An SBVR to SQL Compiler

Alexandros Marinos, Sotiris Moschoyiannis, Paul Krause

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

Abstract. This paper presents the implementation of a compiler of SBVR Structured English to SQL data models and queries, with SBVR Logical Formulation as an intermediate step. The compiler is implemented in OMeta/JS and targets browsers that support Web SQL Databases. We discuss each stage of our compiler as well as the optimizations and necessary tradeoffs.

Keywords: SQL, SBVR, compiler, OMeta

1 Introduction

While the SBVR standard [1] has been in development for several years, no reference implementation of an SBVR parser has appeared in the open source domain, raising a barrier to entry for experimentation with the language. The authors have also previously published a number of papers [2],[3] which require an SBVR parser and an operationalization of its output to SQL to implement. With this motivation, we present in this paper a compiler that uses as input a minimally augmented plaintext representation of SBVR Structured English (SBVR-SE) models and converts them first to SBVR Logical Formulation (SBVR-LF) and eventually to an SQL model, both SQL-DML (data model) and SQL-DDL (queries).

The architecture of the compiler involves 3 stages: Parsing, Preprocessing and Optimization, and SQL Code generation. The first step converts SBVR-SE to SBVR-LF. The second stage eliminates redundancies and reduces some elements that are hard to represent in SQL to others that are more natural. The third stage converts the processed logical formulation into SQL statements. The rationale for requiring SQL as a target is detailed in [2]. To summarize, the design of Generative Information Systems depends on formulating SBVR Vocabularies into SQL data models and SBVR rules into SQL queries. The queries are formulated in a way that returns true if the rule is consistent with the dataset, and false if it is not. Once this transformation is accomplished, each alteration to the state of the database can be checked against the rules of the governing SBVR model to verify that it is consistent with the constraints that it places. This system follows the principles set out by C.J. Date in "What not How: The Business Rules Approach to Application Development" [4].

The compiler has been implemented in OMeta/JS [5],[6], an innovative variation of Parsing Expression Grammars (PEGs) that extends the pattern matching

capabilities so that they can be applied to all stages of compilation, including lexing, parsing, optimization via the visitor pattern and naïve code generation. Each stage traditionally required different tools which use different languages and implementation patterns, significantly impeding experimentation with new languages.

2 SBVR-SE to SBVR-LF Parser

While SBVR-SE is not considered a normative part of SBVR, it has become in many ways the most well-known facet of the language and the core of much of its perceived usefulness. So, while we do note that SBVR-SE is not the only way to verbalize an SBVR model, it is a de facto standard. As such, we have used it as the input of our compiler implementation. There has been significant discussion over the feasibility of parsing SBVR-SE into logical formulation, so we consider this part of our compiler to be a particularly useful contribution towards progressing that debate. The input of our parser is a plain text version of the model with minimal markup to denote whether each line should be interpreted as a rule, a fact type, or a term. An example model would be written as follows:

```
T: student

T: module

T: study programme

F: student is registered for module

F: student is enrolled in study programme

F: module is available for study programme

R: It is obligatory that each student is registered for
at most 5 modules

R: It is obligatory that each student that is registered
for a module is enrolled in a study programme that the
module is available for

F: student is under probation

R: It is obligatory that each student that is under
probation is registered for at most 3 modules
```

A line starting with ‘T:’ denotes a term, a line starting with ‘F:’ denotes a fact type, and ‘R:’ denotes a rule. Aside from this minimal mark-up, we do not require the terms, verbs and keywords to be marked up as is done in the SBVR specification itself. This may theoretically allow certain ambiguities but our grammar resolves

those by favouring terms over verbs in case of conflict. In practice, the possibility of a conflict is vanishingly small as the words that can be used both as verbs and nouns unmodified are few, especially in the formal business vocabulary concerned here. Additionally, for a conflict to arise, the tokens must be in a position where there is ambiguity about their intended role, an extremely unlikely case if at all possible.

The first rule of our model, when coloured as per the specification is this:

It is obligatory that each **student** *is registered for* at most five **modules**

According to the specification, an equivalent logical formulation is the following:

```
The rule is a proposition meant by an obligation formulation.
. That obligation formulation embeds a universal quantification.
. . The universal quantification introduces a first variable.
. . . The first variable ranges over the concept 'student'.
. . The universal quantification scopes over an at-most-n quantification.
. . . The at-most-n quantification has the maximum cardinality 5.
. . . The at-most-n quantification introduces a second variable.
. . . . The second variable ranges over the concept 'module'.
. . . The at-most-n quantification scopes over an atomic formulation.
. . . . The atomic formulation is based on the fact type 'student is registered for module'.
. . . . . The atomic formulation has a role binding.
. . . . . The role binding is of the role 'student' of the fact type.
. . . . . The role binding binds to the first variable.
. . . . . The atomic formulation has a second role binding.
. . . . . The second role binding is of the role 'module' of the fact type.
. . . . . The second role binding binds to the second variable.
```

The output of our parser is a JavaScript nested array:

```
[rule,
   [obl,
      [univQ,
         [var,
         [num, 1],
         [term, "student"]],
      [atMostQ,
         [maxCard,
            [num, 5]],
         [var,
            [num, 2],
            [term, "module"]],
         [aFrm,
            [fcTp,
               [term, "student"],
               [verb, "is registered for"],
               [term, "module"]],
            [bind,
               [term, "student"],
               1],
            [bind,
               [term, "module"],
               2]]]]]
```

This output can be seen to be clearly equivalent to the above verbalization, with the first element of each array denoting the type of element in a shortened form. While the output can be mapped to the specification-mandated XMI format in a fairly straightforward manner, since the output is directly processed rather than stored or communicated to other tools, there was no need to do so at the current stage.

It should be noted that our parser does recognise pluralisation of terms but only trivial forms which are produced by appending an 's' to the singular form. This is not a design limitation of the parser architecture and a more sophisticated inflection library can be plugged in to cover more complex cases. However there are always exceptions for which a general inflector cannot provide. The ideal solution would be for the users to be allowed to declare the plural form as an attribute of the term in the model, which would override the default provided by the inflector, but this would require an extension of the SBVR meta-model itself.

An element of ambiguity in the specification is about the exact form of fact types. While examples of term-verb (unary) and term-verb-term (binary) fact types are provided, there is also discussion of n-ary fact types where n is greater than 2. What is not clear is whether fact types ending with a verb and containing more than one term (e.g. term-verb-term-verb) are considered valid. For the implementation of the parser we have considered such fact types valid as they increase expressivity without any obvious drawback.

Our parser does not implement the full breadth of the SBVR specification but rather a large and usable subset with a focus on expressing complex rules. The parser can be extended to include less common features of SBVR and indeed this is part of the future work planned. The features currently implemented are: declarations of terms and fact types, all modalities for rules, all quantifiers, and the keyword 'that' as a means of introducing atomic formulations that constrain variables. With this subset of SBVR, even complex rules such as the following can be parsed into the appropriate logical formulation:

It is obligatory that each **student** that *is registered for* a **module** *is enrolled in* a **study programme** that the **module** *is available for*

3 Optimizations and Pre-processing

Once the logical formulation is attained, it cannot directly be converted to SQL without some pre-processing, as not all the operators are directly translatable to SQL. Initial theoretical attempts at conversion used first-order logic as an intermediate step. However SBVR's mapping to ISO Common Logic is incomplete, and we found that certain features such as the at-most-n quantifiers would require lengthy conversion which although feasible is impractical for our purposes. This mapping has been published in [3]; however the path taken there is not useful for a practical conversion. The crucial insight that allowed bypassing ISO Common Logic was that SBVR-LF itself can be treated as logic, even though this is not discussed in the specification.

One transformation applied is the elimination of the universal quantifier (∀), as SQL cannot directly express the. As a result, any logic structure to be mapped to SQL must have this quantifier eliminated. To accomplish this, we convert every occurrence of $\forall x$ into the equivalent $\neg\exists x\neg$, directly on the SBVR logical formulation.

Similarly, we replace all occurrences of the at-most-n quantifier ($\exists^{0..n}$) with the negation of the more easily expressed at-least-m (where m=n+1) quantifier ($\neg\exists^{n+1..}$).

As a result of these substitutions, as well as the parsing of impossibilities and prohibitions as necessities of negation and obligations of negation respectively, many negations are introduced into the logical formulation. We then apply a double negation elimination rule as an optimization to remove these redundancies.

Finally, when the at-least-n quantifier carries a modality of 1, it is equivalent to the existential quantifier. Since the latter is more straightforward and efficient to express in SQL, we replace the former with the latter as another optimization.

4 SQL Code Generation

The final step is to convert the preprocessed logical formulation to SQL. The terms and fact types are converted to tables. The fact type tables containing references to the primary keys of the term tables. In this way a simple schema can be derived from the SBVR vocabulary. More complex transformations can be made and have been described in [2], but for this proof of concept a simple transformation is sufficient. The SQL dialect produced is that of the SQLite engine, which is the standard used for modern HTML 5 WebSQL Database [7] implementations found in browsers such as Google Chrome, Apple Safari, and Opera. Since the compiler runs within a Javascript environment, this allows us to execute the results locally within a browser without server roundtrips.

The queries produced utilize the EXISTS keyword of SQL as in the case of relational division. To give an example of relational division, the query ‘a pilot can fly every plane in a given hangar’, would be written as:

```
SELECT DISTINCT pilot_name
FROM PilotSkills AS PS1
WHERE NOT EXISTS(
  SELECT * FROM Hangar
  WHERE NOT EXISTS(
    SELECT *  FROM PilotSkills AS PS2
    WHERE (PS1.pilot_name = PS2.pilot_name)
      AND (PS2.plane_name = Hangar.plane_name)));
```

The above query is notable for its logic-like structure and its usage of the NOT and EXISTS keywords that map to logical negation(¬) and the existential quantifier (∃), which are common in SBVR Logical formulations also. Here we describe the mappings from the logical formulation constructs to SQL fragments. For the existential quantifier, we use the following SQL fragment:

```
EXISTS (SELECT *
          FROM <term> AS var<n>
          WHERE (...))
```

Here essentially the SELECT statement is used to introduce the relevant variable so that it can be later bound to.

The at-most-n quantifier maps to the following SQL fragment:

```
EXISTS (SELECT count(*) AS card FROM <term> AS var<n>
           WHERE (...)
           GROUP BY NULL
           HAVING card <= 5))
```

This fragment uses '`GROUP BY NULL`' to state that the '`HAVING card <= 5`' clause should apply to the whole result, as SQLite does not support a '`HAVING`' clause without a '`GROUP BY`' clause. Other quantifiers (existential, at-least-n, exactly-n, numeric range) have similar mappings.

Atomic formulations map to the following SQL fragment:

```
EXISTS (SELECT *
          FROM <fact_type> AS f
          WHERE var1.id = f.<role1_id>
            AND var2.id = f.<role2_id>)
```

Here, the `WHERE` clause declares the variable bindings. The assumption is that the appropriate quantifiers envelop the fragment such that the variables that are referred to have values to bind to. This is in line with the way the SBVR logical formulation is structured so a legally structured SBVR rule covers this requirement.

The negation operator simply maps to the SQL keyword `NOT`. Also, the '`SELECT`' keyword is prepended to every query.

Applying the above, the result of the conversion of the example rule 'It is obligatory that each **student** *is enrolled for* at most 5 **courses**' to SQL is:

```
SELECT NOT EXISTS(
  SELECT * FROM student AS var1
  WHERE EXISTS(
    SELECT count(*) AS card FROM module AS var2
    WHERE EXISTS(
      SELECT * FROM student_is_registered_for_module AS f
      WHERE var1.id = f.student_id
      AND var2.id = f.module_id)
    GROUP BY NULL
    HAVING card >= 6))
```

Converting the more complex rule 'It is obligatory that each student that *is registered for* a module *is enrolled in* a study programme that the module *is available for*' into a query yields:

```
SELECT NOT EXISTS (
  SELECT * FROM student AS var1
  WHERE EXISTS (
    SELECT * FROM module AS var2
    WHERE EXISTS (
      SELECT * FROM student_is_registered_for_module AS f
      WHERE var1.id = f.student_id AND var2.id = f.module_id
      AND NOT EXISTS (
        SELECT * FROM study_programme AS var3
        WHERE EXISTS (
          SELECT * FROM module_is_available_for_study_programme AS f
          WHERE var2.id = f.module_id
          AND var3.id = f.study_programme_id
          AND EXISTS (
            SELECT * FROM student_is_enrolled_in_study_programme AS f
            WHERE var1.id = f.student_id
            AND var3.id = f.study_programme_id))))))
```

In this example we see a new structured English formulation which is signified by the keyword 'that'. In this role, the keyword 'that' introduces a restriction on the variable that precedes it. In our example, the variable 'student' is restricted by an existential quantifier that introduces the variable 'module' and ranges over the atomic formulation 'student *is registered for* module'. In terms of SQL, this translates to an injection of two more nested SELECT statements in the stack of nested statements that would otherwise occur. The same pattern occurs once more in the example.

It is interesting that this nesting formulation in SQL departs from the tree structure of SBVR's logical formulation. This is because while SBVR-LF is organised as a tree, latter nodes are assumed to have access to variables declared earlier in the formulation, even if the declaration was in another branch entirely. Formulating this as an SQL statement however returns an error as the SQL engine does not give access to variables declared on other branches. The solution to this problem is to nest every clause inside the previous one so that it has access to all variables previously declared, as seen in the example above.

5 Conclusion & Future Work

This implementation is the result of long effort at comprehension of the SBVR specification which is a challenge unto itself. With the progress of this project has come deeper understanding of the structures and rationale of SBVR's design. We hope that the current result as well as that of the ongoing effort will be of use to other

users of SBVR, either as an implementation or as a source of ideas on implementing SBVR. Next steps include expanding the coverage of the grammar, especially with the inclusion of logical operators, definitions for terms, more sophisticated inflection, and the support for the concept-type attribute which will allow for complex hierarchies of terms and highly expressive models as a result. The overall aim is to integrate the result into a web-browser based development environment for developing SBVR-based information systems according to the Generative Information Systems paradigm.

Acknowledgements

We would like to thank Alessandro Warth for his patience and speed in answering our frequent questions about OMeta grammars. 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. Object Management Group, “Semantics of Business Vocabulary and Rules Formal Specification v1.0”, 2008, URL: http://www.omg.org/spec/SBVR/1.0/, Accessed: 20/8/2010.
2. 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.
3. Moschoyiannis, S., Marinos, A., Krause, P., “Generating SQL Queries from SBVR Rules”, RuleML 2010, *to appear*.
4. C.J. Date, “What Not How: The Business Rules Approach to Application Development,” Addison-Wesley Professional, 2000.
5. Warth, A. "Experimenting with programming languages" Technical report, Viewpoints Research Institute, 2008.
6. Warth, A. Piumarta, I., "Ometa: an object-oriented language for pattern matching" DLS’07 DSL, 2007.
7. World Wide Web Consortium, “Web SQL Database”, 2010, URL: http://dev.w3.org/html5/webdatabase/, Accessed: 20/8/2010.