Generating SQL Queries from SBVR Rules

Sotiris Moschoyiannis, Alexandros Marinos, Paul Krause

Department of Computing, FEPS, University of Surrey,
GU2 7XH, Guildford, Surrey, United Kingdom
{s.moschoyiannis, a.marinos, p.krause}@surrey.ac.uk

Abstract. Declarative technologies have made great strides in expressivity between SQL and SBVR. SBVR models are more expressive that SQL schemas, but not as imminently executable yet. In this paper, we complete the architecture of a system that can execute SBVR models. We do this by describing how SBVR rules can be transformed into SQL DML so that they can be automatically checked against the database using a standard SQL query. In particular, we describe a formalization of the basic structure of an SQL query which includes aggregate functions, arithmetic operations, grouping, and grouping on condition. We do this while staying within a predicate calculus semantics which can be related to the standard SBVR-LF specification and equip it with a concrete semantics for expressing business rules formally. Our approach to transforming SBVR rules into standard SQL queries is thus generic, and the resulting queries can be readily executed on a relational schema generated from the SBVR model.

Keywords: SBVR, SQL, Declarative Programming, Business Rules, Predicate Calculus, Formal Semantics

1 Introduction

The Business Rules Approach [1] has made significant strides in bridging the spheres of everyday human interactions and information technology. An outgrowth of that movement was the OMG standard Semantics of Business Vocabulary and Rules (SBVR) [2], which brought together research from linguistics, formal logics, as well as practical expertise. SBVR Models are considered constructs that are supposed to help businesses communicate with each other and also business people to communicate with implementers of information technology. Direct transformation of SBVR models into executable code is generally not encouraged and has often resulted in rather harsh compromises of SBVR's meta-model and intended use when attempted [3]. The reason for this mismatch is the chasm between the declarative paradigm implemented by SBVR and the imperative procedural paradigm that is at the heart of most modern programming and business process languages. So thus far, human programmers are needed to interpret and convert the SBVR models into real-world applications. An alternative approach, called Generative Information Systems

[4], was presented by the authors of this paper that allowed for real-world systems to be produced by inferring the appropriate reaction directly from a model, without the need for an intermediate code generation step, and without the need for explicitly defined business processes. A significant aspect of that model was the method of generating schemas for a relational database from the SBVR Vocabulary, and converting rules into SQL queries to verify the consistency of the data set. This last step, had been only sketched out in the original paper, as the theoretical framework required for this undertaking is significant. This paper addresses precisely these issues and examines in detail the conversion of SBVR rules into SQL queries for the purpose of validating the consistency of a given data set with the SBVR model.

2 Generative Information Systems

This section summarizes the architecture of Generative Information Systems (GIS) in [7] to provide the appropriate context for the rest of the paper. A GIS is based around the concept that the logic of the system is accessible to the owner of the system, and that any change in the logic is immediately reflected in the operation of the system. The architecture as can be seen in Figure 1, specifies that both the RESTful API and the relational database schema (in SQL-DDL) are to be generated from the model.

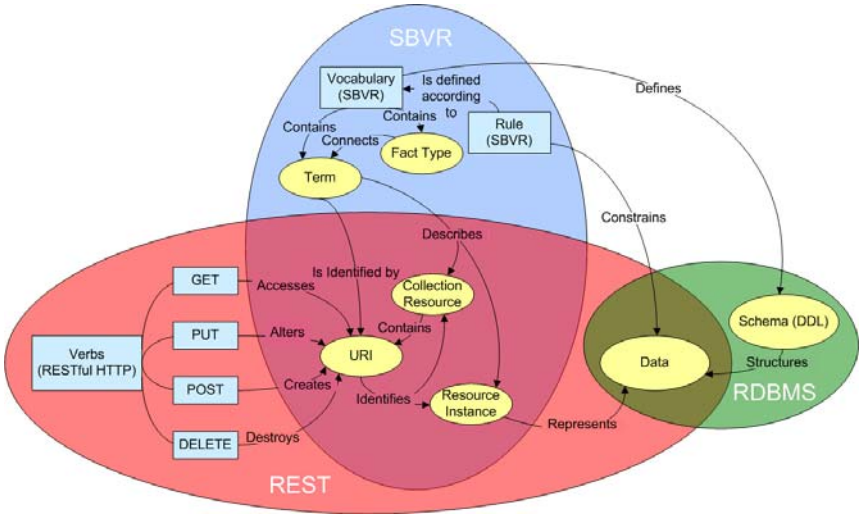

Fig. 1. Connections between REST, SBVR and Relational Databases

The end user can place requests on the system through the API. These requests get evaluated through the ruleset, and if they represent a legal transition and their result is a system in a consistent state, they are applied to the dataset. If not, the inconsistency is presented to the user, who can amend the request to take account of the new

information. Through this back and forth negotiation, the user either concludes that the request is fundamentally incompatible with the system, or reaches a formulation of the request that satisfies both the original goal and the system's consistency requirements. This is, in an abstract sense, what many processes achieve. By guiding the user through a sequence of steps, they determine what change needs to be made to the system state to satisfy the user's request while maintaining consistency. We call thus abstract process the 'meta-process'. Its advantage over the traditional process-driven message is that it can respond to unforeseen requests by the user, in contrast to the hardcoded process model which is constrained to the design-time foresight of the developers. More detail on this process can be seen in Figure 2.

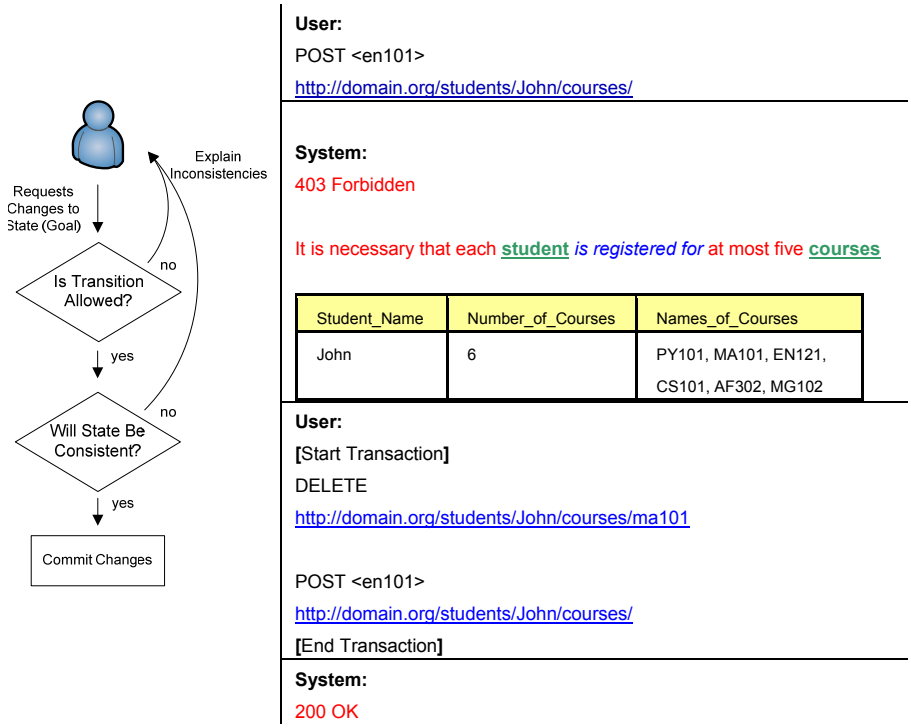

Fig. 2. The meta-process control structure

The way that consistency is currently checked is by performing a sequence of actions on the database as a transactional unit. First, a transaction is initiated. Secondly, the updates are applied to the dataset. If the database schema makes this impossible, the updates are rejected. If it is allowed, the relevant rules are checked against the dataset to make sure they are not violated. If they are violated, updates are rolled back and the details of the violation returned to the user. If the rules are not violated, the transaction proceeds. This mechanism is suitable for a proof of concept, but may have scalability limitations for concurrent systems. Optimizations can be explored that avoid the round-trip to the database. It is however interesting to note that this rough

process of adding tentative information to the knowledge base, then checking for consistency, and deleting in case of violation, counterintuitively seems to be the way that the human brain deals with new knowledge. [5] This does not mean that the method is ideal, but it is an interesting parallel that we noted after setting the foundations for Generative Information Systems architecture.

The step in the above process that was left least defined is the one where the updated dataset is checked against relevant rules for consistency. This is done by transforming each rule to an SQL query that requests violations to the rule to be returned (Figure 3). The precise mechanism by which this is carried out is the focus of this paper.

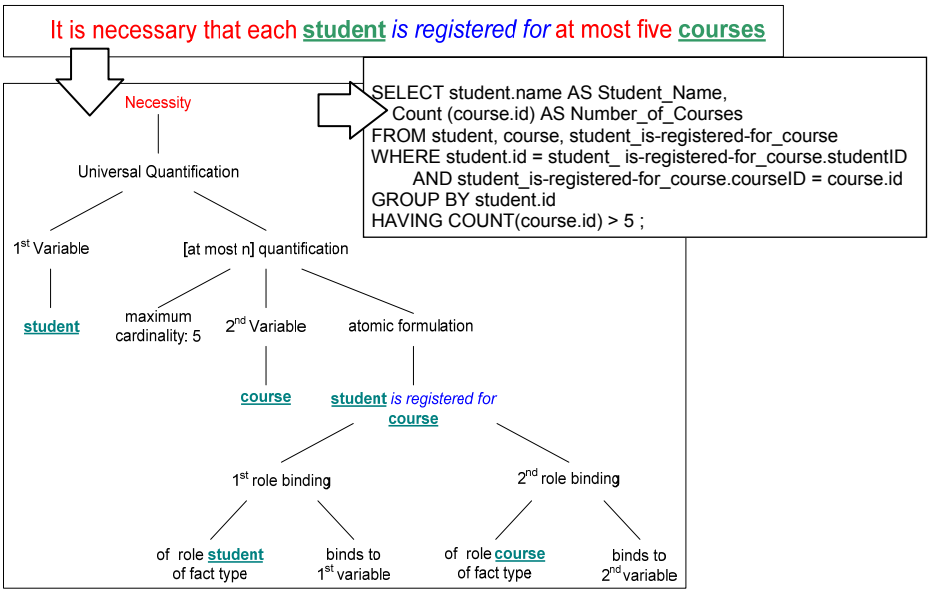

Fig. 3. From SBVR Structured English, to SBVR Logical Formulation, to an SQL query.

3 Vocabularies to SQL Schemas

For rules to be validated over a dataset however, first there must be a schema for that dataset. As our starting point is an SBVR model, it is the vocabulary that is the obvious candidate for becoming the scaffold for our schema. The detailed process has been described in our previous work so here we will instead go through an example scenario which we will use throughout this paper. The model for our example can be seen in Table 1. One aspect not covered in previous work is that of primitive data types. We can see that the term Name has a concept type of Varchar(255). This can be read as a reference to a vocabulary of primitive data types that a generative information system is built on. These terms are essentially terminal symbols that get mapped directly onto programming language data types. We use the data types that

are fundamental to SQL as this is our target data store. Another novel convention is that since the Name is had by Student , this relation is constrained to a one-to-one cardinality, and Name has no other attributes than its value, we render it an attribute of the table with which it is associated rather than representing it in a separate table, similarly for Code and Title.

Table 1. Example SBVR Model

Terms	Fact Types	Rules
Student	Student *is registered for* course	It is necessary that each student *is registered for* at most five courses.
Module	Student *is enrolled in* module	It is necessary that each student that *is under probation is registered for* at most three courses.
Course	Module *is available for* course	It is obligatory that each student *has* exactly one name.
Name	Student *is under probation*	It is obligatory that each course *has* exactly one title
Concept-type: Varchar(255)	Student *has* name	It is obligatory that each module *has* exactly one code.
Code	Course *has* title	
Concept-type: Varchar(255)	Module *has* code	
Title		
Concept-type: Varchar(255)		

The result of converting the vocabulary (and some of the more basic rules) into a schema can be seen in Figure 4.

```
CREATE TABLE student (id INT NOT NULL AUTO_INCREMENT,  name VARCHAR(255),
          is-under-probation BOOL, level INT, primary Key (id));

CREATE TABLE course (id INT NOT NULL AUTO_INCREMENT,
          code VARCHAR(255), primary Key (id));

CREATE TABLE module (id INT NOT NULL AUTO_INCREMENT,
          title VARCHAR(255), primary Key (id));

CREATE TABLE student_is-enrolled-in_module (studentID INT, moduleID INT,
          primary Key (studentID, moduleID),
          foreign Key (studentID) references student(id),
          foreign Key (moduleID) references module(id));

CREATE TABLE student_is-registered-for_course (studentID INT, courseID INT,
          primary Key (studentID, courseID),
          foreign Key (studentID) references student(id),
          foreign Key (courseID) references course(id));

CREATE TABLE course_is-available-for_module (courseID INT, moduleID INT,
          primary Key (courseID, module_id),
          foreign Key (courseID) references course(id),
          foreign Key (moduleID) references module(id));
```

Fig. 4. Resulting SQL DDL Schema

4 A predicate calculus for advanced SQL DML constructs

We have seen how an SBVR vocabulary can be used to generate a relational schema. In the remainder of the paper, we are concerned with translating SBVR rules into SQL queries. This operational rendering of business rules is more challenging. Thus, we want to prove the correctness of the transformation from SBVR-LF to SQL DML. SBVR-LF has a formal foundation based on first-order or predicate logic, and its variations [2]. SQL has established theoretical foundations [8] and a sound semantics for its basic constructs (SELECT-FROM-WHERE) is based on a tuple relational calculus [6]. This standard semantics however does not cover more advanced SQL DML constructs, such as arithmetic operations, aggregate functions [10], grouping, and grouping on condition. In this section we describe a tuple relational calculus extension that equips such constructs with a clearly defined semantics – this is necessary for operationalising SBVR rules which are more expressive than basic SQL queries (e.g. see running example). The result is a predicate calculus with identity, which establishes a generic mapping between SBVR-LF and SQL DML, as discussed in Section 5. We use the student enrollment example to illustrate our approach.

4.1 Basic structure of an SQL query

Our predicate calculus formalisation of SQL DML makes use of tuple variables. A tuple variable is a variable that ranges over a named relation (table). The general form of a query in tuple relational calculus is

$$\{\mathbf{x} \mid F(\mathbf{x})\}$$

where $\mathbf{x}$ is the set of tuples for which the expression $F(\mathbf{x})$ is true. The relation is defined somewhere inside $F(\mathbf{x})$. As we will see, in our approach we make this explicit by separating the filter from the domain.

If only some attributes of $\mathbf{x}$ are of interest, the above expression takes the form

$$\{\mathbf{x} : (x_1, x_2, .., x_m) \mid F(\mathbf{x})\}$$

where $x_1, .., x_m$ are attributes of the relation which is the result of the query (i.e. attributes of a tuple $\mathbf{x}$). This set is created by selecting all tuples $\mathbf{x}$ for which $F(\mathbf{x})$ is true, and then projecting those tuples on attributes $x_1, .., x_m$. The result of a query on a set of tuples (relation) is either a set of tuples matching a certain condition or a value (when using aggregate functions, cf. Section 4.3). For example, the query $\{\mathbf{x} : (name, id) \mid student(\mathbf{x})\}$ returns a set of tuples which contain attributes *name* and *id* from the *student* relation.

A predicate $P(x)$ is a function that maps each element x of a set S to the value 'true' or 'false', i.e., $P : S \rightarrow \{true, false\}$.

Let $x \in N$ - so x is an element of the set of natural numbers. Then the predicate $P_1(x) \equiv x \geq 0$ is true for all x while the predicate $P_2(x) \equiv x < 0$ is false for all x.

Predicates can consist of one expression (as in $x \geq 0$ above) or as a combination of expressions. These combinations arise by combining expressions using the usual first-order or predicate logic operators (e.g. see [9]) given in Table 2.

Table 2. Logical connectives

$\wedge$ (conjunction)	$\vee$ (disjunction)	$\neg$ (negation)	$\Rightarrow$ (implication)

Let $x \in N$, as before. The predicate $P_3(x) \equiv x < 7 \wedge x > 10$ combines the expressions $x < 7$ and $x > 10$ (and is false for all $x \in N$). The predicate $P_4(x) \equiv x \geq 3 \wedge x < 6$ is true for x = 3,4,5 and false for all other $x \in N$. Predicates can be used to define sets. For example, $S_1 = \{x \mid x \in N \wedge P_1(x)\}$ denotes the set of all x such that x is natural number ($x \in N$) and satisfies the predicate P_1 (where $P_1(x) \equiv x \geq 0$, as before).

We have seen that P_1 is true for all x which means that S_1 is the set of natural numbers, and we can write $S_1 = \{x \mid x \in N \wedge P_1(x)\} = N$. Similarly, we have that $S_2 = \{x \mid x \in N \wedge P_3(x)\} = \varnothing$ where predicate P_1 is as defined before.

The fact that predicates can be used to define sets is well-known in mathematics and is central to our approach – we will be using the set membership to identify relations and the predicate as the selection condition on the tuples of these relations.

If *p* and *q* are expressions that valuate to true or false, sometimes called WFFs for Well-Formed Formulae in the literature, e.g. see [6], then the following equations hold. These are standard in first-order logic, e.g. see [9], so we list them here in Table 3 without further explanation. A thorough treatment can be found in [9].

Table 3. Equations on expressions (WWF)

$\neg(\neg p) \equiv p$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
$p \Rightarrow q \equiv \neg p \vee q$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

We now turn our attention to the basic structure of a query expressed in SQL DML in our formalization which is an extension to the tuple relational calculus while staying within the predicate calculus semantics – in particular, we will be concerned with setting up formal semantics for transforming SBVR rules to SQL queries based on a predicate calculus with identity.

Since our interest is in transforming SBVR rules to SQL queries on the relational schema generated by the SBVR model, we will be concerned with predicates that define sets of tuples. The general form of a query in our predicate calculus is

$$\{\mathbf{x} | D(\mathbf{x}) \wedge P(\mathbf{x})\}$$

where $\mathbf{x} : (x_1, .., x_n)$ is the set of tuples from a domain $D(\mathbf{x})$, and $D(\mathbf{x})$ specifies the set of all possible tuples that $\mathbf{x}$ ranges over, i.e. a relation with $x_1, .., x_n$ attributes, and $P(\mathbf{x})$ is a predicate on the set of all tuples in $D(\mathbf{x})$. For example, $\{\mathbf{x} | student(\mathbf{x}) \wedge student.id = '6081958'\}$ returns the set of all tuples $\mathbf{x}$ from the relation *student* whose attribute *id* has the value 6081958. (Note that *student.id* is a primary key in our schema, given in Section 3, so this expression would return a single tuple.)

The expression $\{\mathbf{x} | D(\mathbf{x}) \wedge P(\mathbf{x})\}$ in the extended predicate calculus considered here is mapped to SQL DML as:

SELECT DISTINCT $\mathbf{x}$
FROM $D(\mathbf{x})$
WHERE $P(\mathbf{x})$;

The SQL keyword DISTINCT is used to remove duplicates.

If we are only interested in certain attributes $x_1,..,x_n$ in the result $\mathbf{x}$ and not all attributes $x_1,..,x_m$ of the relation specified in $D(\mathbf{x})$, then we write for the projection

$$\{\mathbf{x}:(x_1,x_2..,x_n)|D(\mathbf{x})\wedge P(\mathbf{x})\}$$

which is mapped to SQL DML as:

SELECT DISTINCT $x_1,..,x_n$
FROM $D(\mathbf{x})$
WHERE $P(\mathbf{x})$;

To express the JOIN statements in SQL DML which applies to two or more relations, we need to take a closer look at $D(\mathbf{x})$. In standard tuple relational calculus semantics, it is well known that joining two relations means taking the Cartesian product ($\times$) of the two relations. In our formalization, the join of two relations (tables) is captured in $D(\mathbf{x})$ which is what is used to specify the set of all tuples from which the returned set of tuples $\mathbf{x}$ come from. The join condition, if any, is then added in the predicate $P(\mathbf{x})$ - and that is in addition to the selection condition, if any.

Therefore, if we want to join tuples from relations $\mathbf{y}_1,\mathbf{y}_2,..,\mathbf{y}_k$ we write

$$\{\mathbf{x}:(x_1,...,x_n)\mid D(\mathbf{y}_1)\times D(\mathbf{y}_2)\times\cdots\times D(\mathbf{y}_k)\wedge P(\mathbf{x})\}$$

where $D(\mathbf{y}_1)\times D(\mathbf{y}_2)\times\cdots\times D(\mathbf{y}_k)=D(\mathbf{x})$. This is mapped onto SQL DML as:

SELECT $x_1,..,x_n$
FROM $D(\mathbf{y}_1)\times D(\mathbf{y}_2)\times\cdots\times D(\mathbf{y}_k)$
WHERE $P(\mathbf{x})$;

Note that $\mathbf{y}_1,\mathbf{y}_2,..,\mathbf{y}_k$ denote relations (sets of tuples) while $x_1,..,x_n$ is the list of attributes returned after the join of the relations, and this is denoted by $\mathbf{x}:(x_1,...,x_n)$. It is also worth pointing out that selection conditions on attributes of $\mathbf{y}_1,\mathbf{y}_2,..,\mathbf{y}_k$ are included in $P(\mathbf{x})$ since they are applied after the Cartesian product on these relations has been applied. For example,

$$\{\mathbf{x}:(student.id,student.name,COUNT(course.id)\mid D(student)\times D(s\_irf\_c)\times \\ \times D(course)\wedge student.id=s\_irf\_c.studentID\wedge s\_irf\_c.courseID=course.id\}$$

is transformed into:

```
SELECT    student.id, student.name, COUNT(student.id)
FROM      student, s_irf_c, course
WHERE     student.id = s_irf_c.studentID AND s_irf_c.courseID = course.id ;
```

We now turn our attention to arithmetic operations and aggregate functions.

4.2 Arithmetic operations and aggregate functions

In SQL, arithmetic operations may appear in the SELECT clause, as in:

```
SELECT      Salary*1.1, EmpID, EmpName
FROM        Employee
WHERE       DeptName = 'Research';
```

which reflects the values of a 10% increase in salaries in the Research department. So we need to apply this arithmetic operation as a function on the returned set of tuples $\mathbf{x}$. For this reason, we write

$$\{E(\mathbf{x}) \mid D(\mathbf{x}) \wedge P(\mathbf{x})\}$$

where $E(\mathbf{x})$ is a function on $\mathbf{x}$ that includes addition (+), subtraction (-), multiplication (*), division (/), or a combination of these on one or more attributes of the tuples in $\mathbf{x}$, i.e. tuples from $D(\mathbf{x})$ which satisfy $P(\mathbf{x})$.

Often, arithmetic operations only apply to certain attributes in the set of returned tuples $\mathbf{x}$. So E should be applied to the attributes of $\mathbf{x}$ rather than across $\mathbf{x}$. Thus,

$$\{\mathbf{x}.(E(x_1), E(x_2),..., E(x_n)) \mid D(\mathbf{x}) \wedge P(\mathbf{x})\}$$

where $E(x_i)$, $i = 1..n$, is applied to some attributes, in which case it is one or more of '+', '-', '*', '/' and not applied to others, in which case we have $E(x_i) = x_i$ (identity).

In similar fashion, we can address the aggregate functions in SQL DML, i.e. SUM, AVG, MIN, MAX, COUNT. To take into account the fact that an arithmetic operation may have been already applied to a certain attribute, we define F as a composite function on E so that $(F \circ E)(x_i) = F(E(x_i))$. In other words, F is applied to the output of E, and we write

$$\{\mathbf{x}.(F(E(x_1)), F(E(x_2)),..., F(E(x_n))) \mid D(\mathbf{x}) \wedge P(\mathbf{x})\}$$

Note that if F is the aggregate function COUNT, for some attribute x_i, then $E(x_i)$ must be the identity, i.e. $E(x_i) = x_i$, so that only attribute names are allowed in this case and no arithmetic operations.

This predicate calculus construction is mapped onto SQL DML as

SELECT	$F(E(x_1)), F(E(x_2)),..., F(E(x_n))$
FROM	$D(\mathbf{x})$
WHERE	$P(\mathbf{x})$;

For example,

$$\{\mathbf{x}.(student.id, student.name, COUNT(student.id)) \mid student(\mathbf{x}) \wedge student.level = 3\}$$

is mapped onto the query:

SELECT	student.id, COUNT(student.id)
FROM	student
WHERE	student.level = '3' ;

and returns the number of final year students in the dataset.

We now turn our attention to grouping and filtering on groups.

4.3 Grouping and Having

The grouping operation on a database comes down to stating the desired grouping attribute(s) and the grouping condition, if any. The grouping condition selects those groups that satisfy the condition and discards those who do not. In our formalisation, the grouping attributes are specified before the projected attributes (and therefore will be mapped onto the SELECT clause in SQL DML) while the grouping condition will

be part of the predicate itself. We note that it cannot be included in $P(\mathbf{x})$, like we did for JOIN, because the grouping condition applies to the results of the grouping operation, i.e., once the groups have been formed by the grouping operations.

Therefore, if we want to group a relation by a set of attributes $x_l,...x_m$ (a subset of all the attributes $x_1,...x_n$ of the relation), we write

$$\{\mathbf{x}{:}(x_l,..,x_m), \mathbf{x} : (x_1,..,x_n) | D(\mathbf{x}) \wedge P(\mathbf{x})\}$$

$(x_l,..,x_m)$ and $(x_1,..,x_n)$ need not be disjoint but both need to be subsets of the set of attributes of $D(\mathbf{x}) = (x_1,..,x_n)$. Finally, we note that $D(\mathbf{x})$ may be the result of the Cartesian product of a number of relations, as before.

The above expression in our predicate calculus is mapped onto SQL DML as:

SELECT	$x_1,..,x_n$
FROM	$D(\mathbf{x})$
WHERE	$P(\mathbf{x})$
GROUP BY	$x_l,..,x_m$

For example,

$$\{\mathbf{x} : student.id, \mathbf{x} : (student.id, student.name, COUNT(course.id) |$$

$$| D(student) \times D(s\_irf\_c) \times D(course) \wedge$$

$$\wedge student.id = s\_irf\_c.studentID \wedge s\_irf\_c.courseID = course.id\}$$

returns the number of courses a student has taken, and does this for every student.

This translates to the following SQL query:

```
SELECT      student.id, student.name, COUNT(course.id)
FROM        student, s_irf_c, course
WHERE       student.id = s_irf_c.studentID  AND
                                  AND s_irf_c.courseID = course.id
GROUP BY    student.id ;
```

Next we may add the grouping condition as an additional predicate $H(\mathbf{x})$ which applies to the result (set of tuples) of the grouping operation, i.e. to the set of attributes in $(x_l,..,x_m) \subseteq (x_1,..,x_n)$. Therefore, we write

$$\{\mathbf{x}{:}(x_1,..,x_n) | \{\mathbf{x}{:}(x_l,..,x_m), \mathbf{x} : (x_1,..,x_n) | D(\mathbf{x}) \wedge P(\mathbf{x})\} \wedge H(\mathbf{x})\}$$

which is mapped onto SQL DML as:

SELECT	$x_1,..,x_n$
FROM	$D(\mathbf{x})$
WHERE	$P(\mathbf{x})$
GROUP BY	$x_l,..,x_m$
HAVING	$H(\mathbf{x})$

Note that this is different to a nested predicate calculus expression because a nested query would simply apply a selection condition to the result of the inner query but could project onto different attributes. In contrast, a grouping condition only filters the groups returned by the grouping operation, and thus cannot apply a further projection. For a nested query we would write

$$\{\mathbf{x}{:}(x_1',..,x_n') | \{\mathbf{x}{:}(x_l,..,x_m), \mathbf{x} : (x_1,..,x_n) | D(\mathbf{x}) \wedge P(\mathbf{x})\} \wedge P'(\mathbf{x})\}$$

which would in turn map onto the following SQL DML:

SELECT $x_1',..,x_n'$
FROM (SELECT $x_1,..,x_n$
FROM $D(\mathbf{x})$
WHERE $P(\mathbf{x})$)
GROUP BY $x_l,..,x_m$)
WHERE $P'(\mathbf{x})$;

It can be seen that $P'(\mathbf{x})$ applies to the result of the inner query, but the result of the nested query as a whole can include a projection on any attributes from $D(\mathbf{x})$.

Going back to our example, if we want to check whether the business rule

It is necessary that each **student** *is registered for* at most five **courses**

expressed in the SBVR model given earlier in Figure 3 is satisfied, we need to restrict to groups (one for each student) who are associated with (registered for) more than five courses. We check for these cases since these are cases where the rule might be violated, and if this happens, the corresponding database operations will need to be executed as a transaction. Taking into account the associated database schema, this rule is expressed in terms of our extended predicate calculus as follows:

$$\{\mathbf{x} : (student.id, student.name, COUNT(student.id)) \mid$$

$$\mid \{\mathbf{x} : student.id, \mathbf{x} : (student.id, student.name, COUNT(course.id)) \mid$$
$$\mid D(student) \times D(s\_irf\_c) \times D(course) \wedge student.id = s\_irf\_c.studentID \wedge$$

$$\wedge s\_irf\_c.courseID = course.id\} \wedge (COUNT(course.id) > 5)\}$$

which is in turn mapped onto the following SQL DML statements:

```
SELECT      student.id, student.name, COUNT(course.id)
FROM        student, s_irf_c, course
WHERE       student.id = s_irf_c.studentID
                        AND s_irf_c.courseID = course.id
GROUP BY    student.id
HAVING      COUNT(course.id) > 5 ;
```

It is in this way that we can take rules from an SBVR model and transform them into SQL DML so that we can then check whether they are satisfied on a relational database schema by executing a standard SQL query. In the next section we attempt to generalize this by taking a closer look at both ends, our predicate calculus -based formalisation and the SBVR-LF, and do so at the semantics level.

5 From SBVR-LF to SQL DML

In this section we turn our attention to the SBVR Logic Formulation (SBVR-LF) as defined in the SBVR specification document [2], and describe a mapping onto the predicate calculus foundation for SQL DML which was given in the previous section. The objective is to obtain a generic mapping between rules expressed in SBVR and queries expressed in standard SQL DML, since this would make business rules amenable to immediate validation against a dataset.

Before we embark on the mapping, we define the quantifiers within the predicate calculus semantics in our approach. There are two quantifiers in predicate logic that can be used in an expression (WFF) to find out how many elements of the corresponding set satisfy the expression.

Let $P(\mathbf{x})$ be a predicate (as before). Let $D(\mathbf{x})$ denote the domain of $\mathbf{x}$, i.e. the set of all possible values for the tuple $\mathbf{x}$. To find out if *for at least one* tuple $\mathbf{x}$ from the domain $D(\mathbf{x})$ the predicate $P(\mathbf{x})$ is true, we write

$$(\exists \mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x}))$$

which is read as "there is an $\mathbf{x}$ for which $D(\mathbf{x})$ holds, and $P(\mathbf{x})$ is true". The result of this expression is either true or false.

To find out if for all tuples in the domain $D(\mathbf{x})$ the predicate $P(\mathbf{x})$ is true, we write

$$(\forall \mathbf{x})(D(\mathbf{x}) \Rightarrow P(\mathbf{x}))$$

which is read as "for all $\mathbf{x}$ for which $D(\mathbf{x})$ holds, $P(\mathbf{x})$ is true". The result of this expression is true or false.

With reference to the example predicates discussed in the start of Section 4.1, the expression $(\exists x)(x \in \boldsymbol{N} \wedge P_3(x))$ is false. The expression $(\forall x)(x \in \boldsymbol{N} \Rightarrow P_1(x))$ is true.

Note the difference between $(\exists \mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x}))$ and $(\forall \mathbf{x})(D(\mathbf{x}) \Rightarrow P(\mathbf{x}))$ which can yield different results (true or false) for the same expression. To avoid such ambiguities the domain $D(\mathbf{x})$ of an expression with a universal quantifier is always placed to the left of the implication logical operator ($\Rightarrow$).

Again, drawing upon first-order logic we have that if $P(\mathbf{x})$ is a predicate and $D_1(\mathbf{x})$, $D_2(\mathbf{x})$ are domains (expressions that define relations from the database schema generated by the SBVR vocabulary, as discussed in Section 3, and hence restrict the set of all possible values for a tuple $\mathbf{x}$), then the following equations hold.

Table 4.

$(\forall \mathbf{x})(P(\mathbf{x})) \equiv \neg(\exists \mathbf{x})(\neg P(\mathbf{x}))$	$(\exists \mathbf{x})(P(\mathbf{x})) \equiv \neg(\forall \mathbf{x})(\neg P(\mathbf{x}))$
$(\forall \mathbf{x})(D_1(\mathbf{x}) \times D_2(\mathbf{x}) \Rightarrow P(\mathbf{x})) \equiv (\forall \mathbf{x})(D_1(\mathbf{x}) \Rightarrow (D_2(\mathbf{x}) \Rightarrow P(\mathbf{x})))$	
$(\forall \mathbf{x})(D_1(\mathbf{x}) \times D_2(\mathbf{x}) \Rightarrow P(\mathbf{x})) \equiv \neg(\exists \mathbf{x})((D_1(\mathbf{x}) \times D_2(\mathbf{x})) \wedge \neg P(\mathbf{x}))$	

We have given these standard equations in terms of predicates that define sets of tuples. In their general form, they apply to an element x rather than a tuple $\mathbf{x}$ and we would also have $\wedge$ instead of $\times$ in the last two.

The specification document of SBVR includes the definition of the Formal Logic and Mathematics Vocabulary [2, pp. 109-118] which provides the logical foundations for SBVR in terms of first-order logic. However, the SBVR specification predefines some numeric quantifiers [2, pp.97-98] in addition to the standard universal and existential quantifiers found in first-order predicate logic. These allow the user to say things like 'exactly one car' or 'exactly two cars' or 'at most 8 and at least 3 cars' or 'at most two cars' and so on. Due to space limitations we do not reproduce the SBVR predefined quantifiers here, and refer the interested reader to the SBVR specification.

The predefined quantifiers can be defined in terms of the quantifiers in our formalization, which were defined earlier in standard predicate logic (Table 4).

Drawing upon the definition schemas in [11], also outlined in [2], we may obtain a rewriting of the SBVR predefined quantifiers in our approach.

The *exactly one quantifier* in SBVR-LF, denoted by $\exists^{1}\mathbf{x}$, can be rewritten as:

$$(\exists^{1}\mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x})) \equiv (\exists\mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x})) \wedge (\forall\mathbf{y})(D(\mathbf{y}) \Rightarrow P(\mathbf{y}) \wedge \mathbf{y} = \mathbf{x})$$

The *at most n quantifier* given in SBVR-LF, denoted by $\exists^{0..n}\mathbf{x}$, can be rewritten in terms of our predicate calculus as:

$$(\exists^{0..n}\mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x})) \equiv \neg(\exists\mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x})) \vee (\exists\mathbf{x}_1)(D(\mathbf{x}_1) \wedge P(\mathbf{x}_1)) \vee$$
$$\vee ((\exists\mathbf{x}_1)(D(\mathbf{x}_1) \wedge P(\mathbf{x}_1) \wedge (\exists\mathbf{x}_2)(D(\mathbf{x}_2) \wedge P(\mathbf{x}_1)) \wedge \neg(\mathbf{x}_1 = \mathbf{x}_2) \vee$$
$$\vdots$$
$$\vee ((\exists\mathbf{x}_1)(D(\mathbf{x}_1) \wedge P(\mathbf{x}_1) \wedge \cdots \wedge (\exists\mathbf{x}_n)(D(\mathbf{x}_n) \wedge P(\mathbf{x}_n)) \wedge \neg(\mathbf{x}_1 = \cdots = \mathbf{x}_n) \wedge$$
$$\wedge (\forall\mathbf{y})(D(\mathbf{y}) \Rightarrow P(\mathbf{y}) \wedge ((\mathbf{y} = \mathbf{x}_1) \vee \cdots \vee (\mathbf{y} = \mathbf{x}_n)))$$

The first disjunction covers the case that there might not exist such a tuple $\mathbf{x}$ (case of 0), the second covers the case there is one such $\mathbf{x}$, the third is for two such $\mathbf{x}$, and so on. The last disjunction says that *n* such $\mathbf{x}$ may exist, but then there cannot be any more (*n+1*) tuples that satisfy the predicate.

Similarly, the *at least n quantifier*, denoted by $\exists^{n..}\mathbf{x}$, can be rewritten as:

$$(\exists^{n..}\mathbf{x})(D(\mathbf{x}) \wedge P(\mathbf{x})) \equiv (\exists\mathbf{x}_1)(D(\mathbf{x}_1) \wedge P(\mathbf{x}_1)) \wedge (\exists\mathbf{x}_2)(D(\mathbf{x}_2) \wedge P(\mathbf{x}_2) \wedge \neg(\mathbf{x}_1 = \mathbf{x}_2)) \wedge$$
$$\wedge \cdots \wedge (\exists\mathbf{x}_n)(D(\mathbf{x}_n) \wedge P(\mathbf{x}_n) \wedge \neg(\mathbf{x}_n = \mathbf{x}_1) \wedge \cdots \wedge \neg(\mathbf{x}_n = \mathbf{x}_{n-1})) \wedge$$
$$\wedge((\exists\mathbf{x}_{n+k})(D(\mathbf{x}_{n+k}) \wedge P(\mathbf{x}_{n+k}) \wedge \neg(\mathbf{x}_{n+k} = \mathbf{x}_1) \wedge \cdots \wedge \neg(\mathbf{x}_{n+k} = \mathbf{x}_n)) \vee$$
$$\vee(\forall\mathbf{x}_{n+1})(D(\mathbf{x}_{n+1}) \Rightarrow P(\mathbf{x}_{n+1}) \wedge ((\mathbf{x}_{n+1} = \mathbf{x}_1) \vee \cdots \vee (\mathbf{x}_{n+1} = \mathbf{x}_n))))$$

The first *n-1* conjunctions refer to each of the *n* tuples $\mathbf{x}$ that must exist, must satisfy the predicate. The last conjunction captures the fact that there may be *k* additional such $\mathbf{x}$ that satisfy the predicate or no other $\mathbf{x}$ (apart from the *n* we already have) may exist that satisfy the predicate.

The *at least n and at most m quantifier* given in SBVR-LF, and denoted by $\exists^{n..m}\mathbf{x}$, can be obtained by combining the rewriting of the *at least n* and that of the *at most n quantifiers* given earlier.

The intention behind SBVR-LF is to (be able to) capture business facts and business rules formally. Formal statements of business rules may then be transformed into logical formulations that can be read in software tools, or readily adopted in approaches like the one we describe in this paper. An example given in the specification [2, pp.90-91] is the formalisation of a static constraint that says 'each person was born on some date' as the logical formulation:

$$\forall x : person, \exists y : Date, x \text{ was born on } y$$

Going back to our example, the rule in our SBVR model can be written as:

$$\forall\mathbf{x} : \underline{\textbf{student}},\ \exists^{0..5}\mathbf{y} : \underline{\textbf{course}},\ \mathbf{x}\ \textit{is registered for}\ \mathbf{y}$$

With reference to the tree representation of this rule given in Figure 3 earlier, it can be seen that the root is a universal quantification ($\forall$), the 1st variable is **student**, the 2nd

variable is **course** and the max cardinality is 5 ($\exists^{0..5}$) while the atomic formulation that completes the [at most n] quantification node is **student** *is registered for* **course** and this binds the 1st variable to x and the 2nd to y .

In fact we are interested in disproving the rule, i.e. identifying students registered for 6 or more courses. This can be encoded by taking the negation of the logical formulation in which case the existential quantifier $\neg\exists^{0..5}$ gives $\exists^{6..}$. Thus, we have

$$\exists \mathbf{x} : \textbf{student},\ \exists^{6..} \mathbf{y} : \textbf{course},\ \mathbf{x}\ \textit{is registered for}\ \mathbf{y}$$

Now **student** and **course** are relations in our database schema (Figure 4) and so is *is registered for*, thus all three appear in the domain $D(\mathbf{x})$ (in a Cartesian product) and consequently in the FROM clause of the resulting SQL query. The primary key of **student** will have to match the foreign key of *is registered for*, similarly for **course**. These join conditions become the predicate $P(\mathbf{x})$ and hence appear in the WHERE clause. The cardinality on the existential quantifier (6 or more) is the condition applied to the resulting tuples (per student), hence becomes the predicate $H(\mathbf{x})$ and appears in the HAVING clause.

It can be seen that the predicate calculus with identity we presented provides a bridge between SBVR-LF and SQL DML. This means that SBVR rules can be re-written systematically as SQL queries, thus enabling their execution to maintain consistency of a database. The modality of the rule, which has not been addressed explicitly here, is taken into account only in enforcing consistency once a violation is observed. A violation of an alethic rule leads to a direct rejection of the update on the dataset while a violation of a deontic rule can be overridden if authorised by a user with sufficient privileges.

6 Conclusions and Future Work

In this paper we have briefly described the concept of generative information systems, and how rule-based modeling is at their core. We have discussed how an SBVR model (terms, fact types) is transformed into a relational schema that can act as a data store for our information system. By showing how the user interacts with the system, we have demonstrated the need for a formal and rigorous approach to transforming SBVR rules to SQL queries. This transformation allows a rule to be validated against the dataset in much the same way as issuing a query on a database.

The correctness of the transformation has been shown using a predicate calculus with identity, which extends standard relational theory to include provision for aggregate functions and arithmetic operations, and also address SQL DML constructs such as grouping (GROUP BY clause) and grouping on condition (HAVING clause).

The work in [12] is also concerned with generating SQL DML from business rules. However, the rules are expressed in the ORM-based language ConQuer and the transformation is not attempted at the semantic level (at least not through relational theory). The problem of operationalising SBVR business rules is challenging. There are transformations to UML class diagrams [13] and R2ML [14] within an MDA context, as well as the reverse transformation from OCL to SBVR [15]. Instead, we

have described the operational rendering of SBVR rules into standard SQL queries, which can then be readily executed to maintain consistency of a database.

To further the research discussed, the transformation needs to be implemented in a tool such that it can be applied to real-world problems. Another possible extension is to add model-checking capabilities to the model execution functionality, described here, such that models with inconsistent, redundant, or needlessly complex rules can be identified and refined accordingly.

Acknowledgements

This work has been supported by the European Commission through IST Project OPAALS: Open Philosophies for Associative Autopoietic Digital Ecosystems (No. IST-2005-034824).

References

1. R. G. Ross, "The Business Rules Manifesto," Business Rules Group. Version 2 (2003)
2. Object Management Group, "Semantics of Business Vocabulary and Rules Formal Specification v1.0", OMG document formal/08-01-02, January 2008. URL: http://www.omg.org/spec/SBVR/1.0/. Accessed: 14/5/2010
3. Open Philosophies for Associative Autopoietic Digital Ecosystems, 2008. "Automatic code structure and workflow generation from natural language models". URL: http://files.opaals.eu/OPAALS/Year_2_Deliverables/WP02/D2.2.pdf . (14/5/2010)
4. Marinos, A., Krause, P., "An SBVR Framework for RESTful Web Applications", Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web, Springer-Verlag Berlin, Heidelberg, 2009, pp. 144-158.
5. Gilbert, D., Tafarodi, R. and Malone, P. 1993. 'You can't not believe everything you read'. '*Journal of Personality and Social Psychology,* **65**(2), 221-233.
6. Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 2004.
7. Marinos, A. Krause, P. "An SBVR Framework for RESTful Web Applications". In Proc. RuleML 2009, LNCS 5858, pp144-158, Springer, 2009.
8. Codd, E.F. "Relational Completeness of Data Base Sublanguages". In Proc. Courant Computer Science Symposia Series 6, R. J. Dustin (ed), Prentice Hall, 1972.
9. Huth, A.R.A, Ryan, M.D., *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2002.
10. Nakano, R. "Translation with Optimization from Relational Calculus to Relational Algebra having Aggregate Functions". *ACM Tran on Database Systems*, 15(4):518-557, 1990.
11. Halpin, T. A. "A Logical Analysis of Information Systems: Static Aspects of the Data-Oriented Perspective". PhD Thesis, University of Queensland, 1989.
12. Bloesch, A.C., Halpin, T.A. "ConQuer: a Conceptual Query Language". In *Proc. 15th Int'l Conf. on Conceptual Modelling*, LNCS 1157, pp. 121-133, Springer, 1996.
13. Kleiner, M., Albert, P., Bezivin, J. "Parsing SBVR-based Controlled Languages". In *Proc. MODELS 2009*, LNCS 5795, pp. 122-136, Springer, 2009.
14. Demuth, B., Liebau, H-B. "An Approach for Bridging the Gaps Between Business Rules and the Semantic Web". In Proc. RuleML 2007, LNCS 4824, pp. 119-133, Springer, 2007.
15. Cabot, J., Pau, R., Raventos, R. "From UML/OCL to SBVR Specifications: A Challenging Transformation". *Information Systems* 35 (2010): 417-440, Elsevier, 2010.