Introducing Custom Language Extensions to SQL:1999

Jernej Kovse, Wolfgang Mahnke

Department of Computer Science
University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{kovse,mahnke}@informatik.uni-kl.de

Abstract. Even though SQL has become widely accepted as a language for implementing relational database schemas and querying data, there are cases where its users experience the need for new language abstractions which would allow them to express data modeling and querying solutions in a clearer and simpler manner. This paper describes the idea of language extensions to SQL:1999 that come in form of independent packages and may be implemented by different vendors. A translator system uses information imported from the packages to translate the statements containing the extensions into SQL:1999-compliant statements.

1 Motivation

Over years, SQL has become a widely accepted language for implementing relational database schemas and querying data, irrespective of application domain. Its usage potential reaches well beyond simple database querying, which is often reflected by proposals aiming at seamless integration and access to heterogeneous data sources and applications via an SQL interface (e.g. SQL/MED [9]). The new ANSI/ISO/IEC standard SQL:1999 [7] supersedes the previous SQL-92 standard and introduces support for the execution of application logic in database servers as well as object-relational extensions to SQL.

A common problem for SQL users is the limitation of its syntax and semantics. There is, as we think, a substantial set of problems, developers typically consider at the application and not at the database level, simply because of lack of appropriate constructs:

- Multiple inheritance of complex structured types and tables is not supported in SQL:1999. There is a number of arguments supporting the elimination of multiple inheritance from a language, such as naming conflicts, repeated inheritance, programmer's temptation to apply multiple inheritance where containment would be the better choice, etc. [23]. Despite this fact, we take the stand that in certain designs, multiple inheritance allows a straightforward implementation of real-world problems (see [14] for examples) and promotes code reuse to the extent otherwise difficult to achieve. An obvious problem caused by the lack of multiple inheritance in SQL:1999 is that even some standardized metamodels defined using multiple inheritance (e.g. OMG's UML metamodel [17]) cannot be directly mapped onto type and table hierarchies in database schemas.

- Zhang [24] notes that current object-relational databases fail to support relationships that contain extended semantics. As a solution, she proposes an extension to SQL called Orient-SQL to allow precise specification of relationship's additional *structural* and *operational* properties. For example, relationship's participants, their superordinated or subordinated roles in the relationships and their cardinalities can be defined using structural properties. Behavior at the events of selecting, inserting, and deleting the relationship's participants in the database can be defined using operational properties.
- SQL:1999 fails to fully support modular implementation of database schemas [13] that would enable reuse of schema parts, assembly of off-the-shelf schema modules and increase schema quality by encouraging unit tests on interrelated structural (type and table definitions) and behavioral (triggers, user-defined routines) elements of a module.
- Preference queries [12] that have to be answered cooperatively by treating preferences as soft constraints are not supported in SQL:1999.

Motivated by the above examples, this paper presents our *X-Translate system*, which allows the users to extend SQL:1999 with general as well as domain-specific language constructs. The extensions defining the constructs are delivered in form of packages which can be imported into the system. The system comes with a translator that invokes methods defined by the extensions to translate statements containing custom language constructs onto SQL:1999 statements.

Sect. 2 will focus on the example of extending SQL:1999 to support modular schema implementation in detail. Sect. 3 presents the X-Translate system architecture. A list of open issues is discussed in Sect. 4, while Sect. 5 gives an overview of related work. In Sect. 6, we conclude our work and present some ideas for future work related to our approach.

2 Modularity in Schema Design

Even though a modular or even component-based design is state-of-the-art in application development ([1], [5]), there is a lack of such concepts in SQL:1999. In SQL:1999, there are only flat schemas and server modules [8], which mainly contain user-defined functions and do not deal with other schema elements. Both concepts do not offer interfaces or explicitly defined relationships between them (see [13], [15] for details). Although concepts like DataBlades or Cartridges offered by some ORDBMS vendors can be used to group schema elements, these concepts serve merely as initialization units that place their elements in a schema. Nevertheless, grouping schema elements together in nestable modules, offering interfaces to these modules and supporting relationships between the modules would make the benefits common to component-based application design also possible at the database schema level. Particularly w.r.t. the object-relational features of SQL:1999, like user-defined routines (UDR) and user-defined types (UDT), this would be extremely beneficial. The object-relational features of SQL:1999 allow to map much more functionality to the database schema which, in turn, leads to a more complex and time-consuming schema design. The main advantages of a modular schema design are in detail:

- *Reuse of parts of the schema.* By grouping semantically interrelated schema elements in modules, the elements can easily be reused in other schemas. For example, developing an XML data type and UDRs for managing XML documents inside the database, is a cost- and time-intensive task. Moreover, this functionality is probably needed by many different schemas. By grouping the schema elements for managing XML documents in a single schema module, they can easily be reused by other schemas. Without the concept of modularity, different schema elements, like UDTs and UDRs, are hidden in the schema making it unclear which elements are actually needed for a given functionality.
- *Easy and rapid schema design by assembling off-the-shelf modules.* If there is a sufficient number of (off-the-shelf) schema modules, new schemas can be developed mainly by combining existing schema modules. This does not only reduce development costs, but also decreases time-to-market.
- *Quality and robustness of a schema.* By (re)using high-quality, well-tested schema modules, both quality and robustness of a schema increase. Because most testing can be done on a small excerpt of a schema (the schema module), testing is much simpler compared to testing the whole schema at a time. The complexity of object-relational technology, especially when considering UDRs, makes testing necessary.
- *Exchange and extension of parts of the schema.* Using interfaces and, thereby, achieving *information hiding*, schema modules implementing the same interface can be replaced in a schema. This allows the optimization of schema parts and the inclusion of new code into an existing schema. Furthermore, the schema can be extended with new functionality by replacing an old schema module with a new one offering more functionality, but still complying with the old interface. We have to mention that most schema modules have a persistent state, that is, represented by tuples of a table. Exchanging stateful schema modules in a running system requires that the state of the old schema module is transferred to the new one. Nevertheless, exchanging schema modules seems to be a promising way to handle schema evolution.
- *Structural and distributed schema design.* Modularity allows structural design by dividing different tasks of the schema in separate schema modules and only define interfaces and relationships between them. The schema modules can be developed independently of each other by distributed groups of programmers.
- *Continued, component-based design.* Nowadays, a component-based development of applications is often blurred as far as the data storage component is concerned. The application components are separated at the application layer, but use a global database schema for their persistent data with overlapping parts. If each application component offers its own schema module and only well-defined relationships between the schema modules are used, the separation and isolation continues at the schema level.

The main objective of schema modularity is to control the dependencies between schema elements of separate schema modules. By schema elements we mean the modeling elements offered by SQL:1999, like tables, triggers, UDRs, UDTs, etc. The dependencies may emerge from a foreign key definition between two tables, but also a

UDR call or a trigger definition can lead to a dependency. In [15], a description and classification of the different dependencies is given. Using narrow interfaces and defining relationships between separate schema modules, the number of possible dependencies is restricted and other dependencies (unknown to the DBMS) are prohibited.

In [13], a framework for a modular schema design is introduced. We do not want to present the framework in detail, an excerpt of this framework may illustrate the idea of modular extensions of SQL:1999. The framework consists of different kinds of schema modules. In this paper, we will focus on one kind, the schema component. A schema component can contain all kinds of schema elements and can exist independently of other schema modules. It is important to know that, unlike the DDL statements of SQL:1999, there has to be a declaration of a schema component apart from the creation of a schema component. This is necessary, because a schema component may be used more than once in a schema, e.g., there may be several instances of a user management schema component, managing different kinds of users. A schema component can implement several interfaces as well as use interfaces of other schema components. Of course, only schema elements defined at the interface are visible outside the schema component. The *schema component declaration* specifies which interfaces are implemented by the schema component and which interfaces are required by the component. Concrete schema components used behind the interfaces are specified at the deployment of a component.

```
DECLARE INTERFACE BookManagementInterface
  TYPE bookT AS ( title CHARACTER(20),
                 ...)
  NOT FINAL
    METHOD getTOC() RETURNS CLOB,
    METHOD addChapter(chap Chapter, number INTEGER)
    ...;
  TYPE ChapterT AS (...) NOT FINAL;
  TABLE book OF bookT;
END INTERFACE;

DECLARE COMPONENT BookManagement
  IMPLEMENTING BookManagementInterface;
  DECLARE TYPE bookT AS ( title CHARACTER(20) NOT NULL,
                                ...)
     NOT FINAL
     METHOD getTOC() RETURNS CLOB,
     METHOD addChapter(chap Chapter, number INTEGER)
     ...;
  DECLARE TYPE chapterT AS ...;
  DECLARE TABLE book OF bookT
     title WITH OPTIONS NOT NULL;
  DECLARE TABLE chapter OF chapterT;
  ...
  DECLARE METHOD addChapter(chap Chapter, number INTEGER)
     FOR bookT
     BEGIN INSERT INTO chapter ....;
     END;
END COMPONENT;

CREATE COMPONENT bm OF BookManagement;
```

Fig. 1. Extended SQL for schema modularity

In Fig. 1, a small example for a DDL statement of a schema component declaration, called `BookManagement`, is given (lines 12-29). Line 13 defines which interfaces are implemented by `BookManagement` (in this example, there is only one interface, called `BookManagementInterface`, declared by lines 1-10). Note that in the example no interfaces and, as a consequence, no other schema modules are needed. Since `BookManagement` implements `BookManagementInterface`, all elements of the interface have to be part of `BookManagement`, like the structured types `chapterT` and `bookT`, supplemented with a constraint (`NOT NULL` in line 14). In addition to the typed table `book`, specified at the interface, `BookManagement` contains another table `chapter` and the implementations of the methods defined in the structured type (lines 25-28). After the schema component is instantiated (line 31), only the elements offered by the interface are accessible outside the schema component. That is, the table `chapter` is not accessible from outside the schema component, it can only be manipulated by method calls (e.g. `addChapter` of the `bookT`). The schema component has its own namespace and the elements are accessible only via this namespace.

3 X-Translate System Architecture

In this section, we introduce our approach to supporting different language extensions for SQL:1999, called the *X-Translate System*. The main goal of such a system is that different language extensions can be integrated into the system and, thereby, a new language containing the extensions is provided. Such an extended language is called *XSQL* (eXtended SQL). Note that XSQL does not mean a fixed combination of language extensions, but the combination specified by the currently installed language extensions of the system. Of course, not all language extensions can work together, e.g., if they use overlapping keywords. The system should be open to define new language extensions; it should be easy to define them and they should not invalidate existing legacy code that already contains SQL statements.

Certainly, we do not want to build a new DBMS from scratch, but translate XSQL to SQL:1999 and use it with an existing ORDBMS. We are quite sure that using query translation and the large expressive power of SQL:1999, most language extensions can be mapped to SQL:1999. Due to the fact that no ORDBMS vendor directly supports the exact syntax of SQL:1999, we also need a mapping from SQL:1999 to different SQL dialects. Because there is no big gap between SQL:1999 and the functionality offered by the ORDBMS vendors, this can easily be done. There are already approaches addressing this problem (see [25] for example), however, it proves possible to apply our approach successively to translate a SQL:1999 statement to a certain SQL dialect.

In Fig. 2, we illustrate the general architecture of the X-Translate system. It consists of a *translator*, an *extension directory* and an *extended system catalog*. Language extensions are defined in form of *extension packages*. Several extension packages can be imported into the system simultaneously. Each of them contains *grammar extensions*, *metamodel extensions* (to manage the metadata of the language extension), and *translation rules*, which describe how the extended grammar should be translated. The system manages the information related to the extensions within a single *extension direc-*

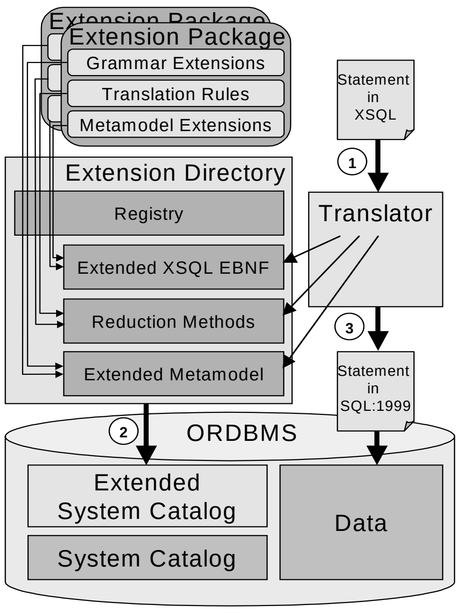

Fig. 2. General architecture of X-Translate

tory. This directory uses a *registry* to keep track of the imported packages. The grammar extensions are included in the *extended XSQL EBNF* (Extended Backus-Naur Form) and the metamodel extensions in the *extended metamodel*. The translation rules are used to generate the *reduction methods* of the system. To store the data of the extended metamodel, an *extended system catalog* is built on top of the SQL:1999 information schema. It is able to include the metadata of the extension packages and additional functionality to manage this metadata. For example, using schema modularity requires metadata for the declared schema modules and functionality to instantiate a schema module.

To translate an XSQL statement, the system works in the following way: The translator takes the XSQL statement as input (1). The statement is parsed utilizing the extended XSQL EBNF. The reduction methods are used to transform the XSQL statement to an SQL:1999 statement (3). Applying the reduction methods often requires accessing metadata stored in the extended system catalog (2). The translated statement itself can also access or manipulate data in the extended system catalog. The following sections will describe the architecture of the X-Translate system in detail.

3.1 Extension Packages

Extension packages are used to provide information needed by the system to support the use of new language constructs. This information comes in form of *metamodel extensions*, *grammar extensions*, and *definitions of translation rules*. A major challenge for extension packages is that they have to define the information in a generic way. This

```
1.  <SQL schema definition statement> ::= !! All alternatives from SQL:1999
2.                                          | <SQL component declaration>
3.
4.  <SQL component declaration> ::= DECLARE COMPONENT <component declaration name>
5.                                 .....
6.                                 END COMPONENT
```

EBNF extensions

```
7.  <SQL component declaration> =>
8.  CALL declareComponent(<component declaration name>, ...);
9.  ...
10. CALL addToComponentDeclaration(
11.                  <component declaration name>,<component declaration element>)
```

Translation rules

```
12. MAP class schema component TO TABLE schema_component;
13. CREATE TYPE namespace (.....);
14. CREATE INTERNAL TABLE namemapping(realname SQL_IDENTIFIER NOT NULL,
15.                                         namesp namespace NOT NULL);
16. CREATE INTERNAL FUNCTION declareComponent(name ...)
17.   BEGIN ...
18.     ... INSERT INTO schema_component VALUES (name, ....); ...
19.   END;
```

Metamodel extensions (extended system catalog hints)

Fig. 3. Excerpt of an extension package introducing schema modularity to SQL:1999

means that the user should be given the possibility of tailoring the extensions during the import process to avoid possible conflicts with already imported packages:

- Metamodel extensions: These extensions are used to add additional constructs to the metamodel. During the import the user has to decide where in the metamodel a construct is to be inserted and what its relationships to other constructs are.
- Grammar extensions: Class implementations for new EBNF symbols are inserted into the existing class hierarchy.
- Definitions of translation rules: The rules are used to obtain the default implementation of the reduction method which reduces a high-level XSQL language construct to lower level constructs that are either directly supported by SQL:1999 or need to pass additional reduction steps. The default implementation of the reduction method as generated from the rules can afterwards be tailored by the user.

In Fig. 3, an excerpt from the extension package that introduces schema modularity extensions (as described in Sect. 2) to SQL:1999 is shown. The grammar extensions are embedded into the SQL:1999 EBNF (see lines 1-6). Translation rules have to be specified for DDL (lines 7-11) and DML statements. It is useful to encapsulate DDL statements into UDR calls to get adequate error messages. Translating a component declaration statement directly into an insert statement for the corresponding extended system catalog table would lead to an incomprehensible error message. DML statements, in contrast, are directly translated to modified DML statements. The metamodel extensions consist of information of how to upgrade the original metamodel and the extended system catalog, storing the data of the metamodel. We use the XML-based Metadata Interchange (XMI) to specify transformations on the metamodel [10] (not shown in Fig. 3 due to space restrictions, the transformed metamodel is shown in Fig. 4). The extended system catalog is modified either by hints based on the metamodel (line 12, simply mapping a class of the metamodel to a table), or by concrete DDL statements (lines 13-19). Most of the constructs are marked as `INTERNAL`, as they have no corresponding class in the metamodel. For example, to support schema modularity, a table containing the mapping information of a virtual schema element with a namespace to the internal schema element without namespace (lines 14-15) is needed.

3.2 Extension Directory

Although language extensions originate from various independent packages, they are merged to form a single-level *extension directory*. Thus, the packages can be viewed as a means for gradually extending the directory. It is the extension directory that is used by the translator and not the packages. Using the packages directly in the translation process would lead to the following problems.

- Extensions introduced from separate packages may (i) define overlapping semantics, but use distinct syntactical notation or (ii) define distinct semantics, but use overlapping syntactical notation.
- In case the extension packages are not aware of each other, how to make translation rules in one package aware of rules in the other package, e. g., to be able to define *schema modules* containing *relationships* with extended semantics (see Sect. 1)?
- How would the translator recognize the package where the translation process has to be initiated? For this purpose, at least a basic global registry system for extensions is required in any case.

As extensions are merged to a single-level directory, we force the user to make decisions on the above issues prior to using the extensions, so that these problems can be avoided at translation time. The *registry* is the simplest part of the extension directory. It is used to uniquely identify the packages that have been imported in the directory. The following sections describe the remaining parts of the directory.

Extended Metamodel. There is a set of (object-relational) constructs present in SQL:1999 that are commonly used for organizing, storing and querying data: complex structured types, typed tables, user-defined routines, etc. Imported language extensions introduce additional constructs, e.g., a table inheriting from more than one table, a relationship with extended semantics or a schema module. The extension directory uses an *extended metamodel* to organize the constructs defined in imported extension packages and thereby available in XSQL (including those already provided by SQL:1999). Instead of building the SQL:1999 part of the metamodel from scratch, we decided to use the OMG's Common Warehouse Metamodel (CWM) [18]. CWM is a metamodel based on Meta Object Facility (MOF) [16] designed to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories. Choosing an already present and accepted OMG metamodel, the users of extension packages might already be familiar with, alleviates the tasks of importing new packages that require user assistance. The extended metamodel is stored in a dedicated repository that provides a GUI for interactive and an API

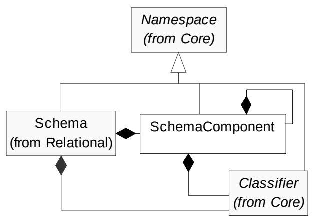

Fig. 4. Excerpt of an extended metamodel

for programmatic manipulation of the extended metamodel. Fig. 4 shows a part of the CWM extended to support schema components.

Extended XSQL EBNF. The syntax of XSQL statements is defined by the XSQL EBNF. The EBNF is extensible, meaning that at import time an extension package will attempt to apply rules that upgrade the current EBNF with new constructs. In a similar fashion as with the extensible metamodel, this process is user-mediated: The user can accept or reject the application of a specific production rule contained by the EBNF upgrade or modify the production rule to avoid possible inconsistencies with other packages and assure compliance with the metamodel. In certain cases, the system is capable of suggesting multiple alternative production rules for the EBNF upgrade. The suggested rules are constructed by exploiting the information in the metamodel and existing XSQL EBNF.

There is a mapping of XSQL EBNF symbols onto class implementations. A class implementation of an EBNF symbol will be instantiated to represent a node in the XSQL syntax tree. For this reason, the class implementation of non-terminal symbols defines associations to implementations representing other symbols, which facilitates the traversal of the syntax tree. Most importantly, a class implementation provides reduction methods used by the translator to transform XSQL statements.

Reduction Methods for Producing SQL:1999 Statements. Reduction methods implement operations used to transform nodes in the syntax tree of the XSQL statement to produce the reduced syntax tree(s) of the SQL:1999-compliant statement(s) – in most cases, a single statement in XSQL is transformed into more than one SQL:1999 statements. Class implementations for the EBNF symbols are organized into an inheritance hierarchy. For example, the class implementation for a table capable of multiple inheritance would extend the implementation for the common table construct. As the translator parses the XSQL statement to build an initial syntax tree, it uses information in the extended system catalog (see Sect. 3.3) to select the class implementation that needs to be instantiated to represent a given node in the syntax tree. Reduction methods defined by class implementations are polymorphic in the following sense:

- A reduction method in the subclass may extend or completely override the default implementation of the reduction method implemented in the superclass.

```
public class SQLComponentDeclaration implements NonTerminalSymbol {

  protected ComponentDeclarationName     name;
  protected ComponentDeclarationElements elems;

  public void reduce( SyntaxTreeSymbol caller, SyntaxTree tree ) {
    this.addReduction(
      new CallFunction( "declareComponent", this.name.getValue() ) );
    Enumeration e = this.getComponentDeclarationElements().elements();
    while( e.hasMoreElements() ) {
      this.addReduction(
        new CallFunction( "addToComponentDeclaration",
                          this.name.getValue(),
                          e.nextElement().getValue() ) );
    }
    // Store additional reduction information into the transformed tree.
  }
  ...
}
```

Fig. 5. An automatically generated reduction method

- The behavior of the reduction method (the generation of nodes for the SQL:1999-compliant syntax tree) depends on the pattern the node resides in, meaning that in order to complete the reduction process, the method may need to gather information from the neighboring nodes.

Fig. 5 shows a reduction method for transforming the declaration of a schema component into routine calls that will insert the declaration (including the component elements) to the extended system catalog. The method has been automatically generated from translation rules illustrated in Fig. 3. For a detailed overview of reduction methods and reduction pattern concepts, we refer to [3].

3.3 Extended System Catalog

The imported packages introduce constructs initially not present in the SQL:1999 or in the SQL dialect of the target database system. Because of this, the database system catalog (i.e., system tables) no longer suffices for storing database metadata (e.g., it does not allow us to store which tables and complex structured types are combined in a schema component). To support complete organization and storage of metadata needed for translating and executing XSQL statements, we introduce an *extended system catalog*. The tables in this catalog are instantiated from the constructs defined in the extended metamodel that is a part of the extension directory using the extended system catalog hints. For example, for the metamodel illustrated in Fig. 4, using the hints in Fig. 3 we obtain a table used to store schema components, a table to map the names, etc. To avoid replication of metadata, relational views are defined on DB-proprietary system tables so that these can be directly reused as part of the extended catalog. View definitions are supplied within the extension package in a specific form for each DBMS product supported by the vendor. However, the definitions can be customized by the user during the import of the package to allow the use of the extended system catalog with further DBMS products initially not supported by the vendor.

```
CALL declareComponent('BookManagement', ...);
CALL addToComponentDeclaration('bookT', ROW(ARRAY(ROW('title',...)...));
CALL addToComponentDeclaration('chapterT',...);
CALL addToComponentDeclaration('book', ...);
CALL addToComponentDeclaration('chapter', ...);
CALL addToComponentDeclaration('addChapter', ...);
...
```

Fig. 6. Excerpt of a translated statement in SQL:1999

When familiar with tables and views contained in the extended system catalog, the user may query the catalog to explore metadata for the database. Manual updates to the catalog by the user are not recommended, since the consistency of the catalog may be jeopardized. However, the translator is aware of any consistency constraints related to the catalog, so that after their execution the translated DML statements that manipulate the catalog will always leave it in a consistent state.

3.4 Translator

The translator accepts a statement in XSQL and constructs a corresponding syntax tree using the XSQL EBNF included in the extension directory. Afterwards, it invokes the reduction method of the root node in the syntax tree. The method delivers one or more reduced syntax trees that represent the translated statements in SQL:1999.

A method may access the extended metamodel and the extended system catalog via a set of dedicated APIs to gather information needed in the course of reduction. In addition, the execution of the reduction method of a node may require information related to the reduction of other nodes residing in remote parts of the XSQL syntax tree. We call constellations of nodes that require uni-directional or mutual information on each other's reduction outcomes *node patterns*. A single XSQL syntax tree may contain multiple (possibly overlapping) node patterns.

The translator exposes the functionality for initiating remote reductions and delivering the results in form of an API. In this way, all reductions in the XSQL syntax tree are initiated by the translator, which maintains a reduction log and makes sure the process does not contain any cycles and will eventually terminate.

Obtaining the second statement of Fig. 1 (line 12-29) as input, the translator would produce an SQL:1999-compliant statement as shown in Fig. 6. Encapsulated in the UDRs, these statements insert data into the extended system catalog.

4 Open Issues

This section attempts to address a set of important issues that assist us in assessing the pros and cons of the approach.

SQL serves its purpose well and the capabilities of language extensions can be simulated by providing an API in a general-purpose language (GPL), such as Java. So why introduce custom extensions?

SQL serves its purpose well as long as you downgrade your problem domain so that it fits the limits of the object-relational model. By doing this, you are losing design information and higher-level abstractions that would ideally be present in your schema. Unfortunately, both of these are key issues for effective schema reuse, matching, reengineering and reverse engineering. How can you be sure that a trigger in a schema was initially meant (a) as a pure trigger, or (b) is the result of downgrading a higher-level abstraction, such as a relationship with extended semantics, so that someone could have expressed it in the object-relational model? How can you be sure that a table in the schema was (a) meant to exist at the same level with other tables or (b) initially originated from a modular and replaceable schema part (i.e., schema module), but has been flattened out of the module to this level, since there was no effective way to represent the module in the object-relational model?

Leaving SQL unadorned and simulating its extensibility in a GPL does not solve the problem either, since the syntax and semantics of such a language may prove too limiting as well and less natural as to extend the syntax of SQL:1999 at any place convenient. In this aspect, by offering the extensions in form of an API, we don't really avoid the problem, but rather migrate it to another level: In the same manner as the syntax of SQL:1999 limits us in efficiently expressing higher-level abstractions, the syntax of the GPL might be a similar obstacle.

Object-relational databases are bloated with features we don't really need [2], [20]. Won't relationships with extended semantics, schema modules and other extensions make things look even worse?

For Date and Darwen [4], orthogonality of language concepts is a key issue: A language is orthogonal, if independent concepts are kept independent and not mixed together in confusing ways. Pitfalls in the orthogonality increase the complexity of the language and reduce its expressive power. In our opinion, careful selection of language extensions to SQL:1999 does not necessarily implicate the loss of orthogonality, but rather allows developers to retain complex high-level abstractions directly in their schema implementations and queries embedded in application code.

Customers will define own extensions and purchase off-the-shelf extension packages, which will lead to the incompatibility of extensions, cause notational havoc, prevent exchange of schema design and burden reengineering of database applications.

This is the same problem Czarnecki and Eisenecker [3] mention as they discuss the role of off-the-shelf active libraries in the context of Intentional Programming [22] (see Section 5). They claim that in the same manner as conventional libraries emerge in different domains today, standard domain-specific language extensions will emerge. Once a market for off-the-shelf extensions will be present, developers will have access to high-quality extensions provided by vendors, which will decrease the need to implement own extensions. For a detailed classification of problems related to interactions among language abstractions originating from different active libraries, we refer to [3]. In X-Translate, we cope with these problems by using a global metamodel for extension constructs, a global system catalog and a global XSQL EBNF with reduction methods. To resolve possible conflicts, the import of extension packages is user-mediated.

The reduction process impedes processing and thereby has an impact on overall performance of database applications. Therefore, language extensions introduce benefits not for customers, but mostly for developers, which can now easily design and maintain database applications.

The outcome of reducing XSQL statements are multiple statements that conform to SQL:1999. XSQL statements cannot be completely translated at compile time of the application, since in most cases the outcome will depend on the state of the system catalog at runtime. To reduce the impact of translation delay, we implemented an approach supporting *pre-compilation*. In X-Translate, the reduced statements in SQL:1999 contain empty placeholders that are filled by values fetched at runtime from the system catalog using a user-defined routine. Thereby, excessive invocations of reduction methods at run-time are avoided.

The execution time of statements once fully translated is another issue: This time can be improved by assuring that reduction methods produce efficient translations of proprietary SQL statements for a given language extension.

To a certain extent, we admit that performance drawbacks are an issue that has to be accepted if language extensions are used. However, it would be interesting to examine the trade-offs related to the total cost of ownership of applications that are easier to maintain, but on this account demonstrate minor performance disadvantages due to language extensions used.

5 Related Work

The idea of an extensible programming and metaprogramming environment is materialized in the Simonyi's work [22] on *Intentional Programming* (IP). As a programming environment, IP supports developers in the programming tasks by allowing them to load extension libraries in order to extend the programming language with general-purpose and domain-specific abstractions [3]. Among other components, the environment includes dedicated *editor and browsing tools*, allowing the developers to browse the program source in form of a syntax tree and a *reduction engine* that is used to generate an implementation based on a set of primitive abstractions. Our X-Translate system can be seen as an attempt to examine how the idea of extensible programming environment concepts applies to SQL:1999.

An early paper on query language translators by Howells et al. [6] does not discuss the notion of introducing and translating custom extensions, but rather focuses on language-to-language translation. The approach uses a common internal relational algebra tree with input language translation schemes specified as sets of extendible PROLOG clauses.

A substantial amount of work is done on transforming schemas and translating queries in one query language into SQL. For example, Keim et al. [11] describe the translation of Structured Object Query Language (SOQL) into SQL, arguing that SOQL queries are shorter, easier to write and understand and more intuitive than corresponding SQL queries. Shanmugasundaram et al. [21] present the XPERANTO system that involves the evaluation of XML queries over XML views of relational data. It supports processing arbitrary complex queries specified using the XQuery query language. Most computation is pushed to the relational engine to increase the efficiency of the system.

Query translation proves important for accessing heterogeneous information sources via a single application interface. Queries posed to this interface have to be translated to source-specific (sometimes called native) queries or commands [19]. Since SQL:1999 was chosen as a target platform, which already includes an approach for integrating heterogeneous data sources (see SQL/MED [9]), considering diverse mappings to multiple native query languages would be superfluous in our case. Instead of tackling the problems that arise from translating a single common query language to diverse native languages, our approach focuses on problems that arise on behalf of extensibility of the common language.

6 Conclusion and Future Work

In this paper, we presented X-Translate, a system supporting the definition of custom language extensions to SQL:1999. Language extensions allow the formation of complex data definition and manipulation statements as well as queries that involve custom language constructs. The main purpose of these constructs (which can be general purpose as well as domain specific) is to allow developers the expression of high-level abstractions that exist at the time the database application is developed, but usually get lost in the mapping to the limited set of original SQL:1999 constructs. We believe the loss of these abstractions is a major obstacle for efficient schema reuse and database application reengineering.

The paper has shown that the idea of extensible programming language environment (resembling the one described by Simonyi [22]) can prove useful for the implementation of schemas and the development of database applications. It is possible to support the introduction of language extensions in form of packages that are imported into the system and to semi-automatically extend the corresponding metamodel and XSQL EBNF (which are shared among different extensions). The process of importing extension packages has to be user-mediated to assist the system in resolving possible conflicts.

In the course of our future work we intend to:

- Consider further language extensions that might be useful for developers. As our first goal, we attempt to focus on version management of table data.
- Consider a number of interactive tools that would alleviate the import of extension packages into X-Translate, such as visualization and interactive editing of the extended metamodel and XSQL EBNF as well as dedicated browser tools for the extended system catalog. In addition, especially when importing new language packages, visualization of applied reductions of a statement plays an important role for discovering inconsistencies with existing language extensions. In our opinion, a dedicated debugging tool capable of tracking and visualizing successive reduction phases as well as accesses to the extended system catalog would prove useful. We will try to integrate the mentioned tools in a single visual environment to be used with the X-Translate system.

- Explore techniques for improving overall performance of the system.
- Empirically evaluate the scope in which SQL:1999 extensions alleviate the development, maintenance, and reengineering of database applications using a large set of selected sample applications.

References

[1] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Peach, B., Wust, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison Wesley, 2002.

[2] Carey, M., Hellerstein, J., Stonebraker, M.: A Sketch of Regres, Seminar presentation, UC Berkeley, 1999, http://db.cs.berkeley.edu/postmodern/stonebraker-final.ppt

[3] Czarnecki, K., Eisenecker, U.W.: Generative Programming, Methods, Tools and Applications, Addison-Wesley, 2000.

[4] Date, C.J., Darwen, H.: A Guide to SQL Standard, Addison-Wesley, 1996.

[5] D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison Wesley, 1998.

[6] Howells, D.I., Fiddian, N.J., Gray, W.A.: A Source-to-Source Meta-Translation System for Relational Query Languages, in: Proc. VLDB 1987, Brighton, Sept. 1987, pp. 227-234.

[7] ANSI/ISO/IEC 9075-1:1999, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework), ISO, 1999.

[8] ANSI/ISO/IEC 9075-2:1999, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM), ISO, 1999.

[9] ANSI/ISO/IEC 9075-9:2000, Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED), ISO, 2000.

[10] Kovse, J., Härder, T.: Generic XMI-Based UML Model Transformations, in: Proc. 8th Int. Conf. on Object-Oriented Information Systems (OOIS'02), Montpellier, Sept. 2002, pp. 192-198.

[11] Keim, D.A., Kriegel, H.-P., Miethsam, A.: Object-Oriented Querying of Existing Relational Databases, Technical Report 93-08, Dept. of Computer Science, Ludwig-Maximilian University, Munich, Germany.

[12] Kießling, W.: Foundations of Preferences in Database Systems, in: Proc. VLDB 2002, Hong Kong, Aug. 2002, pp. 311-322.

[13] Mahnke, W.: Towards a modular, object-relational schema design, in: Proc. 9th Doctoral Consortium at CAiSE'2002, Toronto, May, 2002, pp. 61-71.

[14] Meyer, B.: Harnessing multiple inheritance, in: Journal of Object-Oriented Programming 1:4, 1988, pp. 48-51.

[15] Mahnke, W., Steiert, H.-P.: Modularity in ORDBMSs - A new Challenge, in: Proc. 13. Workshop "Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, Juni 2001, pp. 83-87.

[16] OMG: Meta Object Facility Specification, Version 1.4, April 2002, http://www.omg.org/.

[17] OMG: Unified Modeling Language Specification, Version 1.4, Sept. 2001, http://www.omg.org/.

[18] OMG: Common Warehouse Metamodel Specifications, Version 1.0, Oct. 2001, http://www.omg.org/.

[19] Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J.: A Query Translation Scheme for Rapid Implementation of Wrappers, in: Proc. DOOD 1995, Singapore, Dec. 1995, pp. 161-186.

[20] Schek, H.-J.: Panel: Future Directions of Database Research - The VLDB Broadening Strategy, Part 1, in: Proc. VLDB 2000, Cairo, Sept. 2000, pp. 663-664.

[21] Shanmugasundaram, J, Kiernan, J., Shekita, E.J., Fan C., Funderburk J.: Querying XML Views of Relational Data, in: Proc. VLDB 2001, Rome, Sept. 2001, pp. 261-270.

[22] Simonyi, C.: The Death of Computer Languages, the Birth of Intentional Programming, Tech. Report MSR-TR-95-52, Microsoft Research, Sept. 1995.

[23] Viega, J., Tutt, B., Behrends, R.: Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages, Technical Report CS-98-03, University of Virginia, 1998.

[24] Zhang, N., Ritter, N., Härder, T.: Enriched Relationship Processing in Object-Relational Database Management Systems, in: Proc. 3rd Int. Symposium on Cooperative Database Systems for Advanced Applications (CODAS'01), Bejing, Apr. 2001, pp. 53-62.

[25] ZsqlML: XML Markup Language for SQL http://zsqlml.sourceforge.net