) The hierarchy of the concept Person that originally had five levels is shorten to only one level including the sub-concepts Researcher and Student;
   b) The hierarchy of the concept Project that originally had two levels is deleted;
c) The hierarchy of the concept ResearchArea is shorten to the first level only. Consequently, we use 20 subconcepts instead of 80 subconcepts in the original hierarchy.

The hypothetical conditions given above are used for query rewriting. For example, from the original query in the form of (Professor, pastProject, Knowledge_Acquisition), meaning that a user is interested in information about professors whose past project was related to the knowledge acquisition, one gets the rewritten query in the form (Researcher, Project, Knowledge_Based_System).

2. We started searching (inferencing) using these queries.
3. We calculated interesting rate IRate (formula 1 in the section 3.2.2) for concepts Person, Researcher, Project and research areas Knowledge_Based_System and E-Commerce. In order to simplify the analysis, for the coefficient k we used the value 1. Table 2 shows the result of our analysis.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Researcher</th>
<th>Project</th>
<th>Knowledge_Based_System</th>
<th>E-Commerce</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRate Value</td>
<td>(202/1000) *(1) = 0,202</td>
<td>(100/1000) *(1) = 0,1</td>
<td>(10/1000) *(1) = 0,01</td>
<td>(2/1000) *(1) = 0,002</td>
<td>(4/1000) *(1/2) = 0,002</td>
</tr>
</tbody>
</table>

Discussion:
We made a hypothetical situation in which the ontology is badly modelled and some hierarchies are not explored at all. A user can select only some restricted, higher-level concepts and for each specialisation he or she has to use one of higher-level concepts (e.g. for the query about professors she has to use the concept Researcher). In such a way we modelled the situation in which the underlying ontology did not correspond to the users’ needs. The task of our method was to recognize which of badly modelled hierarchies do not reflect users’ needs. We discuss several results:

- **The concept Researcher has the highest IRate** - it should be considered firstly - This is the right decision while a lot of queries contain concept Researcher and it has no hierarchy in the hypothetical situation. It means that we could conclude that concept Researcher is used as a replacement for the users’ need to search for some specializations of researchers.

- **The concept Knowledge_Based_System should be considered before the concept E-Commerce** - In our experiment the both hierarchies are shorten. However, in the original ontology the first one was larger and therefore should be firstly considered for a change. The number of queries, which contain topic “knowledge-based system”, reflects users’ needs for more specialized areas of the knowledge-based system.

- **The concept Person has the lowest IRate** - This is the right estimation, since the concept Person has one level of the hierarchy, which satisfies users’ needs regarding this concept.
4. Related work

This section gives an overview of the researches related to our approach. In [7] authors discuss the possibilities to combine the two research areas Semantic Web and Web Mining. The idea is to improve, on the one hand, the results of Web Mining by exploiting the new semantic structures in the Web, and to make the use of the Web Mining to, on the other hand, for building up the Semantic Web. Our paper shows the benefits of the both of methods. We define semantics for the log information in the form of the log ontology and we use the log information to enhance ontologies that are the underlying model for the Semantic Web.

Change mining in the context of decision tree classification for real-life application is studied in [8]. Their primary goal is to know what is changing and how it has changed in order to provide the right products and services to suit the changing market needs. In contrast to this approach, our system discovers changes for the users’ interactions with the systems with the goal not only to predict changes but to improve the efficiency of the system.

[9] sets up a framework for stability of conceptual schemas and proceeds to develop a set of metrics from it. The metrics are based on straightforward measurements of conceptual features. A set of measures we defined takes into account the conceptual structure of the ontology, quality of the annotations as well as the users’ needs.

In [10] the author presents the guiding principles for building consistent and principled ontologies in order to alleviate their creation, the usage and the maintenance in the distributed environments. They define set of the operational guidelines based on the structure of the ontology. In our approach it corresponds to the set of heuristics for the structure-driven change discovery. Moreover, we go a step further by incorporating the data-driven and the usage-driven change discovery.

The area of maintaining knowledge management systems is rather seldom explored in the research community, although the practical importance is elsewhere announced [11]. Our approach enables a systematic analysis of changes in the user's needs and dynamic adaptation of the system to these changes.

5. Conclusion

In this paper, we present a novel approach for the change discovery in ontology-based knowledge management systems. The approach is based on the analysis of the user’s interaction with the system in providing annotations for knowledge resources, as well as in the process of accessing the knowledge by querying the knowledge repository. Our previous work in ontology evolution is used as a basis for this research. We defined several assessment criteria to estimate the quality of annotations and the user’s needs from the point of view of knowledge management. These criteria result in the recommendations for the continual system improvement. The benefits of the proposed approach are manifold: dynamic adaptation of the system to the changes in the business environment, dynamic analysis of the user’s needs and the usefulness
of particular knowledge resources and the organisation of the knowledge repository to fulfil these needs, to name but a few.

The evaluation experiments show that our approach can be applied in the real-world applications successfully. We find that it represents a very important step in the achievement of a self-adaptive knowledge management system, which can discover some changes from the user’s interactions with the system automatically and evolves its structure correspondingly.

References


An Architecture for Managing Database Evolution *

Eladio Domínguez, Jorge Lloret, María A. Zapata

e-mail: {noesis,jlloret,mazapata}@posta.unizar.es

Abstract. This paper presents an architecture for managing database evolution when all the components of the database (conceptual schema, logical schema and extension) are available. The strategy of evolution in which our architecture is based is that of ‘forward database maintenance’, that is, changes are applied to the conceptual schema and propagated automatically down to the logical schema and to the extension. In order to put into practice this strategy, each component of a database is seen under this architecture as the information base of an information system. Furthermore, a translation information system is considered in order to manage the translation of conceptual elements into logical schema elements. A current Oracle implementation of this architecture is also presented.

Keywords: Information Systems, Database Evolution, Forward Database Maintenance, Meta-modelling.

1 Introduction

The requirements of a database do not remain constant during its life time and therefore the database has to evolve in order to fulfil the new requirements. In general, database evolution activities are considered of great practical importance since they normally consume a large amount of resources [10]. As a consequence, much research has been focused on analyzing ways of facilitating this type of activity [1, 16].

Several problems related with databases evolution have been outlined in [6]. In particular, we are interested in the forward database maintenance problem (‘redesign problem’ according to [16]), that is, how to reflect in the logical schema and in the extension changes that have occurred in the conceptual schema of a database. Although a lot of research papers have been written in relation with this problem (see, for example, [16] and [10]) no completely satisfactory solution has been proposed.

As a contribution towards achieving a more satisfactory solution, in this paper we propose an architecture for managing database evolution within the context of forward engineering. The main difference with respect to other proposals

* This work has been partially supported by DGES, projects TIC2000-1368-C03-01 and PB-96-0098-C04-01, and by University of Zaragoza, project UZ-00-TEC-04.
is that we consider a translation component besides the conceptual, logical and extension components. The translation component stores information about the way in which a concrete conceptual database schema is translated into a logical schema. This component plays an important role in enabling the automatic propagation of the conceptual database schema evolution down to the logical database schema making it possible to reflect in the extension of the database the changes performed in its conceptual schema.

Another important difference with respect to other authors [16, 8] is that a meta-modelling approach [3, 11] has been followed for the definition of the architecture. We have chosen this approach because it allows modelling knowledge to be represented and because it has been proven that it facilitates the definition of data model translations [11]. Within the architecture, three meta-models are considered which capture, respectively, the conceptual, logical and translation modelling knowledge. At the same time, the notion of information system, such as is defined in [5], is brought into play not only at the model level (which is the way in which it is normally used) but also at the meta-model level (like in [10]).

So as to show a concrete application of our architecture, we present a current Oracle implementation, which follows the common approach [16, 15] of modelling the conceptual (logical) schema of the database by using the ER (relational) model. It should be noted that we have chosen these concrete models only with the aim of illustrating our architecture. However, the architecture is of general applicability and therefore can also be applied to other approaches such as, for example, within the context of object oriented databases [1, 2]. Like other authors [10], we have chosen to represent the meta-schemas by means of UML class diagrams [13] in the belief that they will be easily understood thanks to the fact that UML is an increasingly widely accepted standard modelling language.

The remainder of the paper is organized as follows. Section 2 explains our view of dealing with database evolution, presenting in Section 3 the architecture we propose. Section 4 is devoted to showing a current Oracle implementation of our architecture. In Section 5 we discuss related work and finally conclusions and future work are outlined in Section 6.

2 The Database Evolution Issue

In order to design a database, users’ information requirements are represented by means of a conceptual database schema $S_C$ (for example, an ER model or a UML diagram). This schema is translated into a logical database schema $S_L$ (see Figure 1) which will be implemented by means of a DBMS. The database is then populated to create a consistent database state $\sigma$.

Within this framework, the database evolution issue can be stated in a general way rephrasing the ideas explained in [16] as follows. Due to varied reasons (changes in the real world [16], optimization procedures for improving the performance of the system [7],..), the conceptual schema $S_C$ is modified generating a new conceptual schema $S'_C$. Ideally this modification at the conceptual level should be managed following the strategy of ‘forward database maintenance’ [6]
Fig. 1. Database Evolution

Fig. 2. Examples of Conceptual and Logical Database Schemas

according to which changes at the conceptual level should be automatically propagated down to the logical schema and its population. That is, the logical schema $S_L$ has to be modified in order to generate a new logical representation $S_L'$ and the database state $\sigma$ has to be mapped into a new database state $\sigma'$ consistent with $S_L'$.

The changes to be performed in the conceptual database schema in order to carry out the desired evolution can be expressed by means of schema transformations [7]. A schema transformation accomplishes modifications in the structure of the database and maps the population of the source schema into an allowable population of the resultant schema [2, 7].

In order to illustrate an example of database evolution, we will use the schema of Figure 2(a) as the conceptual schema $S_C$, which has been obtained combining different examples included in [9]. In this example the E/R model has been used as the conceptual modelling technique and, as usual, entity types, relationship types and attributes are represented, respectively, by rectangles, diamonds and ovals. This schema represents a company where it is perceived that there are employees and projects. The employees can be managers (managing projects) or administrative staff (working for projects or auditing projects).

Traditionally the relational model has been used as the logical modelling technique when the E/R model is used at the conceptual level. Following this criteria, the proposed example of E/R schema has been translated into a relational model (see Figure 2(b)) using the algorithm proposed in [4]. In Figure 2(b) the foreign key constraints have been represented by means of arrows. With regard to this example we only want to emphasize that the n–n relationship type
manages has been translated into a relational table which is not the case with the 1–n relationship types works_for and audits.

In the course of this paper we are going to consider two examples of evolution of the conceptual schema of Figure 2(a). One is the case in which the audits relationship type, together with its instances, must be deleted. This example seems to be very simple but we will see later on that it is more complex than it appears. Moreover, it will serve to illustrate the suitability of the translation component we include in our proposed architecture. As for the other example, we are going to suppose that the attribute department of the entity type employee must be transformed into an entity type. This transformation (1) adds to the conceptual schema an entity type department described by means of two attributes (id_department and department), id_department being its primary key, (2) adds a relationship type employee has department and, (3) deletes the attribute department. With respect to the extension, this transformation maps each distinct non–null value of the attribute department in the old schema into a distinct ‘department’ entity in the new schema. Furthermore, the corresponding ‘employee has department’ relationships are added.

3 Database Evolution Architecture

The architecture we propose aims at providing a general framework which makes it feasible to manage database evolution following a forward maintenance strategy. Therefore, the architecture has to be defined in such a way that the changes performed in the conceptual database schema can be reflected in the logical schema and its extension. It is more or less obvious that some component has to allow conceptual, logical and extensional information to be stored. Let us go on to illustrate, by means of an example, the necessity of also storing knowledge with regard to the translation process from the conceptual into the logical schema.

Let us suppose that the audits relationship type must be deleted from the conceptual schema of Figure 2(a). This modification must be automatically reflected in the logical schema of Figure 2(b) deleting some element. According to the translation algorithm applied to the conceptual schema, it is known that the audits relationship type has been translated into an attribute of the project table, this attribute being a foreign key referencing the administrative table. The problem is that this table contains two columns (id_administrative_1 and id_administrative_2) verifying these conditions. If there is no information about the specific process followed for obtaining the logical schema of Figure 2(b), it is not known which attribute should be deleted. Our proposal is to store knowledge about the translation process explicitly in a component of our architecture, in the same way that the conceptual, logical and extensional information is stored. This component will include, for example, information about the column which the relationship type audits has been translated into.

A meta–modelling approach has been followed for the definition of the components storing conceptual, translation and logical information (a modelling ap-
The meta-modelling approach consists of representing modelling knowledge by means of a meta-model, where a meta-model is a conceptual schema of the elements constituting a data model or technique [3]. Following this approach the elements of a conceptual database schema, logical database schema or translation process are seen as instances of the corresponding meta-model. In order to capture this fact, [10] inspired us with the idea of bringing into play the notion of information system, such as is defined in [5], not only at the model level (which is the way in which it is normally used) but also at the meta-model level.

According to [5] an information system (see Figure 3) consists of three components: an information schema, an information base and an information processor. The information schema defines all the knowledge relevant to the system, the information base describes the specific objects perceived in the Universe of Discourse, and the information processor receives messages reporting the occurrence of events in the environment. In order to respond to the events received, the information processor can send structural events towards the information base and/or towards the information schema and can generate internal events that inform other information systems of the changes performed in it.

The notion of information system is used within our architecture giving rise to four information systems which are used to store, respectively, the conceptual modelling knowledge, the translation process, the logical modelling knowledge and the extension. The corresponding components of each one of these information systems as well as the way in which they are related appear in Figure 4. The name of each one of these components has been modified in an attempt to capture the type of knowledge that they store (in any case the graphical symbol that surrounds each component stands for the type of component it represents).

It must be noted that three different abstraction levels are involved in the architecture. On the one hand, the information schemas of the three former information systems are situated at the most abstract level (metamodel layer according to [13]) and, on the other hand, the information base which stores the population of the database is situated at the least abstract level (user data layer [13]). All the other elements are situated at the model layer [13]. Let us now explain each one of the four information systems of the architecture.

\[\text{Fig. 3. Components of an Information System}\]

---

1 In fact, in [5] this component is called ‘conceptual schema’. However, with the aim of avoiding misunderstandings, we have considered it inappropriate to use the term ‘conceptual’ since we are going to use this component not only at the conceptual level of the database but also at the logical and physical levels.
3.1 Conceptual Information System

The population of this information system (that is, its information base) represents the constituent elements of a given conceptual database schema. For example, with respect to the E/R schema of Figure 2(a), facts such as that employee is an entity type and that department is an attribute have to be stored. These facts will be stored following the structure established in the conceptual meta–schema.

The conceptual information system has to react to external events received from the environment. The type of events in which we are interested are those related with database evolution. Each event issued from the environment is handled by the information processor which checks its validity according to the restrictions imposed by the conceptual meta–schema. If it is valid, the information processor induces a collection of structural events necessary to change the conceptual database schema (that is, the information base) according to the semantics of the received event. The conceptual information processor also generates a collection of internal events that inform the translation information system of the changes performed in the conceptual database schema.

3.2 Translation Information System

The goal of this information system is to store all the information necessary to enable any change performed in the conceptual database schema to be automatically reflected in the logical schema. This goal is achieved by storing the way in which conceptual schema elements are translated into logical schema elements. In order to do this, the translation accomplished by the chosen translation algorithm is specified as a set of elementary translations each of which represents the translation of only one conceptual element into a logical element. For example, the translation of the E/R schema of Figure 2(a) following the algorithm proposed in [4] performs, among others, the elementary translations of transforming the entity type employee into a table with the name employee, and transforming the attribute department into a column with the same name. The elementary
translations are stored in the translation base (that is, the information base) specifying the type of translation, the element of the conceptual schema it is applied to and the element of the logical schema it gives place to.

The translation information system has to react to the internal events generated by the conceptual processor which inform it about the changes performed in the conceptual database schema. In accordance with these events, the information processor determines the elementary translations that must be added, deleted or updated from the translation base. After these changes, the translation base contains the set of elementary translations that translates the new conceptual schema (resulting from the evolution process) into a logical one. The information processor also generates a collection of internal events that inform the logical information system of the changes performed in the translation base.

It must be noted that the new set of elementary translations is determined without it being necessary to apply once again the translation algorithm from scratch. The knowledge stored in the translation information system avoids having to recalculate the logical elements that result from the conceptual elements that have not been modified. The idea of using an information system to store the elements related with the translation process and the way in which this knowledge is used during the database evolution process are, from our point of view, a significant contribution of our work.

3.3 Logical and Extensional Information Systems

The logical information base stores the elements of the logical schema obtained as a translation of a given conceptual database schema. For example, with respect to the relational schema of Figure 2(b), facts such as that employee is a table and that department is a column of it have to be stored. These facts will be stored following the structure established in the logical meta-schema.

The logical information processor receives a collection of internal events from the translation information processor, according to which it generates the structural events necessary to change the logical database schema in order to reflect the evolution performed at the conceptual level.

In Figure 4, the information base of the logical information system is surrounded with both the information base and information schema symbols. This is because the logical database schema can be seen as the information base of the logical information system (as we have explained) or as the information schema of the extensional information system. For this reason two different components of our architecture store the same information. This fact obliges us to define some rules, called correspondence rules (in the same sense as in [10]). These rules govern the correspondence between the elements of each one of the two components. In order to hold these rules, the logical information processor sends the internal events reporting the changes made in the logical database schema, and the extensional information processor, according to the received events, changes the database extension and the database schema (that is, the information schema). This is the only case in which the information processor changes the information schema of its information system.
4 An Implementation of our Architecture

In this section, we describe a concrete implementation of our architecture. This implementation is based on the RDBMS Oracle 8i Release 8.1.7 and the programming language PL/SQL Release 8.1.7 [17]. Within this implementation, the E/R technique has been chosen as the conceptual modelling technique and the relational as the logical modelling technique. We will use UML meta-schemas in order to conceptually describe the structure of the three information schemas that belong to the meta-model level. The graphical representation of these meta-schemas appears in Figure 5 in which only the name is included for each class (its attribute compartment does not appear).

The meta-schema of the conceptual information system (see Figure 5(a)) conceptualizes the different modelling elements of the E/R model (the version we use is based on the E/R model proposed in [9]). For example, it is represented that a relationship type is related with exactly two participants and that an entity type is related with the attributes that conform to its primary key (PK).

![Figure 5. Meta-models of the Information Systems](image)

This meta-schema also has several meta-constraints associated to it. For example, two constraints (expressed by means of OCL [13]) with context EntityType are the following:

1. If an entity type is not a subtype of any entity type, then it must have one primary key.
   \[
   \text{SubtypeOf} \to \text{size} = 0 \implies \text{HasPK} \to \text{size} \geq 1
   \]

2. An entity type cannot be a subtype of itself.
   \[
   \text{SubtypeOf} \to \forall (e \mid e \not< \text{self})
   \]

In order to carry out database evolution tasks the external event types that we have implemented appear in Table 1 (their arguments have been omitted). Six of these external events are basic operations of addition or deletion of modelling elements (entity type, attribute or relationship type), two of them allow the modification of a primary key, one adds an entity type as a specialization of another entity type and, finally, one transforms an attribute into an entity type.
We are aware that this is a relatively small number of external events, and that more are needed for facilitating the database evolution.

As an illustration of database evolution, the example of schema evolution described at the end of Section 2, according to which the attribute department is transformed into an entity type, will be performed using the external event:

\[ \text{AttributeToEntityType('employee.department')} \]

This event has only one parameter which expresses the attribute that has to be transformed into an entity type. As a consequence of this event the following tasks have to be performed: (1) create a new entity type called department with two attributes (id\_department and department) and primary key id\_department, (2) create a new relationship type, called has, between employee and department, and (3) delete the attribute department of the entity type employee.

The information processor of the conceptual information system has been implemented as a set of PL/SQL procedures, one for each of the established external event types. For example, there exists a procedure that is executed when the event AttributeToEntityType occurs and this procedure accomplishes the tasks associated with the event.

The meta–schema of Figure 5(b) represents that the translation information system stores the elementary translations that have to be applied to the given conceptual database schema in order to be translated into a logical schema (the elementary translations are determined following the translation algorithm proposed in [4]). Each translation is related with the conceptual element to which it is applied and with the logical element to which it gives rise.

The information processor of this information system is implemented as a collection of PL/SQL triggers which are fired by the insert, delete or update operations performed in the conceptual database. For example, the addition of the entity type department fires a trigger which adds to the translation base an elementary translation that translates the entity type department into a table. The deletion of the attribute department fires a trigger which deletes the elementary translation that translates the attribute department into a column.

The meta–schema of the logical information system (see Figure 5(c)) conceptualizes the different elements that conform to a relational model. The information processor of this information system is also implemented as a collection of PL/SQL triggers which are fired by the insert, delete or update operations performed in the translation base. For example, the addition of the elementary translation that translates the entity type department into a table fires a trigger which adds the table department. In the same way, the deletion of the elementary translation that translates the attribute department into a column fires a trigger which deletes the column department.
1 CREATE TABLE department(department varchar2(30), id_department integer);
2 INSERT INTO department (department) SELECT DISTINCT department 
   FROM employee WHERE department IS NOT NULL;
3 execute giveidvalues('department','id_department','department');
4 ALTER TABLE department ADD (PRIMARY KEY (id_department));
5 ALTER TABLE employee ADD id_department integer;
6 ALTER TABLE employee ADD (CONSTRAINT restr22 FOREIGN KEY 
   (id_department) REFERENCES department(id_department));
7 execute matchvalues('employee','id_department','department',
   'department','id_department','department');
8 ALTER TABLE employee DROP COLUMN department;

<table>
<thead>
<tr>
<th>Table 2. SQL sentences automatically generated and executed</th>
</tr>
</thead>
</table>

The extensional information system stores the Oracle 8i database schema and its data. The information processor of this information system is also implemented as a collection of PL/SQL triggers which are fired by the insert, delete or update operations performed in the logical database. These triggers automatically generate and execute the SQL sentences that perform the changes that have to be made in the Oracle 8i database in order to accomplish the correspondence rules and to reach a consistent database state. For example, the SQL sentences generated in order to transform the attribute department into an entity type appear in Table 2. These sentences perform the following tasks:

1. Create the new table department with the values of attribute id_department created by means of the procedure giveidvalues (lines 1–4).
2. Create the relational structures corresponding to the new relationship type between entity types employee and department (lines 5–6).
3. Assign values to attribute employee.id_department using the procedure matchvalues (line 7).
4. Drop attribute employee.department (line 8).

5 Related Work

Database evolution has been widely discussed in the literature and therefore very varied approaches have been proposed. The evolution of object-oriented databases and relational databases, including the propagation of changes automatically down to the extension of the database, has received great attention and the research results have been included in prototypes or in commercial DBMS (see, for example [1]). However they lack the consideration of a conceptual level which allows the designer to work at a higher level of abstraction [10].

In [6] an abstract framework which takes into account both conceptual and logical levels is presented and the necessity of automatically propagating down (forward strategy) the changes performed at the conceptual level is stated. The different papers dealing with forward engineering mainly differ in the way they address the propagation of the conceptual changes down to the logical schema.
and to the extension. For example, a taxonomical approach is followed in [15], which proposes a taxonomy of changes for ER structures and the impact of these changes on relational schemes is analyzed. However this paper does not study how to reflect the schema evolution in the extension of the database.

Other approaches, more similar to ours, propose various ways to capture knowledge about the mappings performed to obtain the logical schema of a conceptual schema. This information is used subsequently in order to obtain the new logical schema associated to the changed conceptual schema. In [8], for example, the sequence (called history) of mappings performed in order to obtain the logical schema is stored. In this way the mappings affected by the changes can be detected and modified, whereas the rest can be reexecuted without any modification. Our approach has the same aim as this one but differs in that we follow a meta-modelling approach.

A meta-modelling approach is also proposed in [10], [14] and [12]. In the case of [10] only a conceptual meta-model is considered whereas we also make use of a logical and a translation meta-model. With respect to [14], the authors make use of a meta-modelling approach with a different goal since the paper deals with the definition of a query language for evolving information systems. In [12] a generalization of the traditional information system notion similar to ours has been proposed. However, some differences with respect to our proposal are worth noting. Firstly, in [12] not only data modelling is taken into account (as we do) but also process and behaviour specification. Secondly, in [12] only the conceptual level is under consideration so that the proposed architecture includes only one information system. Finally, the information processor of an information system is concerned with the modification of the structure and also of the population, instead of using different information processors for each one of these processes as we propose.

6 Conclusions

In this paper we have presented an architecture for managing database evolution with a forward strategy. The architecture consists of four information systems whose information schema capture the relevant modelling elements. As the main contribution, a translation information system is considered, which reflects the translations performed between the conceptual and logical schemas of the database. Evolution changes performed in the conceptual database schema are reflected in the logical schema and the extension of the database making use of structural and internal events. An implementation of our architecture using Oracle has also been presented.

As a direction of future work, the problems related with the evolution of integrity constraints have to be analyzed. Furthermore, a comprehensive support within our architecture for relationship evolution, following the ideas of [2], is a goal for further development.
References

Reifying Design Patterns to Facilitate Systems Evolution

Florida Estrella\textsuperscript{1,2}, Zsolt Kovacs\textsuperscript{2}, Jean-Marie Le Goff\textsuperscript{2}, Richard McClatchey\textsuperscript{1} and Norbert Toth\textsuperscript{1,2}

\textsuperscript{1}Centre for Complex Cooperative Systems, UWE, Frenchay, Bristol BS16 1QY UK, Tel: +44 1179 656261, FAX: +44 22 767 8930
{Florida.Estrella, Richard.McClatchey, Norbert.Toth}@cern.ch
\textsuperscript{2}EP Division, CERN, Geneva, Switzerland. Tel +41 22 767 6559, FAX: +41 22 767 8930
{Zsolt.Kovacs, Jean-Marie.Le.Goff}@cern.ch

Abstract. In the Web age systems must be increasingly flexible, reconfigurable and adaptable in addition to being rapidly developed. As a consequence, designing systems to cater for change is becoming critical to their success. Allowing systems to be self-describing or description-driven is one way to enable this. To address the issue of evolvability in information systems, this paper proposes a pattern-based description-driven architecture. The proposed architecture embodies four pillars - firstly, the adoption of a multi-layered and reflective meta-level architecture, secondly, the identification of four modeling relationships that must be made explicit to be examined and modified dynamically, thirdly the identification of five patterns which have emerged from practice and have proved essential in providing reusable building blocks, and finally the encoding of the structural properties of these design patterns by means of one pattern, the Graph pattern. A practical example of this is cited to demonstrate the use of description-driven data objects in handling system evolution.

1 Background

A crucial factor in the creation of flexible web-based information systems dealing with changing requirements is the suitability of the underlying technology in allowing the evolution of the system. Exposing the internal system architecture opens up the architecture, consequently allowing application programs to inspect and alter implicit system aspects. These implicit system elements can serve as the basis for changes and extensions to the system. Making these internal structures explicit allows them to be subject to scrutiny and interrogation.

A reflective system utilizes an open architecture where implicit system aspects are reified to become explicit first-class meta-objects [1]. The advantage of reifying system descriptions as objects is that operations can be carried out on them, like composing and editing, storing and retrieving, organizing and reading. Since these meta-objects can represent system descriptions, their manipulation can result in change in system behaviour. As such, reified system descriptions are mechanisms that can lead to dynamically modifiable and evolvable systems. Meta-objects, as used here, are the self-representations of the system describing how its internal elements can be accessed and manipulated. These self-representations are causally connected to the internal structures they represent: changes to these self-representations immediately effect the underlying system. The ability to dynamically augment, extend and re-define system specifications can result in a considerable improvement in flexibility. This leads to dynamically modifiable systems that can adapt and cope with evolving requirements, essential in a web-oriented system.
There are a number of OO design techniques that encourage the design and development of reusable objects. In particular design patterns are useful for creating reusable OO designs [2]. Design patterns for structural, behavioural and architectural modeling have been documented and have provided software engineers rules and guidelines that they can immediately (re-)use in software development. Reflective architectures that can dynamically adapt to new user requirements by storing descriptive information which can be interpreted at runtime have lead to so-called Adaptive Object Models [3]. These are models that provide meta-information about domains that can be changed on the fly. Such an approach, proposed by Yoder, is very similar to the approach adopted in this paper.

A Description-Driven System (DDS), as defined by the work reported in this paper, is an example of a reflective multi-level architecture [4]. It makes use of meta-objects to store domain-specific system descriptions, which control and manage the life cycles of meta-object instances, i.e. domain objects. The separation of descriptions from their instances allows them to be specified and managed to and to evolve independently and asynchronously. This separation is essential in handling the complexity issues facing many computing applications and allows the realization of interoperability, reusability and system evolution since it gives a clear boundary between the application’s basic functionalities from its representations and controls. As objects, reified system descriptions of DDSs can be organized into libraries or frameworks dedicated to the modeling of languages in general, and to customizing its use for specific domains in particular.

This paper shows, for the first time, how the approach of reifying a set of design patterns can be used as the basis of a description-driven architecture and can provide the capability of system evolution. (The host project, CRISTAL, is not described in detail here. Readers should consult [5] for further detail). The next section establishes how semantic relationships in description-driven systems can be reified using a complete and sufficient set of meta-objects that cater for Aggregation, Generalization, Description, Dependency and Relationships. In section 3 of this paper the reification of the Graph Pattern is discussed and section 4 investigates the use of this pattern in a three-layer reflective architecture.

2 Reifying Semantic Relationships

In response to the need to treat associations on an equal footing with classes, a number of published papers have suggested the promotion of the relationship construct as a first-class object [6]. Reification is used in this paper to promote associations to the same level as classes, thus giving them the same status and features as classes. Consequently, associations become fully-fledged objects with their own attributes to represent their states, and their own methods to alter their behaviour. This is achieved by viewing the relationships themselves as patterns.

Different types of relationships, representing the many ways interdependent objects are related, can be reified. The proper specification of the types of relationships that exist amongst them is essential in managing the relationships and the propagation of operations to the objects they associate. This greatly improves system design and implementation as the burden for handling dependency behaviour emerging from relationships is localized to the relationship object. Instead of providing domain-
specific solutions to handling domain-related dependencies, the relationship objects handle inter-object communication and domain consistency implicitly and automatically.

Reifying relationships as *meta-objects* is a fundamental step in the reification of design patterns. The next sections discuss four types of relationships, as shown in Figure 1. A structural relationship is one that deals with the structural or static aspects of a domain, examples being the Aggregation and the Generalization relationships. A behavioural relationship deals with the behavioural or dynamic aspects of a domain such as the Describes and Dependency relationships.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Structural Relationship</th>
<th>Behavioral Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td>Generalization</td>
<td>Describes</td>
</tr>
</tbody>
</table>

Fig 1: A classification of relationships.

There is insufficient space in this paper for an exhaustive discussion of each of these relationships. Those that are covered are the links that have proved essential in practice and have emerged from a set of five design patterns: the Type Object Pattern [7], the Tree Pattern, the Graph Pattern, the Publisher-Subscriber Pattern and the Mediator Pattern [8]. Interested readers should refer to [9] for a more complete discussion about the taxonomy of semantic relationships.

2.1 The Aggregation Meta-object

The familiar Tree Pattern [10] models the Aggregation relationship and the objects it relates. Reifying the Tree pattern provides developers with the Tree pattern meta-object, which provides applications with a reusable construct – and essential prerequisite for evolvability. An essential requirement in the reification of the Tree pattern is the reification of the Aggregation relationship linking the nodes of the tree. For this, aggregation semantics must first be defined.

Typically, operations applied to composite objects are by default propagated to their aggregates. This is a powerful mechanism as it allows the implicit handling of the management of interrelated objects by the objects themselves through *the manner in which they are linked together*. By reifying the Aggregation relationship, the three aggregation properties of transitivity, anti-symmetry and propagation of operations can be made part of the Aggregation meta-object. Thus, the state of the Aggregation relationship and the operations related to maintaining the links among the objects it aggregates are localized to the link itself.
Figure 2 illustrates the inclusion of the reified Aggregation relationship in the Tree pattern. The Aggregation meta-object manages and controls the link between the tree nodes, and enforces the propagation of operations from parent nodes to their children.

2.2 The Generalization Meta-object

The semantics of generalization revolve around inheritance, type checking and reuse, where subclasses inherit the attributes and methods defined by their superclass. The subclasses can alter the inherited features and add their own. This results in a class hierarchy organized according to similarities and differences. Unlike the Aggregation relationship, the Generalization semantics are implemented by most programming languages, as built-in constructs integrated into the language semantics.

This paper advocates extending the programming language semantics by reifying the Generalization relationship as a meta-object. Consequently, programmers can access the generalization relation as an object, giving them the capability of manipulating superclass-subclass pairs at run-time. As a result, application programs can utilize mechanisms for dynamically creating and altering the class hierarchy, which commonly require re-compilation for many languages.

Generalization exhibits the transitivity property in the implicit propagation of attributes and methods from a superclass to its subclasses. The transitivity property can also be applied to the propagation of versioning between objects related by the Generalization relationship. Figure 3 illustrates the Tree pattern with the Generalization and Aggregation relationships between the tree nodes reified.

2.3 The Describes Meta-object

In essence the Type Object pattern [7] has three elements, the object, its type and the Describes relationship, which relates the object to its type. The Type Object
pattern illustrates the link between meta-data and data and the Describes relationship that relates the two and is a fundamental component of description-driven systems (see later). The reification of the Describes relationship as a meta-object provides a mechanism for explicitly linking object types to objects. The Describes meta-object provides developers with an explicit tool to dynamically create and alter domain types, and to modify domain behaviour through run-time type-object alteration.

![Diagram of the Type Object Pattern with a reified Describes relationship.](image)

**Fig 4**: The Type Object Pattern with a reified Describes relationship.

The Describes relationship does not exhibit the transitivity property. This implies that the propagation of some operations is not the default behaviour since it cannot be inferred for the objects and their types. Thus, the Describes meta-object should include a mechanism for specifying propagation behaviour and consequently, programmers must accept the default relationship behaviour or override it to implement domain-specific requirements. Figure 4 illustrates the transformation of the Type Object pattern with the use of a reified Describes relationship. The object pointer (in Figure 4a) is dropped since it is insufficient to represent the semantics of the link relating objects and their types. Instead, the Describes meta-object (in Figure 4b) is used to manage and control the Type Object pattern relationship.

### 2.4 The Dependency Meta-object

In the Publisher-Subscriber pattern, subscribers are automatically informed of any change in the state of its publishers. Thus, the association between the publisher and the subscriber manages and controls the communication and transfer of information between the two. Reifying the Publisher-Subscriber dependency association (hereafter referred to as the Dependency association) means that these mechanisms can be **generically implemented and automatically enforced by the Dependency meta-object itself and taken out of the application code**. This represents a significant breakthrough in the simplification of application codes and in the promotion of code reuse.

The reification of the Dependency relationship is significant in that it provides an explicit mechanism for handling change management and consistency control of data. The Dependency meta-object can be applied to base objects, to classes and types, to components of distributed systems and even to meta-objects and meta-classes. This leads to an homogeneous mechanism for handling inter-object dependencies within and between layers of multi-layered architectures.

The Event Channel of the Publisher-Subscriber pattern [11] and the Mediator of the Mediator pattern are realizations of the Dependency relationship. The Event
Channel is an intervening object that captures the implicit invocation protocol between publishers and subscribers. The Mediator encapsulates how sets of objects interact by defining a common communication interface. By utilizing the Describes relationship, an explicit mechanism can be used to store and manage inter-object communication protocols. Figure 5 illustrates the use of reified Dependency meta-object in the Publisher-Subscriber pattern (a) and the Mediator pattern (b).

![Diagram](image)

**Fig 5**: The Event Channel and the Mediator as reified dependencies.

### 2.5 The Reified Relationship Meta-object Protocol (MOP)

The relationship Meta-Object Protocol, or MOP, defines the set of methods for querying information about relationships and for manipulating the relationship meta-objects [12]. It is the relationship MOP that controls and manages the relationship meta-object and its object instances. As the relationship semantics involve both the structural and dynamic aspects of a domain, the relationship meta-objects and their MOPs are mechanisms for defining and configuring system behaviour.

Before detailing the relationship MOP specification, it is essential to define the basic set of Relationship attributes. The relationship is a link between parent nodes and children nodes, as shown in Figure 6. The figure shows the cardinality of the relationship with respect to its parents and its children. The Aggregation, Generalization, Describes and Dependency relationships are restricted to having exactly one parent node. The fact that the Aggregation relationship has a single parent node is defined by its semantics. With regards to Generalization, Describes and Dependency links, the cardinality of one for the parent is a design decision that simplifies the relationship model. Allowing many superclasses in the Generalization link results in multiple inheritance. Allowing many types in the Describes link creates a multi-typed object, and allowing many publishers in the Dependency link implies that subscribers get data from many subjects.

The four relationship meta-objects discussed above manifest the links that exist among the objects participating in the five design patterns listed in the introduction to this section. With the use of reified relationships, these five patterns can be modeled as a single graph, using the Graph pattern. Consequently, the five design patterns can
be structurally reified as a Graph pattern, as shown in the next section, with the appropriate relationship meta-object to represent the semantics relating the individual pattern objects.

Fig 6: The cardinality of relationships.

3 The Reified Graph Pattern

The graph and tree data structures are natural models to represent relationships among objects and classes. As the graph model is a generalization of the tree model, the graph model subsumes the tree semantics. Consequently, the graph specification is applicable to tree representations. The compositional organization of objects using the Aggregation relationship also forms a graph. Similarly, the class hierarchy using the Generalization relationship creates a graph. These two types of relationships are pervasive in computing, and the use of the Graph pattern to model both semantics provides a reusable solution for managing and controlling data compositions and class hierarchies and a valuable approach to enabling system evolution.

Fig 7: An UML diagram of the Graph Meta-object.

The way dependent objects are organized using the Dependency association also forms a graph. Dependency graphs are commonly maintained by application programs, and their implementations are often buried in them. The reification of the Dependency meta-object ‘objectifies’ the dependency graph and creates an explicit Publisher-Subscriber pattern. Consequently, the dependency graph is treated as an object, and can be accessed and manipulated like an object. The same argument applies to the Describes relationship found in the Type Object pattern. The link between objects and their types creates a graph. Reifying the Describes relationship results in the reification of the Type Object pattern. With the reification of the Type Object pattern, the resulting graph object allows the dynamic management of object-type pairs. This capability is essential for environments that dynamically change.

A UML diagram of the Graph meta-object is shown in Figure 7. The Node class represents the entities of the domain objects, classes, data, meta-data or components. The Relationship is the reification of the link between the Nodes. The aggregated
links between the Node and the Relationship are bi-directional. Two roles are defined for the two aggregated associations - that of the parent, and that of the child. The parent aggregation, symbolized by the shaded diamond, implies that the lifecycle of the relationship is dependent on the lifecycle of the parent node. The child aggregation behaves similarly.

The use of reflection in making the Graph pattern explicit brings a number of advantages. First of all, it provides a reusable solution to data management. The reified Graph meta-object manages static data using Aggregation and Generalization meta-object relationships, and it makes persistent data dependencies using the Describes and Dependency relationships. As graph structures are pervasive in many domains, the capture of the graph semantics in a pattern and objectifying them results in a reusable mechanism for system designers and developers. Another benefit of having a single mechanism to represent compositions and dependencies is its provision for interoperability. With a single framework sitting on top of the persistent data, clients and components can communicate with a single known interface. This greatly simplifies the overall system design and architecture, thus improving system maintainability. Moreover, clients and components can be easily added as long as they comply with the graph interface.

Complexity is likewise catered for since related objects are treated singly and uniformly. The semantic grouping of related objects brings transparency to clients' code and the data structures provided by the Graph meta-object organize data into atomic units, which can be manipulated as single objects. Objectifying graph relationships allows the implicit and automatic propagation of operations throughout a single grouping. Another benefit in the use of the reified graph model is its reification of the link between meta-data and data. As a consequence, the Graph meta-object not only provides a reusable solution for managing domain-semantic groupings, but can also be reused to manage the links between layers of meta-level architectures.

Fig 8: A three-layer reflective Description-driven system architecture.
4 Putting It All Together – Reified Patterns as the Basis of DDSs

This paper proposes that the reified Graph pattern provides the necessary building block in managing data in any DDS architecture. Figure 8 illustrates a proposed description-driven architecture. The architecture on the left-hand side is typical of layered systems such as the multi-layered architecture specification of the OMG [13]. The relationship between the layers is an Instance-of. The instance layer contains data that are instances of the domain model in the model layer. Similarly, the model layer is an instance of the meta-model layer. On the right hand side of the diagram is another instance of model abstraction. It shows the increasing abstraction of information from meta-data to model meta-data, where the relationship between the two is also an Instance-of. These two architectures provide layering and hierarchy based on abstraction of data and information models.

This paper proposes an alternative view by associating data and meta-data through description (the is Described by relationship). The Type Object pattern makes this possible. The Type Object pattern is a mechanism for relating data to information describing data. The link between meta-data and data using the Describes relationship promotes the dynamic creation and specification of object types. The same argument applies to the model meta-data and its description of the domain model through the Describes relationship. These two horizontal dependencies result in an horizontal meta-level architecture where the upper meta-level describes the lower base-level (see figure 9). The combination of a multi-layered architecture based on the Instance-of relationship and that of a meta-level architecture based on the Describes relationship results in a description-driven architecture (DDS).

Fig 9: The reuse of the Reified Graph Pattern in a description-driven system.

The reified Graph pattern provides a reusable mechanism for managing and controlling data compositions and dependencies. The graph model defines how domain models are created. Similarly, the graph model defines how meta-data are instantiated. By reifying the semantic grouping of objects, the Graph meta-object can be reused to hold and manage compositions and dependencies within and between layers of a DDS (see figure 9). The meta-level meta-data are organized as a meta-level graph. The base-level data are organized as a base-level graph. Relating these two graphs forms a further graph whose nodes are related by the Describes
relationship. These graphs indicate the reuse of the Graph pattern in modeling relationships in a DDS architecture.

5. CRISTAL as an Example of a Description-Driven System

The research which generated this paper has been carried out at the European Centre for Nuclear Research (CERN) based in Geneva, Switzerland. CERN is a scientific research laboratory studying the fundamental laws of matter, exploring what matter is made of, and what forces hold it together. Scientists at CERN build and operate complex accelerators and detectors whose construction processes are very data-intensive, highly distributed and ultimately require a computer-based system to manage the production and assembly of components. In constructing detectors like CMS [14], scientists require data management systems that can cope with complexity, with system evolution over time (primarily as a consequence of changing user requirements and extended development timescales) and with system scalability, distribution and interoperation.

A research project, entitled CRISTAL (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles [4],[5]) has been initiated to facilitate the management of the engineering data collected at each stage of production of CMS. CRISTAL is a distributed product data and workflow management system which makes use of an OO database for its repository, a multi-layered architecture for its component abstraction and dynamic object modeling for the design of the objects and components of the system. CRISTAL is based on a DDS architecture using meta-objects.

![Diagram of CRISTAL description-driven system architecture.](image)

**Fig 10**: The CRISTAL description-driven system architecture.

The design of the CRISTAL prototype was dictated by the requirements for adaptability over extended timescales, for system evolution, for interoperability, for complexity handling and for reusability. In adopting a description-driven design...
approach to address these requirements, the separation of object instances from object description instances was needed. This abstraction resulted in the delivery of a three-layer description-driven architecture. The model abstraction (of instance layer, model layer, meta-model layer) has been adopted from the OMG MOF specification [15], and the need to provide descriptive information, i.e. meta-data, has been identified to address the issues of adaptability, complexity handling and evolvability.

Figure 10 illustrates the CRISTAL architecture. The CRISTAL model layer is comprised of class specifications for type descriptions (e.g. PartDescription) and class specifications for classes (e.g. Part). The instance layer is comprised of object instances of these classes (e.g. PartType#1 for PartDescription and Part#1212 for Part). The model and instance layer abstraction is based on model abstraction and is an instance of relationships. The abstraction based on meta-data abstraction and is described by relationships leads to two levels - the meta-level and the base-level. The meta-level is comprised of meta-objects and the meta-level model that defines them (e.g. PartDescription is the meta-level model of PartType#1 meta-object). The base-level is comprised of base objects and the base-level model which defines them.

Separating details of model types from the details of single parts allows the model type versions to be specified and managed independently, asynchronously and explicitly from single parts. Moreover, in capturing descriptions separate from their instantiations, system evolution can be catered for while production is underway and therefore provide continuity in the production process and for design changes to be reflected quickly into production. The approach of reifying a set of simple design patterns as the basis of the description-driven architecture for CRISTAL has provided the capability of catering for the evolution of a rapidly changing research data model. In the two years of operation of CRISTAL it has gathered over 26 Gbytes of data and been able to cope with more than 20 evolutions of the underlying data schema without code or schema recompilations.

**Fig 11**: Extending the UML Meta-model using a Reified Graph Pattern.
6 Conclusions

As shown in figure 11, the reified Graph pattern and the reified relationships enrich the meta-model layer by giving it the capability of creating and managing groups of related objects [15]. The extension of the meta-model layer to include constructs for specifying domain-semantic groupings is the proposition of this paper. The meta-model layer defines concepts used in describing information in lower layers. The core OMG/UML meta-model constructs include Class, Attribute, Association, Operation and Component meta-objects. The inclusion of the Graph meta-object in the meta-model improves and enhances its modeling capability by providing an explicit mechanism for managing compositions and dependencies throughout the architecture. As a result, the reified Graph pattern provides an explicit homogeneous mechanism for specifying and managing data compositions and dependencies in a DDS architecture.

This paper has shown how reflection can be utilized in reifying design patterns. It shows, for the first time, how reified design patterns provide explicit reusable constructs for managing domain-semantic groupings. These pattern meta-objects are then used as building blocks for describing compositions and dependencies in a three layer reflective architecture - the description-driven systems architecture. The judicious use and application of the concepts of reflection, design patterns and layered models create a dynamically modifiable system which promotes reuse of code and design, which is adaptable to evolving requirements, and which can cope with system complexity. In conclusion, it is interesting to note that the OMG has recently announced the so-called Model Driven Architecture as the basis of future systems integration [16]. Such a philosophy is directly equivalent to that expounded in this and earlier papers on the CRISTAL description-driven architecture.

Acknowledgments

The authors take this opportunity to acknowledge the support of their home institutes.

References

2. E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.
Managing Configuration with Evolving Constraints in Design Databases

T. W. Carnduff¹, J. S. Goonetillake²
¹School of Computing, University of Glamorgan, Pontypridd CF37 1DL UK
tcarnduf@glam.ac.uk
²Computing Division, UWIC Business School, UK

Abstract Artifacts in engineering design are structurally complex and may be represented in software as recursively composite objects. Due to the evolutionary nature of the design process each artifact and its components may evolve through several versions. This paper describes enhanced database system facilities that are used to group mutually consistent component versions together into useful configurations. The versioning system includes integrity management facilities that allow evolving design constraints to be captured flexibly at individual component/object level. In order to permit evolution, integrity constraints are represented within versionable objects, so-called constraint version objects (CVOs). Inter-dependency constraints can be modelled to express the consistency semantics necessary to combine component artifact versions into useful configurations. The evolution of these configurations can be captured in the database, as configurations are also represented as versioned objects.

1. Introduction

Design objects are either primitive (fine-grained objects) or composite (coarse-grained). Generally a composite object is composed of its own attributes and other objects. These constituent objects may themselves be composite or primitive. A composite design artefact can be logically decomposed into its component parts which in turn may be decomposed recursively in such a way to allow the individual/group of designers to address them separately [1]. Subsequently during the design process, this decomposed complex artifact is recomposed by combining its constituent component objects. In a versioning environment each composite and component design object may have number of versions making this composition task cumbersome. For example, if a composite object/artifact is composed of m objects, each one in n versions, there can be up to n^m different combinations to be used for the construction of configurations out of which only a few may be actually consistent or relevant. A configuration is defined as a structurally composite object formed by combining other configurations (known as sub configurations) and versions of different objects. Useful configurations are formed from versions of constituent objects that are consistent together [2]. Even though several version models have been proposed in the literature only a few address configuration management. Since different configurations may exist due to differing constituent version combinations, it would be useful if the designer had the facility to store meaningful configurations and to keep track of configuration evolution. This can be achieved if configurations can be managed as
versions. Another benefit in this situation is that versions and configurations may be freely combined to construct higher-level configurations [3].

To produce a consistent configuration all of the component versions participating in that configuration should satisfy the inter-dependency constraints imposed on them. Inter-dependency constraints are known as global constraints since their validity spans multiple objects. Successful validation of inter-dependency constraints ensures that a configuration is consistent with its constituent component versions. Consequently, it is important to augment configuration management with an integrity mechanism that checks each configuration for data correctness and consistency. A common problem encountered during the design process is that of frequent constraint changes. Different versions of the same configuration object may have to satisfy different sets of inter-dependency constraints at different times. In [4] an integrity validation model for object versions was presented, which considered the means of representing and managing evolving integrity constraints in a collaborative engineering design environment. We have not located in the literature, a version model that deals with the consistency of design configurations, through constraint management. The objective of this paper is to show how this may be achieved. For the sake of simplicity, we frame our explanation around an easily understood engineering artifact, a bicycle. The object composition graph for a bicycle is depicted in figure 1. This hierarchy is achieved through object aggregation.

![Figure 1. Bicycle composition graph](attachment:image.png)

2. Related Work

Even though version management and configuration management are strongly related areas, little attention has been paid in existing version models to configuration management. Most proposed configuration management systems are either superficial or otherwise unsatisfactory. Versions of different objects are bound together either statically or dynamically to form a configuration [5, 6, 7, 8, 9]. Dynamic references are resolved later by placing the versions that go together in the same layer [10], environment [11], database version [2] or assigning these versions to the same configuration identifier [12]. The selection of these versions is based on either the most recent or the default version. Significantly, these models do not provide any integrity mechanism to capture possible inconsistencies between the versions forming a configuration. This makes the formation of inconsistent configurations very likely. It
is a designer’s responsibility to check whether the constructed configuration is consistent. Furthermore, some configuration management mechanisms are inherently unsatisfactory due to:

- version proliferation (cause by static binding) [5, 6, 7, 8, 9], or
- problems in sharing component versions between configurations (e.g. an object version cannot be part of more than one configuration) [13], or
- explicit copying of versions every time they are used in a configuration [12], or
- lack of facilities for the designer to save and track important configurations for his/her reference, as these systems can deal with only one configuration at a time, e.g. at the released time [14].

Versions and configurations are treated as different concepts in most of the existing version models. This distinction provides some drawbacks, such as difficulty in freely combining versions and configurations to form higher-level configurations [3]. Some research has focused on reducing this distinction between versions and configurations by treating and managing configurations as versions [3, 15]. Nevertheless, none of these version models addresses support for the consistency of configurations using evolving constraints.

The model proposed by Doucet and Monties [16] addresses both versions and evolving integrity constraints using a Database Version (DBV) approach [2]. Each database version contains a configuration composed of one version of each constituent object. Constraint evolution also involves the production of multiple DBVs. Consistency is confined to each database version and consequent configuration level. The checking of some constraints spans multi-database versions, which adds to the complexity. It is not clear whether this model can be applied to a cooperative and distributed design environment. We believe that in a cooperative design environment, the system may end up with a large number of database versions, which will impose a considerable storage and organisational overhead.

3. Version Binding

There are two object-binding mechanisms used in forming configurations namely, static and dynamic (or generic). With static binding each composite version refers directly to specific sub component versions. The creation of a new sub component version may lead to the creation of a new composite object version with consequent version proliferation. To illustrate version proliferation (or version percolation) we consider a bicycle tyre system (figure 2(a)), which consists of the two components wheel and tyre. A wheel is in turn composite, with the two components hub and rim. The first version of the composite tyre system is depicted in figure 2(b). Suppose a new version of rim (R2) is created then figure 2(c) shows how this leads to the creation of new versions of the other components in the composition hierarchy. With dynamic binding on the other hand a composite version refers to the generic version of its components and not to a specific version. In this paper, a configuration which uses dynamic binding is referred to as a generic configuration. Generic references are often more appropriate than static references for two reasons [14]. Firstly, versioning a component object does not lead to the creation of a new version of the composite artifact (and vice versa). Secondly, it is possible to bind the generic reference to any
required version of the component object so long as the inter-dependency constraints are satisfied. The replacement of a dynamic reference with a reference to a specific version is called *dynamic reference resolution*. The disadvantages associated with dynamic referencing relate to the extra levels of indirection. On balance, however, dynamic references are preferable to static references and we have adopted this technique in our configuration management model. The generic configuration concept is used initially to identify the constituent component objects of the composite artifact. To this end, we recognise two possible ways of using dynamic references. We illustrate these points using a bicycle wheel consisting of the two components, rim and hub.

![Figure 2. Static Binding](image)

### Figure 2. Static Binding

i) *Version-Generic*: In this method each version of a composite object refers to the generic versions of its component objects as depicted in figure 3 (a). The shaded nodes denote a generic configuration. With this method the composite object is modelled (in this case wheel) as shown in figure 3(b) with references to generic versions of the component objects. Each instance of this class represents a generic configuration.

![Figure 3. Dynamic Binding –(Version-Generic)](image)

### Figure 3. Dynamic Binding –(Version-Generic)

ii) *Generic-Generic*: In this method the generic version of the composite object refers to the generic versions of its components (figure 4 (a)). The versions of the composite
object can be handled independently from its components, while an indication of its components can still be provided through the generic version. This mechanism enables the composite object to be modelled as a separate object independent from generic references as shown figure 4(b). The shaded nodes in figure 4(a) indicate the generic configuration. There is only one generic configuration unlike the version-generic approach. This is a deviation from the widely used generic configuration representation, however, it still achieves its main purpose of identifying component objects in a generic configuration.

Figure 4. Dynamic Binding – (Generic-Generic)

Most existing configuration management systems that employ dynamic binding use the Version-Generic approach. Dynamic reference resolution takes place only when the configuration is to be released or accessed. In general, one configuration is constructed when required, based on a version selection procedure (for example by combining all of the default versions, or the most recent versions in the composition hierarchy). This method leads to the creation of different configurations at different times as new versions are introduced. Consequently information on previously created configurations is lost. We submit that this is inappropriate since the designer should be able to save important configurations for later reference. Furthermore, there is no guarantee that the selected versions are consistent together. For example it would be incorrect to assume that the latest version of one component would always be consistent with the latest version of another component, without checking their inter-dependency constraints.

4. Configuration Management Requirements

In practical terms, the designer should be able to construct configurations by selecting and combining component versions. The environment should be flexible enough to allow experimentation with different version combinations. For example the designer may experiment with a bicycle structure version (a sub configuration of bicycle) made of titanium with both hybrid and mountain bicycle versions (figure 5(a)). Alternatively
the designer may experiment with a mountain bicycle version with structure versions made of different materials (figure 5(b)). The designer should be provided with a facility to store important configurations and to keep track of configuration evolution. The sub configurations should be able to be combined freely with other versions and sub configurations to construct higher-level configurations. These objectives can be achieved by managing configurations as versions. From the system point of view maintaining a number of configuration versions should not lead to the data redundancy that would arise due to the unnecessary copying of object versions for each configuration.

Figure 5. Combining versions for configurations

Consistency between the version objects participating in a configuration is achieved by satisfying the inter-dependent constraints imposed on them. In terms of the designer’s role in managing the data validation aspects of configuration consistency, he/she should be freed from the need for:

- typing selection criteria to produce views every time a configuration is constructed, or
- changing and compiling application programs when constraints change, or
- any manual checking of the inter-dependency constraints to ensure the consistency of configurations.

Checking the consistency of the selected versions of the configuration should be handled automatically by the configuration management system based on the pre-defined inter-dependency constraints.

5. Proposed Configuration Management Model

In meeting the configuration management requirements set out in the previous section, we have utilised the version model described by Goonetillake et al. [4]. We begin by looking at how composite artifacts are represented within versioned configurations using dynamic referencing. If a composite artifact is modelled using dynamic referencing (figure 3(a)), it is not possible to replace the generic version reference with a specific version reference unless the generic and specific versions are both instances of the same class. Since they contain different information they are instances of different classes. A class must be declared which incorporates references to the specific version types, so that specific configurations can be constructed.
5.1. The Configuration Object and Version Sharing

It is evident that both of the methods of moving from generic to specific configurations are problematic. Attribute values must be copied every time a specific configuration is created. Consequently, there will be considerable data duplication as different component versions are selected for configuring with a single composite version during tentative experimentation. Figure 6(a) shows the data redundancy involved in tentatively combining a wheel version with a number of rim and hub versions. As a solution to these issues we have used the *generic-generic* approach (option (ii) in section 3) to represent the generic configuration. This allows us to define a composite object with a separate class which is independent of references to generic versions - see figure 4(b). A separate configuration object is used to combine the corresponding object versions to form a configuration - see figure 6(b). This provides the flexibility required to combine object versions whilst avoiding redundancy through version sharing. Modelling the configuration object in this way allows the selection of specific versions of a particular object (irrespective of whether it is physically composite) and versions of a configuration object. For example a wheel object and wheel configuration are two different entities. A wheel object version is part of a wheel configuration version. This may not represent the aggregation (is part of) relationship that should exist between a composite object version and its component versions (as in figure 6(a)). However, the advantage of this approach is that it enables the same composite object version to be shared by many configurations without data duplication (figure 6(b)). The version of the main/composite object (in this case the wheel object version) is the key object in the configuration version. The component versions should be selected in combination with this key object version. The key object version should reflect the main characteristics of the whole configuration version (for example the wheel object version should represent the main features of the wheel configuration version, for example a small wheel with 32 spokes, coloured blue weighing less than 900g). This is important in the validation of inter-dependency constraints when constructing higher-level configurations.

![Figure 6. Constructing Specific Configuration versions](image)

95
5.2. Configurations as Versions

We now go on to illustrate how configurations are managed as versions. For illustrative purposes we return to the composite tyre system artifact (figure 2(a)). From the configuration management point of view, the composite object wheel with its hub and rim form a sub configuration version. Unless otherwise stated sub configurations are referred to as configurations since there is no semantic difference in the way they are both managed. The generic version of a wheel refers to the generic versions of both hub and rim, to form a generic configuration (figure 7 (a)). Each generic version in turn refers to its corresponding version set. The designer explicitly creates specific configuration versions by selecting specific versions from each version set. A specific configuration version is an instance of the corresponding configuration object class, which is modelled to combine the object versions that go together – see figure 7(b). To this end, references are made from the configuration object to the corresponding constituent objects (e.g. figure 7(b)). As shown in figure 7(c) it is possible to share object versions between configuration versions without duplicating data.

Figure 7. Construction of wheel sub configuration
A configuration version can have at most one version from each component. Each configuration version is associated with its version information including configuration number, time of creation and status details (e.g. default, active, last, deleted). The configuration number is generated and assigned by the system so that each configuration is identified uniquely within the system. The designer has freedom to set any configuration as the default configuration so long as it adheres to the currently applicable set of constraints (see section 6). The new configuration versions are not derived from preceding configurations, but are created from scratch by selecting corresponding object versions. Consequently a configuration version set is represented as a list and it is not therefore possible to logically cluster revisions of alternative configurations. However, an alternative configuration version and its revisions are identifiable through the use of configuration number. As with version management, this list is referenced by a generic configuration version (not the generic configuration referred to earlier), which contains meta information relating to the configuration version set (e.g. the default configuration, total number of configurations, the last configuration, owner of the configuration). In this way, higher-level configurations can be constructed by freely combining configuration object versions.

A version may or may not belong to a configuration. Moreover, an object version which is consistent in one configuration, may not be consistent in another configuration. A configuration version can be constructed in either a top down or bottom up manner. Our approach allows partial configurations to be defined where one or more constituent object versions have not yet been selected.

6. Integrity Validation Model

6.1. Constraint Evolution

It is necessary to check whether the selected versions constitute a consistent configuration through satisfying a given set of constraints. The constraints that are used to validate configurations are known as inter-dependent or global constraints. The scope of these constraints spans multiple objects. At the configuration level, validation is mainly concerned with selecting valid sub components. However, a set of design constraints will evolve over time with modifications [17, 18, 19, 20], the addition of new constraints and the omission of existing constraints. A constraint management framework for engineering design was described by Goonetillake et al. [4]. In essence this framework is based around an object version model in which the constraint categories are value based (e.g. range, enumeration, relationship). The constraints are represented and stored in Constraint Version Objects (CVOs). A CVO aggregates a set of currently active constraints applicable to a particular artifact. Constraint evolution is effected by producing new CVOs, that is new versions of the set of constraints contained in a CVO. This model takes cognisance that a designer/user is not a computer programmer. The designer is provided with a graphical user interface to create new CVOs in a form-filling manner in the event of
constraint changes. The creation of an executable CVO is transparent to the designer and handled by the system.

In summary a child CVO contains only modified and new constraints and inherits unchanged constraints from its parent CVO. This avoids the unnecessary redefinition of constraints. Usually, the last CVO created in the system for a particular artifact will become the default CVO for that artifact. The default CVO for each artifact determines the validity of its versions. The new versions are automatically associated with the default CVO at the creation time. In a collaborative and distributed design environment the default CVO enables designers to identify consistently the current active CVO for a particular artifact. The default object version of an artifact should generally be linked to the default CVO. This work on CVOs is related to object versioning, and as the configuration management model presented here deals with configuration versions, CVO concepts can be applied to configuration management. In constructing a configuration version it is important that each constituent version is individually consistent with the set of constraints in the corresponding default CVO.

In a CAD environment some (global) constraints on complex objects may be decomposed into a set of local constraints on sub components or leaf objects [21]. For example a weight constraint on an artifact may be decomposed into local weight constraints on its sub components as shown in figure 8. A change in such a global constraint will require a change to the corresponding local constraints. However, this change propagation is not automatic and should be carried out by collaborating designers through a process of discussion. Because constraints are created and refined by human specialists through analysis and engineering judgement, no design framework/model should change the design information (constraints) arbitrarily without the necessary human involvement.

![Figure 8. Breaking down of global constraints to local constraints](image)

The terms local and global are relative in relation to CVOs and depend on the number of levels in the design database hierarchy. A global object at one level may be a local object at the next level up in the hierarchy.

### 6.2. Configuration Versions and CVOs

In this section we give a detailed explanation of the means by which the integrity validation system validates configuration versions. The explanation is framed around the wheel structure example. As shown in figure 9 versions of the wheel object are
first validated against their local constraints (e.g. diameter < 55 cm, \{blue, red\} ⊇ finish). These local constraints are imposed on the attribute values of the wheel object. Each CVO contains the active set of local constraints for the wheel object at some point in time.

Since the sub components are first validated against their local constraints, the constituent component versions of a configuration are assumed to be individually consistent with their local constraints. Consequently, a configuration version will only have to be validated against its global constraints (or inter-dependency constraints) as imposed on the constituent versions of that configuration.

These global constraints specify the dependencies between attribute values in each selected component object version. For example for the wheel configuration, the inter-
dependencies could be (number of spoke holes in hub = number of holes in rim) and
(wheel finish = blue) ⇒ (rim finish = black and hub finish = blue). Each configuration
version is automatically associated with the default CVO to be validated against the
constraints in that CVO at its creation time.

Tyre system object versions are managed in the same way as wheel object versions
and are validated independently against their local constraints (figure 10 (a)). The
construction of a tyre system configuration version requires the combination of a tyre
system object version, a wheel configuration version and a tyre object version. The
resulting configuration version will have to be validated against its corresponding
global constraints (figure 10(b)). These global constraints specify the dependencies
between attribute values in the selected tyre system, wheel configuration and tyre
object version (for example wheel weight + tyre weight < 1.5 kg). Each global CVO
contains the active set of global constraints for the tyre system configuration at some
point in time (figure 10(b)).

7. Conclusion

In this paper we have presented a configuration management model in which
configurations are managed as versions. This enables the designer to store useful
configurations and to keep track of configuration evolution. Moreover, it reduces the
distinction between the version and configuration concepts and thus provides the
facility to freely combine configurations and versions together to form higher-level
configurations. The way in which the configuration object is modelled provides the
means to be able to flexibly combine any composite artifact version with any number
of constituent versions in the construction of different configurations, without data
duplication. The uniform treatment of versions and configurations provides the means
to combine them freely in the construction of higher-level configurations. The
integrity validation system checks the consistency of versions selected for a
configuration. The configuration model manages the consistency of object versions
(including configuration versions) through the management of constraint evolution.
The system described has been implemented in prototype form using the Java
programming language, the Objectivity object database system, and for constraint
management aspects of the work, the Prolog declarative programming language. For
reasons of space no further consideration is given to implementation here, but will be
described in a future paper.

References

Engineering Design Systems, ASCE Journal of Computing in Civil Engineering, 6(3), 248-
265.
2. Cellary, W., Jomier, G. (1990). Consistency of Versions in Object Oriented Databases,
Proc. 16th Int. Conf. on Very Large Databases (VLDB 16), 432-441, Australia.
Intentions of Operations –
Characterization and Preservation

Mira Balaban*, Steffen Jurk**

Ben-Gurion University of the Negev, Beer-Sheva, Israel
mira@cs.bgu.ac.il

Brandenburg Technical University, Cottbus, Germany
sj@informatik.tu-cottbus.de

Abstract. Frequent changes of software requirements imply changes of the underlying database, like database schema, integrity constraints, as well as database transactions and programs. Tools like ERWin, DBMain and Silverrun help developers in applying these changes. Yet, the automatic derivation might pose a problem: Since the developer is not aware of the details of the derivation applications, the resulting programs might include contradictory actions. That is, intentions of programs might be reversed by the automatic derivation, resulting a different behavior than expected by the developer.

In this paper, a compile-time algorithm that achieves preservation of intentions is suggested. The algorithm revises a composite program into a program without contradictory actions. It is based on a fine analysis of effects, that is sensitive to computation paths. The output program is expressive and efficient since it interleaves run-time sensitive analysis of already reduced effects within the input program. The compile-time reduction of effects accounts for the efficiency; the run-time sensitivity of effects accounts for the expressiveness. The novelty of the proposed approach is in combining static and dynamic analysis in a way that run-time overhead is minimized without sacrificing the expressivity of the resulting program.

1 Introduction

Frequent changes of software requirements imply changes of the underlying database, like database schema, integrity constraints, as well as database transactions and programs. Methods supporting the design of database programs [9, 10, 8, 13] and tools like ERWin, DBMain and Silverrun help developers in applying these changes. Yet, the automatic derivation might pose a problem: Since the developer is not aware of the details of the derivation applications, the resulting programs might include contradictory actions. That is, intentions of programs might be reversed by the automatic derivation, resulting a different behavior than expected by the developer. The work described in this paper deals with the characterization and preservation of program intentions.

* This work was supported in part by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev.
** This research was supported by the DFG, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).
Example 1. Assume a developer is designing (changing) a database where inclusion and exclusion dependencies have to be inserted or removed, due to some changing requirements. The design tool repairs insertions or deletions by adding necessary insertions or deletions, so that the current dependencies are satisfied. The problem is that certain dependency combinations can lead to intention contradiction, of which neither the developer nor the users, are aware.

For example, assume a unary relation $A$ and a binary relation $B$, with an inclusion relation between $\pi_1(B)$ (the first column of $B$) and $A$, and an exclusion relation between $\pi_2(B)$ and $A$. The design tool repairs an insertion of $(x, y)$ into $B$ by inserting $x$ into $A$ and deleting $y$ from $A$. Analogously, an insertion of $x$ into $A$ is repaired by deleting from $B$ all tuples with $x$ as a first element. For the action $\text{insert}(B, (x, y))$, the following consistent program $S$ is derived:

$$S(x, y) = \text{insert}(B, (x, y));$$

$$\text{if } x \notin A \text{ then}$$
$$\text{insert}(A, x);$$
$$\text{while } \sigma_{2=x}(B) \neq \emptyset \text{ do: delete}(B, \bar{t}) \text{ where } \bar{t} \in \sigma_{2=x}(B);$$
$$\text{if } y \in A \text{ then}$$
$$\text{delete}(A, y);$$
$$\text{while } \sigma_{1=y}(B) \neq \emptyset \text{ do: delete}(B, \bar{t}) \text{ where } \bar{t} \in \sigma_{1=y}(B);$$

Now consider the action $\text{insert}(B, (x, x))$, whose natural intention is $(x, x) \in B$. A careful examination of $S(x, x)$ reveals that while the dependency requirements are indeed being enforced, the original intention of an action $\text{insert}(B, (x, x))$ might get lost. That is, the tuple $(x, x)$ is not inserted, although $A$ might be changed. An intention preservation policy requires that if the original intention is lost than the repair program is rejected and not completed successfully. In this example it means that $S(x, x)$ should be rejected if the tuple $(x, x)$ can not be inserted properly.

Problems of intention contradiction arise frequently in Rule Triggering Systems (RTSs) ([13, 7, 14]), and have also been studied within the context of integrity enforcement in databases, where repairing updates can undo each other ([2, 11]). However, the problem is not handled in regular database maintenance systems like RTSs.

In this paper we introduce an intention (effect) preserving mechanism, that combines static analysis of an action with run-time tests. The mechanism is based on characterization of contradictory actions. Actions are assumed to have intentions (effects), which are logic constraints. Actions contradict if their intentions do so ([5]). The intentions of actions (atomic operations) are set by the user (action initiator). The intentions of programs (composite operations like transactions) are computed by the program management system.

The effect preserving mechanism that we introduce in this paper is a program transformation. The main idea is to augment computation paths that might lead to an unsatisfiable intention, with necessary intention preserving tests. The delicate problem here is to avoid unnecessary tests that might lead to unnecessary rejections, and still achieve performance that improves run-time testing for intention violation. Therefore, there is a need to characterize intentions (effects)
in a way that distinguishes between actions that lie on different computation
cp-paths, and reduce the amount of intention testing that is required at run-time.
The mechanism suggested here combines static transformation with run-time
tests. At compile-time, a computation tree that analyzes all computation paths
of the program is constructed, and the program is augmented with tree naviga-
tion and intention tests. The nodes of the tree are associated with maximally
reduced constraints that capture the intention preservation requirements.

The mechanism achieves a better performance than run-time methods for
intention preservation since the time consuming operation of tree construction
and constraint reduction is applied at compile-time. Only statically unresolved
intentions are left for run-time evaluation. The run-time overhead is linear in
the size of the program since the size of the reduced conditions is independent of
the program size, and is expected to be rather small and easy to evaluate. The
novelty of our approach is in combining compile-time and run-time processing in
a way that run-time overhead is minimized, without sacrificing the expressivity
of the revised program.

In Section 2 the language of programs is defined, and Section 3 introduces the
path sensitive characterization of effects (intentions of actions and programs).
Section 4 presents the combined static-dynamic revision of programs, that en-
forces the effect preservation property. Section 5 concludes the paper.

2 A Restricted Language of Database Programs

The language of database programs studied in this paper is restricted to an
imperative language with a sequence and a conditional combinator. Programs
are built over a finite set of typed state variables \( X \) (a state space). A state is a
well typed value assignment to the variables in \( X \). For example, in a relational
database with relations \( R_1, \ldots, R_n \), the state space is \( \{R_1, \ldots, R_n\} \), and any
assignment of concrete relations to the relation variables results a database state.
A language over \( X \) is denoted \( L(X) \).

The language symbols include, besides the state variables, input and local
variables, and self-evaluating symbols (language constants). Input variables are
not assignable. The primitive actions are \( \text{skip} \) (a no-op operation), \( \text{fail} \) (rollback,
the impossible action), and \( \text{assignment action} \) – well typed assignments to state
variables. Assignments include, besides the assigned state variable, only input
variables and constants, i.e., non-assignable symbols. The \( \text{fail} \) action leads to
the undefined state, which in transactional databases correspond to a rollback,
which undoes undefined states by restoring the old state.

The two constructors that are studied in this paper are sequential composit-
ion, denoted \( (S_1; S_2) \) and guarded deterministic choice (conditional), denoted
\( (\text{if } P \text{ then } S_1 \text{ else } S_2) \) where \( S_1 \) and \( S_2 \) are programs and \( P \) a condition.
We use \( \text{if } P \text{ then } S \text{ for abbreviating } \text{if } P \text{ then } S \text{ else } \text{skip} \). The formal
semantics of the language is defined as in Dijkstra’s guarded commands lan-
guage ([1, 6, 2]).
3 Path Sensitive Characterization of Effects

In this section we define the effects of composite programs, on the basis of effects of primitive actions. First we define effects of sequences of primitive actions. We distinguish between desired effects, denoted $\text{effects}_D(S)$, to executed effects, denoted $\text{effects}_E(S)$: The first, expresses the aggregated desired effects of all primitive actions in a program and it might not hold after the program is completed. The second, expresses the historical effects of all primitive actions in a program, as they were when executed, and it always holds after a program is executed. Then, we introduce the notion of a computation tree associated with a program. Finally, we define the desired/executed effects of a program as the conjunction of the guarded desired/executed effects of the leaves in its computation tree, respectively. The section ends with a formal definition of the effect preservation property of programs.

3.1 Effects of Primitive Actions

Primitive actions are atomic elements of the language $L(X)$. Their intentions, denoted $\text{effects}(S)$ express postconditions that should hold following the action. The desired/executed distinction does not apply to primitive actions, since their effects are both desired and executed. For the two primitives skip and fail, their effects derive from their intended semantics$^1$: $\text{effects}(\text{skip}) = \text{true}$, $\text{effects}(\text{fail}) = \text{false}$. For assignments, the effects are domain specific and developer provided. Clearly, we expect that developer provided effects are non-trivial, e.g. $\text{effects}(S) = \text{true}$.

Example 2 (Possible effects of primitive assignments in different domains).

- **Sets** – insert or delete an element $e$ from a list $x$: $\text{effects}(x := \text{insert}(x,e)) = (e \in x)$, and $\text{effects}(x := \text{delete}(x,e)) = (e \notin x)$.
- **Lists** – insert an $n$th element $e$ to a list $x$: $\text{effects}(x := \text{insert}(x,n,e)) = (e = \text{element}(x,n))$.
- **Trees** – insert an element $e$ to a tree $x$: $\text{effects}(x := \text{insert}(x,\text{path},e)) = (e = \text{element}(x,\text{path}))$.

The effects in the last example involve conditions that can be expressed in terms of the final values of the state variables, i.e., static conditions. In this paper we deal only with static effects, and most of the examples are taken from the domain of sets.

3.2 Effects of Sequences of Primitive Actions

Primitive actions modify the values of state variables. These modifications have to be considered in the account of the desired and executed (historical) effects, because repetitive modifications of a state variable might interfere. Therefore,

$^1$ Recall that $\text{fail}$ is the impossible action, i.e., rejection. Its semantics in Dijkstra’s guarded commands language [1, 6] states that “everything holds following a $\text{fail}$".
the computation of effects requires repeated application of the modifications to the state variables. The following examples demonstrate these notions in the set domain. They use the actions \( x := \text{insert}(x, e) \) and \( x := \text{delete}(x, e) \), for a set variable \( x \) and an element variable \( e \), with the effect formulae \( e \in x \) and \( e \notin x \), respectively.

**Example 3 (Desired Effects).** Consider the program:

\[
S(e_1, e_2, e_3) = (x := \text{insert}(x, e_1); x := \text{insert}(x, e_2); x := \text{delete}(x, e_3))
\]

The desired effects are: \( \text{effects}_D(S(e_1, e_2, e_3)) = e_1 \in x \wedge e_2 \in x \wedge e_3 \notin x \). Clearly, \( \text{effects}_D(S(e_1, e_2, e_3)) \) does not hold in case that \( e_1 = e_3 \) or \( e_2 = e_3 \).

**Example 4 (Executed (Historical) Effects).** Consider the program \( S \) above. The executed effects of \( S \) certainly include \( e_3 \notin x \). The insertion of \( e_2 \) occurs before the last deletion. Therefore, in terms of the final value of \( x \), its effect is \( e_2 \in \text{insert}(x, e_3) \), i.e., the last \text{delete} \ action must be reversed. The insertion of \( e_1 \) occurs before the insertion of \( e_2 \). Therefore, in terms of the final value of \( x \), its effect is \( e_1 \in \text{delete}(\text{insert}(x, e_3), e_2) \). Altogether we have: \( \langle e_1 \in \text{delete}(\text{insert}(x, e_3), e_2) \rangle \land \langle e_2 \in \text{insert}(x, e_3) \rangle \land \langle e_3 \notin x \rangle \). Yet, these executed effects fall short of handling the case where \( e_1 = e_2 \). The problem has to do with idempotent actions (like set insertion and deletion), where repetitions are redundant since \( \langle x \text{ op } a \rangle \text{ op } a = \langle x \text{ op } a \rangle \). If \( e_1 = e_2 \), the second insertion is a \text{skip} \ action, but in the executed effects account it is reversed by a non-\text{skip} \ action.

We handle sequences of redundant primitive actions by factoring out repetitions. In this example, the second action \( x := \text{insert}(x, e_2) \) is transformed into \( x := \text{insert}(x, e_2 - e_1) \), where the element subtraction stands for singleton set subtraction. Therefore, the executed effects are: \( \text{effects}_E(S(e_1, e_2, e_3)) = (e_1 \in \text{delete}(\text{insert}(x, e_3), e_2 - e_1)) \land (e_2 \in \text{insert}(x, e_3)) \land (e_3 \notin x) \)

**Definition 1 (Desired Effects).** For simple actions that involve a single state variable, the desired effects of a sequence are simply the conjunction of the effects of the primitive actions (as in the above example).

**Definition 2 (Executed Effects).** Let \( S \) be a sequence of primitive actions \( x_1 := A_1, \ldots, x_n := A_n \). The executed effects of \( S \) are obtained in two steps:

1. **Factorization:** Each action \( x_i := A_i \) is factored with respect to all previous actions \( x_j := A_j \) where \( x_i = x_j \) and \( j < i \). The exact factorization procedure is type dependent.
2. **Inductive definition:**
   - (a) For a primitive action \( U \), \( \text{effects}_E(U) = \text{effects}(U) \).
   - (b) \( \text{effects}_E(S; \text{skip}) = \text{effects}_E(S) \).
   - \( \text{effects}_E(S; \text{fail}) = \text{false} \).
   - \( \text{effects}_E(S; x := A) = \text{effects}(x := A) \land (\text{effects}_E(S) \setminus \{x/A^{-1}\}) \), with \( A^{-1} \) as inverse action of \( A \).

It can be shown that \( \text{effects}_E(S) \) of sequences of primitive actions are valid postconditions of any execution of \( S \). A formal proof requires introduction of a calculus for reasoning about imperative operations, such as Dijkstra’s guarded commands. The proof appears in the full paper [4].
3.3 Computation Paths and Effects

Compile-time effect preservation requires fine analysis of the computation paths of a program and their effects. The following examples demonstrate the weakness of effects that are assigned to a program as a whole, and the advantage of an effect preservation theory that relies on path sensitive effects.

Example 5 (Path Sensitive Effects).

The program
\[
S(\text{"A"}, \text{"B"}) = \\
\text{if } P \text{ then } \text{Car} := \text{insert}(	ext{Car}, \text{"A"}); \\
\text{if } Q \text{ then } \text{Car} := \text{insert}(	ext{Car}, \text{"B"})
\]

has four computation paths,

\[
\begin{align*}
\text{insert(Car,} \text{"B"}) & \text{ skip (\text{true})} \\
\text{skip} & \text{ skip (\text{false, false})}
\end{align*}
\]

with the desired effects: \(\text{"A"} \in \text{Car} \land \text{"B"} \in \text{Car}, \text{"A"} \in \text{Car} \land \text{true}, \text{"B"} \in \text{Car} \land \text{true}, \text{true}\) (the latter effect means that only \text{skip} is performed). Each formula corresponds to the effects of a possible sequence of primitive actions.

In order to define effects on paths of a computation, we use an auxiliary data structure termed the computation tree of a program. Each program is associated with a single finite computation tree, whose paths span all the possible evaluations of the conditions in the program. The nodes of the computation tree are labeled with primitive actions, and the arcs are labeled by conditions.

Definition 3 (Computation Tree). Let Node\((U)\) be a constructor for a node tree labeled with a primitive action \(U\), and addLeft(node, \(P, T\)) and addRight(node, \(P, T\)) methods that add a left or right child \(T\) to a node, with \(P\) as the arc condition. The computation tree Tree\((S)\) of a program \(S\) is inductively defined:

1. For a primitive action \(U\), Tree\((U) = \text{Node}(U)\).
2. \(\text{Tree}(S_1; S_2) = \text{for all leaves } l \text{ of Tree}(S_1) \text{ do: addLeft}(l, \text{true}, \text{Tree}(S_2))\).

\(\text{Tree}(\text{if } P \text{ then } S_1 \text{ else } S_2) = \text{root} = \text{Node}(\text{skip}), \text{addLeft}(\text{root}, P, S_1), \text{addRight}(\text{root}, \neg P, S_2)\).

Each node in a computation tree stands for a sequence of primitive actions given by the labels on the path that leads to the node. Further, each node is associated with a guard constraint that characterizes the initial condition for following the path (a pre-condition).

Example 6 (Guards and Sequences of a Computation Tree Nodes). Consider example 5. The four leaves are associated with the following sequences and guards:

<table>
<thead>
<tr>
<th>sequence</th>
<th>guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(Car, “A”); insert(Car, “B”)</td>
<td>(P \land Q {\text{Car/insert(Car,&quot;A&quot;)}})</td>
</tr>
<tr>
<td>insert(Car, “A”); skip(≡ insert(Car, “A”))</td>
<td>(P \land \neg Q {\text{Car/insert(Car,&quot;A&quot;)}})</td>
</tr>
<tr>
<td>skip; insert(Car, “B”) (≡ insert(Car, “B”))</td>
<td>(\neg P \land Q)</td>
</tr>
<tr>
<td>skip; skip(≡ skip)</td>
<td>(\neg P \land \neg Q)</td>
</tr>
</tbody>
</table>
Guards of nodes can be defined inductively, similarly to the definition of executed effects ([4]). Computation trees can be optimized by pruning branches that include internal fail nodes, and removing nodes whose guards are equivalent to false.

Each node in a computation tree is associated with the desired effects of its sequences. For example, the left most leaf in the computation tree in example 5 is associated with the desired effects “A” ∈ Car ˄ “B” ∈ Car. The guarded desired/executed effects of a node n are the implication conditions guard(n) ⇒ effectsD(n) and guard(n) ⇒ effectsE(n), respectively.

3.4 The Effect Preservation Property

The effects of a general composite program should take into account its different computation paths. For example, the program of example 5 cannot have as its effects simply the conjunction of the contradictory effects of its primitive actions.

Definition 4 (The Desired/Executed Effects). The desired/executed effects of a program S are given by the conjunction of the guarded desired/executed effects of the leaf nodes in the computation tree of S.

Following the intuitive discussion of effect preservation above, we define a program as effect preserving if its desired effects follow from its executed effects. That is, the history that followed a modification did not affect its effect:

Definition 5 (The Effect Preservation Property). A program S is effect preserving if effectsE(S) ⇒ effectsD(S).

Proposition 1. A program is effect preserving if and only if for each leaf node n of its computation tree, effectsE(n) ⇒ effectsD(S) is valid.

Example 7. The program from example 5 is effect preserving, since it can be easily verified that for each leaf node the last proposition holds. For example, for the left most leaf, [“A” ∈ delete(Car, “B”) ˄ “B” ∈ Car] ⇒ “A” ∈ Car ˄ “B” ∈ Car is valid.

4 The Effect Preservation Transformation

An effect preserving transformation turns an input program into an effect preserving one, by inserting tests that act as guards against violation of previous intentions. The two criteria for a good transformation are: (1) Minimize rejections caused by effect preservation and (2) Minimize the run-time overhead. The task of effect preservation can be carried out at run-time, or at compile-time, or in a combined mode. A run-time only transformation achieves the first criterion, since rejections are triggered only in case of past effect violation. However, the run-time overhead is maximal, since at every modification all past effects must be checked. A compile-time only transformation achieves the second criterion since all effect violations are pre-computed and interleaved within the program.
but the first criterion is not achieved since evaluation of effects requires run-time information.

In a combined compile-time run-time approach both criteria can be achieved using static analysis and partial evaluation of effects. The path sensitive effects characterized in the previous section guarantee that tested effects reflect the intentions of actions that are actually executed, and do not enforce effects of computation paths that are not followed. The static analysis in the presented transformation includes the construction of the computation tree with its path sensitive effects. Furthermore, in each tree node, the relevance between the aggregated path effects and the immediate primitive modification is pre-computed. This relevance singles out delta-conditions that must be checked, following the previous tests. This way, repeated time consuming evaluations of full path effects before or after modifications are saved. The delta-conditions provide a form of partial static evaluation of effects. The final evaluation of the delta-conditions is left to run-time. Run-time overhead is minimized by interleaving the tree navigation within the resulting program, and computation of delta-conditions in compile-time.

4.1 Delta-Conditions

Intuitively, executing an assignment, corresponding to some node $n$ of the tree, the effects can be preserved, if the condition $\text{effects}_D(n)$ is tested after $n$ has been executed. However, this is still costly, since the size of $\text{effects}_D(n)$ is proportional to the size of the program. Therefore, the run-time overhead is in the worst case $O(\text{size}(S)^2)$, for a program $S$, assuming that primitive effect tests take constant time, and that primitive actions have atomic effects. Unfortunately, this is the run-time overhead of run-time only effect preservation. The run-time overhead can be improved by minimizing the formula $\text{effects}_D(n)$ which is the idea behind delta-conditions.

Delta-conditions extract the possible interaction between aggregated path effects and an immediate primitive action. Consider, for example, the program $S(u, v) = (x := \text{insert}(x, u); x := \text{delete}(x, v))$. The desired effects of the 2nd assignment are: $u \in x \land v \notin x$. Again, static analysis of these effects leaves the delta-condition $u \neq v$ to be tested prior to the application of the second assignment.

**Definition 6 (Delta Conditions).** Let $n$ be an assignment node in a computation tree of a program, with an action label $x := f(\vec{e})$. If $n$ is the root node, then $\text{delta-condition}(n) = \text{true}$. Otherwise, $\text{delta-condition}(n)$ is any condition satisfying: $\text{effects}_D(\text{parent}(n)) \land \text{delta-condition}(n) \Rightarrow \text{effects}_D(\text{parent}(n))_{\{x/f(\vec{e})\}}$

Thus, delta-conditions guarantee that the desired past effects of the parent node hold after the assignment. Clearly, $\text{effects}_D(\text{parent}(n))_{\{x/f(\vec{e})\}}$ is a legal delta-condition($n$), but the worst. The best delta-condition($n$) is simply true, which indeed is the case, if the action of $n$ does not interfere with $\text{effects}_D(\text{parent}(n))$.  

109
Delta-conditions can be computed by a rough syntactic method that is based on a “no common state variables” consideration, as in the above example with a delta-condition \( u \neq v \). Or by a fine, domain dependent method, that exploits knowledge about the domain primitives, i.e., actions and predicates. Since delta-condition specifies a test for possible interference between an action and a constraint, its static evaluation can be performed with methods developed in the field of Integrity Constraint Checking [12, 5], where static analysis results minimal conditions that are left to be tested at run-time.

4.2 Effect Preservation Transformation

A given, possibly non effect preserving, program is transformed into a effect preserving one by interleaving the program code with commands for navigating the computation tree. Thus, precomputed delta-conditions are taken from the tree in a path sensitive manner and checked at run-time.

Algorithm 1 (Effect-Preservation with Delta-Conditions) The ref variable refers to the current position in the computation tree.

\[
\text{reviseProgram}( S, T ) = \\
\text{replace each } [\text{if } P \text{ then } S_1 \text{ else } S_2] \text{ in } S \text{ by:} \\
\text{(if } P \text{ then } ref:=\text{left}(ref); S_1 \text{ else } ref:=\text{right}(ref); S_2) \\
\text{for each primitive } U \text{ in } S \\
\text{if } U = \text{skip} \text{ or } U = \text{fail} \text{ then replace } U \text{ by: } (U; ref:=\text{left}(ref)) \\
\text{else} \\
\text{if for all } n \in \text{nodes}(U) \text{ delta-condition}(n) = \text{true} \text{ then} \\
\text{replace } U \text{ by: } (U; ref:=\text{left}(ref)) \\
\text{else} \\
\text{replace } U \text{ by:} \\
(\text{if delta-condition}(ref) \text{ then } U \text{ else } \text{fail}; ref:=\text{left}(ref))
\]

Algorithm 1 includes an additional static optimization step aimed at saving redundant tests of delta-conditions. Clearly, such tests at run-time are needed only if there exists at least one execution path in \( \text{nodes}(A) \) with a delta-condition different from \( \text{true} \). This optimization is especially important since we expect that effect violations would be rather rare, and the more common situation is that delta-conditions reduce to \( \text{true} \) at compile-time.

Finally, we present the main result showing that effect preservation is indeed achieved by the above algorithm, with run-time overhead that depends only on the delta-conditions. Since we believe that delta-conditions tend to be small and do not depend of the overall program size, then the run-time overhead is linear in the size of the program, instead of a quadratic overhead, in run-time only effect preservation.

Proposition 2 (Correctness and Efficiency of the Algorithm). For a program \( S \) in \( \mathcal{L}(X) \), \text{reviseProgram}(S) is an effect preserving program. The expected run-time overhead of \text{reviseProgram}(S) with respect to \text{Tree}(S) is smaller than \( \text{size}(S) \times \text{size(delta-condition)} \), i.e., \( \mathcal{O}(\text{size}(S)) \).
4.3 An Example of the Effect Preservation Transformations

Finally, we demonstrate the transformation of a non effect preserving program into an effect preserving one. Consider the following artificial, but comprehensive, program $S(e_1, e_2, e_3)$, where $P$ is an arbitrary condition:

\[
S(e_1, e_2, e_3) = \text{insert}(table, e_1); \\
\quad \text{if } P \text{ then insert}(table, e_2) \\
\quad \text{delete}(table, e_3)
\]

Clearly, $\text{delete}(table, e_3)$ might violate the effects of both insertions, depending on $P$ and the value of $e_3$. The desired effects and delta-conditions of nodes are as follows:

<table>
<thead>
<tr>
<th>node</th>
<th>effects $D$</th>
<th>delta-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e_1 \in table$</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>$e_1 \in table$</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>$e_1 \in table \land e_2 \in table$</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>$e_1 \in table$</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>$e_1 \in table \land e_2 \in table \land e_3 \notin table$</td>
<td>$e_1 \neq e_3 \land e_2 \neq e_3$</td>
</tr>
<tr>
<td>6</td>
<td>$e_1 \in table \land e_3 \notin table$</td>
<td>$e_1 \neq e_3$</td>
</tr>
</tbody>
</table>

The delta-conditions of the nodes 1, 2 and 4 reduce to $true$, since there is no previous action whose effects could be violated. The delta-condition of node 3 also reduces to $true$ due to domain specific set considerations (an insertion does not interfere with set membership effects). The delta-conditions of nodes 5 and 6 reduce to comparisons of the inserted and deleted elements. Applying Algorithm reviseProgram to $S$ returns the following program:

\[
S'(e_1, e_2, e_3) = \text{insert}(table, e_1); \quad \text{ref} := \text{childLeft}(\text{ref}); \\
\quad \text{if } P \text{ then } \\
\quad \quad \text{ref} := \text{childLeft}(\text{ref}); \\
\quad \quad \text{insert}(table, e_2); \quad \text{ref} := \text{childLeft}(\text{ref}); \\
\quad \text{else } \\
\quad \quad \text{ref} := \text{childRight}(\text{ref}); \\
\quad \quad \text{skip}; \quad \text{ref} := \text{childLeft}(\text{ref}); \\
\quad \quad \text{if } \text{delta-condition}(\text{ref}) \text{ then } \text{delete}(table, e_3) \text{ else fail}; \\
\quad \text{ref} := \text{childLeft}(\text{ref})
\]

5 Conclusion and Future Work

In this paper we defined the problem of effect preservation, and provided an efficient effect preservation algorithm with a linear run-time overhead. The algorithm relies on an exhaustive static analysis of a program, and on constraint reduction techniques. The static analysis includes path sensitive construction of effects, and reduction of interference conditions (delta-conditions). The resulting transformation is as expressive as the original program, but prevents effect violation.
The language considered in the paper is restricted to sequencing and conditional composite programs alone. In order to strengthen the approach, and enable partial transformations, it is necessary to extend the transformation to apply to a larger language of programs. In particular, it is important to add bounded loops to the language of programs.

Another extension is in the direction of more powerful dynamic effects, i.e., effects that allow simultaneous reference to initial and final values of state variables. Such effects are necessary for expressing evolution intentions such as "the value of a state variable can only grow".

Finally, we intend to apply our approach in multiple domains. For web based systems it is necessary to understand what are the characteristic effects of programs. For Rule Triggering Systems there is a need to study different modes of embedding an effect preservation algorithms within a system. For concurrent, reactive and mobile systems there is a need to study how effects of primitive actions should be defined. Our goal is to construct a generic open tool that can be extended in terms of language and effects, and can be applied to new domains.

References

An Active Approach to Model Management for Evolving Information Systems

Henrik Gustavsson1, Brian Lings2 and Bjorn Lundell1

1 University of Skovde, Department of Computer Science, P.O. Box 408, SE-541 28 Skovde, Sweden
{Henrik.Gustavsson, Bjorn.Lundell}@ida.his.se
http://www.his.se/ida/

2 University of Exeter, School of Engineering and Computer Science, Prince of Wales Road, Exeter EX44PT, UK
B.J.Lings@dcs.exeter.ac.uk

Abstract. It is desirable to be able to interchange design information between CASE tools. Such interchange facilitates cooperative development, helps in avoiding legacy problems when adopting new tools, and permits the use of different tools for different life-cycle activities. Exchanging model transformation information is particularly demanding in the context of cooperative maintenance of evolving systems. In this paper we suggest an approach using active transformation rules. We show how transformation rules can be expressed using a modest extension of the Object Constraint Language of the UML standard, and actively interpreted. The approach allows existing UML-based tools or repository systems to be readily extended to actively manage models in evolving information systems.

1 Introduction

In CASE tools in general, and in repository systems in particular, it is desirable to be able to interchange information between different tools in a toolset [4], [9]. Such interchange can facilitate cooperative development, through the exchange of design documents. It can also help in preserving design information when a new tool is adopted. Under some circumstances it may also permit the use of different tools for different life-cycle activities.

Many CASE tools today use mappings or design transformations to automate different tasks within an IS lifecycle design process. Few tools, however, support user definable design transformations, and those that do use proprietary languages for their definition [13]. Interchanging a set of models for the purpose of cooperative development can therefore be problematic. In particular, if the interchange is between tools that do not support the same set of design transformations, inconsistencies will be introduced. We argue, therefore, that design transformations must themselves be interchanged, and in such a way as to facilitate their use in an importing tool.
What is required is an architecture which will allow efficient execution of design transformations in a tool, and the export of both transformations themselves and details of their usage in a design, for example in transforming a conceptual model into a logical one. The set of interrelated design models could then be successfully manipulated using an importing tool, without it previously having been configured to handle the transformations involved. Such extensibility in the IS modelling area is the goal of the research reported here.

1.1 Transformations

In the area of meta modelling and repository systems, adoption of the UML [26], MOF [27] and XMI [25] standards has made it possible to interchange information and to guarantee extensibility in meta modelling systems [9]. A number of implementations of repository systems that currently support UML also support XMI interchange of data, including the France Telecom model repository tool [1] and the SPOOL design repository [19]. These projects support extensible, standards based meta models. In addition to supporting an extensible meta model, the Microsoft Repository [3] also supports the expression of transformations. It uses a model called the OTM (open transformation model) to define and store transformations. The main purpose of this is to support transformations in data warehouses, so the model for transformations is affected by its roots within the data warehousing field. However, in most other cases, mappings and transformations (or model management features [2]) used in tools connected to repositories are neither interchangeable nor extensible.

The use of proprietary languages for transformations inhibits interchange between tools. The Microsoft Repository uses a proprietary combination of SQL and OLE software to execute transformations. The DB-MAIN project [13] mainly uses a procedural language with a constraint language to define pre and post conditions for transformations. A logic-based language has been proposed elsewhere [23].

Many different uses have been found for design transformations (see [12], [13], [15], [16], [23]). Most of these sources, however, do not discuss how the modelling decisions leading up to a design transformation are to be represented. Nor do they discuss how to represent the information required to perform a design transformation - not all relevant information is directly present in a model (for example, required parameters from the user). Other systems, such as the DAIDA project, use a knowledge-based approach to design transformations. This allows sophisticated deductions to be made from the knowledge base, for instance which transformations were used in mapping between models [17].

The DB-MAIN project, however explores the use of design histories as a means to represent design decisions [14]. The model before a transformation is not maintained in its entirety, but modelling decisions are indirectly captured through state transitions. Using this ‘history’, a model can be wound back to the time before a certain
transformation was applied, so that the transformation can be reapplied with, for example, a different set of input parameters, resulting in a different destination model.

However, the use of design histories to achieve a higher degree of modelling transparency has a number of drawbacks. Firstly, a very sophisticated versioning system is required which supports multiple and branching histories. In addition, the system must be capable of inferring modelling decisions and properties from these histories. Secondly, it may in some cases be difficult to tell whether a certain change in a model is due to the application of a transformation or direct user intervention. This is because design histories serve a dual purpose, in that they support both historical information about previous versions of a model and an indirect representation of those modelling decisions. Knowledge based approaches, on the other hand, require sophisticated inference engines, and use proprietary languages and representation techniques which thus make them unsuitable for a scenario where interoperability between different tools is of prime importance.

In an earlier paper [10] we have suggested an approach by which design transformations can be freely interchanged between tools. Transformations are represented using a conservative extension of the OCL language. The approach is independent of proprietary languages and techniques. Further, since more and more tools are adding support for OCL [24], it will become increasingly straightforward to adopt this approach in existing tools. The approach in that paper, however, has a significant drawback: rule execution is independent of the event that triggers a transformation. This problem makes the approach less powerful and significantly less suitable for maintenance. In this paper we suggest how to remove this drawback, developing the OCL language to support reactive behaviour suitable for co-operative management of model evolution.

1.2 OCL as a Conceptual Language for Metamodelling Constraints and Actions

Active databases have been proposed, over the years, for a wide variety of different tasks in many different application areas. In repository systems, active databases have been used to automate common tasks such as general model processing [18] and change management [8]. Such proposals, however, have relied in large part on platform dependent models or languages. This would impede successful application of the techniques to systems which do not share a common platform.

The Object Constraint Language [20] is part of the UML standard [26]. Amongst other things, it is used to introduce pre and post conditions to, and to place guards on, methods. The language is platform independent, declarative and efficient for querying and navigating object-oriented data. Even though the OCL language, as a conceptual language, is intended for object-oriented modelling, it can be used with other forms of modelling. It can, for instance, be used to specify constraints on SQL databases.
This versatility makes OCL a good language for platform independent specification of conditions in repository systems and meta models.

An extension to the OCL language to support actions [21] has been put forward, but the proposal stopped short of suggesting that such actions be executed using an OCL interpreter. However, to achieve a high degree of modelling transparency [5], the simultaneous update of interrelated parts of dependent models is needed. The most common way to achieve this is by transforming a model into one which reflects the required changes – that is, active behaviour.

1.3 A Novel Approach to Modelling Transparency

This project takes a novel approach to increasing modelling transparency in that transformation patterns representing modelling decisions made by a user are represented explicitly as part of the modelling information stored in a repository. The repository thus directly represents transformations, the parameters needed to perform each selected instance of a transformation, and the results of such transformations (in the form of updated models). In order to completely support the desired increase in modelling transparency, the objects that result from a transformation are also connected to the source objects using ordinary associations. This allows a connection to be navigated, for instance to allow a tool user to find the set of relational tables that result from the transformation of an entity type.

In a previous paper we have shown that transformation rules can be expressed using a conservative extension of OCL. In this paper we show how a tool can be made to react to state changes in its meta modelling repository through the addition of events to transformation rules. A further modest extension to the OCL language is proposed, to support context variables to receive parameters from event occurrences. We have tested the ideas through the implementation of an active repository system with an event detector and rule manager suitable for model management in a UML environment. The proof of principle system used to test the examples used in the paper is available on request (henke@ida.his.se).

2 Overview of Approach

Although the approach outlined is designed to be generally applicable for multi-model management, our chosen application context is CASE data interchange for cooperative design. We believe it is beneficial in such contexts to support the active interchange of design transformations.

In the general approach, each design transformation is represented by a set of rules which, given specific model and parameter information, can be used to bring about that transformation. This offers better support for the incremental update of models typical of cooperative design. The OCL language has been chosen to represent design
transformations\(^1\). However, OCL traditionally supports neither updates nor active behaviour. Other authors have suggested extensions to OCL for introducing active behaviour [21]. In our work, we extend the OCL language and its interpreter to allow the expression of active behaviour with update. In general, the fundamental issues to be addressed in moving to active behaviour are [28]: event specification and detection, access to context information by rules, and access to state transition information.

The proposed extension uses an ECA (Event Condition Action) format for rules to provide support for active design transformations. The use of ECA rules gives a number of benefits over a condition action based approach [6]. Firstly, events and conditions play different roles in the system, allowing the repository to react directly to state changes in the context in which they occur. Making the event explicit thus allows finer grained control over when execution occurs. This increases flexibility in execution semantics. This latter is important, since this project strives to be as platform independent as possible so that the ideas in the approach can be adapted to fit existing tool environments. There is also a performance benefit in that by using an event to trigger a condition check, fewer conditions have to be evaluated for the same database state.

2.1 Meta Model Extensions for Active Behaviour

There are different ways in which reactive behaviour can be supported, each placing different requirements on the rule scheduler and event detection mechanism. One very important consideration for this project is the ease with which the techniques can be incorporated into existing CASE technology. The suggested methodologies should thus be simple enough to be easily implemented in existing tools or tool infrastructures.

It has only been found necessary to include primitive database events in the rule system; no transformations so far studied have required the introduction of temporal or composite events. Hence, the rule system proposed only recognises events that occur when objects are inserted into the model, when objects are updated in the model, and when objects are deleted from the model. However, it has been found useful to distinguish a separate set of event types for the modification of collections, i.e. the creation, deletion and update of associations between objects in models.

In order to support the specification of active OCL rules, it has been necessary to extend the OCL rule language beyond that suggested in our previous work [10] by the addition of a list of events that can trigger a transformation rule:

\begin{verbatim}
Contextclass: <Context class specification>
Event: <event specification>
Condition: <condition specification>
Declaration: <declaration specification>
Action: <action specification>
\end{verbatim}

\(^1\) An explanation of the rationale behind the choice of this language can be found in [8].
In order to be able to represent transformations directly within the models, some kind of meta modelling support is necessary. In order to achieve a high degree of interoperability this model has been kept as simple and as generic as possible, in contrast with, for example, the Microsoft Repository approach [3] that models transformations with proprietary and domain specific structures. A number of superclasses have been proposed previously [10] for this purpose in a non-active environment; these can be inherited by the other meta model classes. The only extension required is the addition of an event property to the rule class.

2.2 Event Types and Context Variables

The behaviour of an active set of models may be heavily dependent on the cascading of rules (if multiple levels of models are used). It has been found useful to differentiate between primary events and secondary events. Primary events occur as a direct result of user interaction, for example check-in of a model or direct modification of a modelling object. Secondary events occur when the model contents are modified by a transformation rule. By differentiating between the six primary event types and the six secondary event types, it is possible to perform different actions depending on whether an update came from inside or outside the repository. This gives increased control over rule cascades since a rule, through its triggering event, has knowledge of whether it is executing as the consequence of a cascaded event or as a direct result of an update by a tool user. For example, it allows increased control over cycles in rule cascades.

In active databases, context variables (event parameters [6]) contain information about the state of the data before and after updates, so that rule behaviour can be expressed in terms of state change. In order to support this type of behaviour for the active set of models it is necessary to extend OCL to allow the use of context variables in transformation rules. In earlier work [10], we introduced updates to the OCL language. In particular, the aliasing mechanism present in the OCL standard definition was used to allow new objects to be referenced, and thus be modified in different ways. In order to limit the scope of the necessary changes to an OCL interpreter, it would be beneficial to further use the aliasing mechanism in supplying context variables.

In the prototype implementation, user updates to the repository are taken to be atomic actions. Transformation rules are therefore executed only after the metadata updates corresponding to these atomic actions. We have introduced two new global aliases for use as context variables. They work in the same way as the “SELF” alias, which identifies the context object. The “OLD” alias identifies the object state before the triggering update, and the “NEW” alias the object state after the triggering update. These context variable aliases have slightly different meanings depending on the type of event triggering the rule using the aliases (see table 1).
Table 1. Event types and context variable descriptions.

<table>
<thead>
<tr>
<th>Event</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>NEW</td>
<td>Provides access to the values of the newly inserted object. Since the object is new, there is no “OLD” variable.</td>
</tr>
<tr>
<td>Delete</td>
<td>OLD</td>
<td>Provides access to the object as it was before deletion.</td>
</tr>
<tr>
<td>Update</td>
<td>OLD</td>
<td>Provide pointers to the information before and (respectively) after the update.</td>
</tr>
<tr>
<td>Collection Insert</td>
<td>NEW</td>
<td>Provides access to the object which has been added to the collection.</td>
</tr>
<tr>
<td>Collection Delete</td>
<td>OLD</td>
<td>Provides access to the object which is going to be removed from the collection.</td>
</tr>
<tr>
<td>Collection Update</td>
<td>OLD</td>
<td>Only occurs for 0..1 or 1..1 cardinalities. In this case, the “OLD” alias provides access to the object in the collection before the update, and the “NEW” alias provides access to the object in the collection after the update.</td>
</tr>
</tbody>
</table>

Since the OCL language displays collections and properties in the same way, but since the actual handling of associations and properties must account for conceptual differences, this separation simplifies the management of these two types of event.

A specific update collection event type is also introduced for collections that contain at most one object. When this is changed so that it identifies some other object, an update collection event is fired, indicating an update of an existing collection. The main difference between collection events and property events is that in property events, the object in the new and old variables is of the context class type. In contrast, for collection events, the new and old variables refer to the object that was added/removed from a collection, which may not be of the same type as the context class.

3 Example of Approach

To allow easy comparison with existing approaches, an example rule set has been created which demonstrates some important types of transformation handled in the related work cited earlier (section 1.1). The set of example transformation rules and the example meta model (see figure 1) are not intended to be complete, but are designed to show the characteristics of an active approach. We have successfully captured the design transformations used in the real-world example cited in [22], where a company was unable to adopt CASE technology because none of the current generation of CASE-tools evaluated allowed import of the existing models.

The example in this paper uses a simplified version of a well known E-R notation [11]. It is not intended to suggest new or more advanced transformations than those used to demonstrate alternative approaches. However, through the use events the
The goal in the example is to transform an ER Entity to a relational model table, and transform ER attributes to table attributes while retaining the attribute names and the key attributes of the original entity. This simple transformation requires two separate rules, one that transforms the entities and one that transforms the attributes. An important feature of this rule set is that it can handle incremental updates of the attributes; when a new attribute is added to an existing entity, the corresponding attributes are generated automatically. Since supporting incremental updates of other modelling objects would require more rules, this example only supports incremental updates for the ER attributes. Rules expressed in logic for performing the inverse of this transformation were suggested in [23].

**Contextclass:** EREntity  
**Event:** INSERT  
**Declaration:** RelTable T1  
**Condition:** Implementedby->isempty  
**Action:** T1.create; T1.name:=self.name; T1.Implements:=self

**Contextclass:** ERAAttribute  
**Event:** INSERT  
**Declaration:** Relattribute RA  
**Condition:** IMPLEMENTEDBYRA->isempty  
**Action:** RA.CREATE; SELF.implementedbyra:=RA; RA.attname:=self.attname; RA.containedin :=

---

![Diagram](image-url)  
**Fig. 1.** A simple example meta model without transformation metadata included
self.Containedinentity.implementedby; RA.Pkstate :=
self.Pkstate

One well-known transformation suggested in another related paper [13] is the transformation of relationships into foreign keys. This transformation is performed by the rule listed below. Another rule would be required to deal with the case when the cardinalities are reversed. Further rules could also be added if options other than generating a foreign key are to be handled.

**Contextclass:** ERRelationship  
**Event:** INSERT  
**Declaration:** RELFK FK  
**Condition:** Fromcard="1" and Tocard="N" and implementedbyfk->isempty  
**Action:** FK.create; FK.relname:=self.name; FK.implementsrel:=self; FK.fromtable:=self.toentity.implementedby; FK.totable:=self.fromentity.implementedby

**Contextclass:** RELATTRIBUTE  
**Event:** INSERT  
**Declaration:** RELattribute RA  
**Condition:** Referencedby->isempty and containedin.fromFK->notempty and PKstate="1"  
**Action:** containedin.fromfk->iterate(FK| RA.create; RA.containedin:=fk.fromtable; RA.PKstate:="0"; RA.attname:=self.attname; RA.createdby:=FK; RA.references:=self)

The example rule set also contains rules to perform cascade update and cascade delete of ER attributes. These rules will allow the tool user to delete attributes or to rename ER attribute names without having to regenerate the whole relational model. Using events significantly reduces the number of conditions to be evaluated to support this type of behaviour.

**Contextclass:** ERATTRIBUTE  
**Event:** DELETE  
**Condition:** IMPLEMENTEDBYRA->notempty  
**Action:** IMPLEMENTEDBYRA.REFERENCEDBY->iterate(RA | RA.delete); IMPLEMENTEDBYRA.DELETE  
**Contextclass:** ERATTRIBUTE  
**Event:** UPDATE ATTNAME  
**Condition:** IMPLEMENTEDBYRA->notempty  
**Action:** IMPLEMENTEDBYRA.REFERENCEDBY->iterate(RA | RA.attname:=self.attname); IMPLEMENTEDBYRA.attname := self.attname

### 4 Analysis and Discussion

We have proposed an approach to concisely representing design transformations through the use of reactive behaviour, implemented using active rules expressed in a
conservative\textsuperscript{2} extension of the OCL language. The approach allows a compliant tool to use an enhanced set of design transformations for a set of models without the tool itself having to be modified. Since the language used to represent transformations is based on standardized languages and representation techniques (UML/MOF), with standardised ways to interchange models (XMI), existing tools or repository systems can be extended for compliance with relative ease. The approach has advantages over proposals which use proprietary languages, such as [15] and [23]. Examples have been presented to illustrate efficient, incremental update of interchanged models is supported using the approach.

There are a number of implications for tool vendors wishing to benefit from the proposed approach; we perceive three different classes of tool that may utilize an imported model or export a tool-specific model.

Tools in the first class have no support for alternative design transformations, their representation or interchange. Design transformation information has no meaning to these tools, and will be ignored. Such a tool can only usefully export a model; any importing tool must be provided with the transformations used. Building the corresponding model connections would be a non-trivial task.

Tools in the second class allow a user to select from a set of alternative design transformations. For collaboration purposes, these selections should be interchangeable using standardised export formats. Translators must be written for this purpose. Any information that documents a transformation alien to the tool has no meaning to it. The importing translator for the tool has to distinguish the associations between ordinary modelling objects from those between modelling objects and transformation pattern objects, which behave differently. As with class 1, any tool importing from a class 2 tool must be provided with the transformations used.

Tools in the final (class 3) class support all of the different types of information. Such a tool has to distinguish between the various association types only by looking at the meta information. From our earlier analysis, this final class is likely to require support for global navigation between the different modelling objects, which together form a complete set of models for a given domain.

To turn a class 2 tool into a class 3 tool, it is only necessary to develop an extension of the internal transformation engine to interpret OCL actions. The prototype implementation has shown this to be a modest extension to existing OCL interpreters.

\footnote{2 In the sense that existing tools can be readily updated to support the extensions.}
References


### Author Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balaban, Mira</td>
<td>102</td>
</tr>
<tr>
<td>Carnduff, Tom W.</td>
<td>89</td>
</tr>
<tr>
<td>Chen, Victor</td>
<td>39</td>
</tr>
<tr>
<td>Combi, Carlo</td>
<td>13</td>
</tr>
<tr>
<td>Diep, Cuong</td>
<td>39</td>
</tr>
<tr>
<td>Domínguez, Eladio</td>
<td>64</td>
</tr>
<tr>
<td>Edelweiss, Nina</td>
<td>1</td>
</tr>
<tr>
<td>Estrella, Florida</td>
<td>76</td>
</tr>
<tr>
<td>Galante, Renata de Matos</td>
<td>1</td>
</tr>
<tr>
<td>Goonetillake, Jeevani S.</td>
<td>89</td>
</tr>
<tr>
<td>Guan, De Ming</td>
<td>39</td>
</tr>
<tr>
<td>Gustavsson, Henrik</td>
<td>113</td>
</tr>
<tr>
<td>Jurk, Steffen</td>
<td>102</td>
</tr>
<tr>
<td>Kane, Bintou</td>
<td>39</td>
</tr>
<tr>
<td>Kovacs, Zsolt</td>
<td>76</td>
</tr>
<tr>
<td>Le Goff, Jean-Marie</td>
<td>76</td>
</tr>
<tr>
<td>Lings, Brian</td>
<td>113</td>
</tr>
<tr>
<td>Lloret, Jorge</td>
<td>64</td>
</tr>
<tr>
<td>Look, Jennifer</td>
<td>39</td>
</tr>
<tr>
<td>Lundell, Bjorn</td>
<td>113</td>
</tr>
<tr>
<td>Maedche, Alexander</td>
<td>51</td>
</tr>
<tr>
<td>McClatchey, Richard</td>
<td>76</td>
</tr>
<tr>
<td>Pozzi, Giuseppe</td>
<td>13</td>
</tr>
<tr>
<td>Rundensteiner, Elke A.</td>
<td>39</td>
</tr>
<tr>
<td>Santos, Clesio Saraiva dos</td>
<td>1</td>
</tr>
<tr>
<td>Solomonides, Tony</td>
<td>76</td>
</tr>
<tr>
<td>Stojanovic, Ljiljana</td>
<td>51</td>
</tr>
<tr>
<td>Stojanovic, Nenad</td>
<td>51</td>
</tr>
<tr>
<td>Su, Hong</td>
<td>39</td>
</tr>
<tr>
<td>Toth, Norbert</td>
<td>76</td>
</tr>
<tr>
<td>Wang, Fusheng</td>
<td>26</td>
</tr>
<tr>
<td>Zaniolo, Carlo</td>
<td>26</td>
</tr>
<tr>
<td>Zapata, Maria Antonia</td>
<td>64</td>
</tr>
</tbody>
</table>