Automated Verbalization for ORM 2

Terry Halpin and Matthew Curland

Neumont University, Utah, USA
{terry, Matthew.Curland}@neumont.edu

Abstract. In the analysis phase of information systems development, it is important to have the conceptual schema validated by the business domain expert, to ensure that the schema accurately models the relevant aspects of the business domain. An effective way to facilitate this validation is to verbalize the schema in language that is both unambiguous and easily understood by the domain expert, who may be non-technical. Such verbalization has long been a major aspect of the Object-Role Modeling (ORM) approach, and basic support for verbalization exists in some ORM tools. Second generation ORM (ORM 2) significantly extends the expressibility of ORM models (e.g. deontic modalities, role value constraints, etc.). This paper discusses the automated support for verbalization of ORM 2 models provided by NORMA (Neumont ORM Architect), an open-source software tool that facilitates entry, validation, and mapping of ORM 2 models. NORMA supports verbalization patterns that go well beyond previous verbalization work. The verbalization for individual elements in the core ORM model is generated using an XSLT transform applied to an XML file that succinctly identifies different verbalization patterns and describes how phrases are combined to produce a readable verbalization. This paper discusses the XML patterns used to describe ORM constraints and the tightly coupled facilities that enable end-users to easily adapt the verbalization phrases to cater for different domain experts and native languages.

1 Introduction

To help ensure that a conceptual schema accurately models the universe of discourse, the schema should be validated by a business domain expert. One effective way to facilitate this validation is to *verbalize* the schema in language that is both unambiguous and easily understood by the domain expert, who may be non-technical. Various proposals and tools exist to facilitate verbalization of business rules. The RuleSpeak sentence templates [19] provide basic rule verbalization patterns, but their informal nature obviates automatic transformation into executable code. The Object-oriented Systems Analysis (OSA) model [6] supports high level, informal rules as well as formal rules in a predicate calculus notation. Our approach instead uses a single language that is both formal and conceptual, so that it can serve for communication and validation with domain experts, as well as being executable. While its motivation is similar to that of Common Logic Controlled English (CLCE) [20], its syntax is higher level (e.g. pronouns are often used instead of variables), and it is designed for ease of localization into different native languages.

R. Meersman, Z. Tari, P. Herrero et al. (Eds.): OTM Workshops 2006, LNCS 4278, pp. 1181 – 1190, 2006.

In industry, the most popular high level information modeling approaches are the Entity-Relationship (ER) approach [5] and the Unified Modeling Language (UML) [18], with dialects of Object-Role Modeling (ORM) [e.g. 2, 8] arguably being in third place. The Barker ER approach [3] provides a discipline for relationship readings that enables internal uniqueness and mandatory constraints to be verbalized. The NaLER [1] approach extends this somewhat. However these approaches handle only a small fragment of ORM constraints, are restricted to binary relationships, and are unsuited to verbalizing fact instances. For textual expression of rules, UML advocates the use of the Object Constraint Language (OCL) [21], but the syntax of this language is too mathematical to enable reliable validation by non-technical business domain experts.

While many ORM languages exist for model specification, such as RIDL [16] and LISA-D [15], few tools support automatic verbalization of ORM models. In the 1990s, one of the authors specified automated support for verbalization in ORM [7], and later extended this for Microsoft's ORM source model solution [14]. More recently, we specified and implemented a substantially improved verbalization mechanism for Neumont ORM Architect (NORMA) [17], an open-source tool for entering second generation ORM (ORM 2) [10] models and transforming these to application code. In addition to catering for new features in ORM 2, such as deontic modality, role value and explicit subtyping constraints, we now support improved verbalization in both positive and negative forms, including verbalization of the set-based nature of spanning uniqueness constraints, and the absence of relevant constraints [9].

While preliminary versions of a few of our simpler verbalization patterns have appeared in popular journals [e.g. 11], this paper is the first to discuss the detailed specification and implementation of the verbalization patterns. Section 2 provides a brief overview of verbalization support in NORMA. Section 3 specifies some typical verbalization patterns. Section 4 details how the verbalization engine is implemented in NORMA, including reasons for certain design decisions. Section 5 summarizes the main results, suggests topics for further research, and lists references.

2 Overview of Verbalization in NORMA

Our verbalization language for ORM 2 was architected to meet five main design criteria: expressibility, clarity, flexibility, localizability, and formality [9]. For expressibility reasons, both alethic and deontic *modalities* are supported [12]. Localization concerns as well as support of natural verbalization for predicates of any arity dictated use of *mixfix* predicates (e.g. … introduced … to … on …). For clarity and flexibility reasons, constraint verbalizations may be presented in *positive or negative form* (showing how to satisfy or violate the constraint), and may use *relational or attribute style* (employing predicate readings or role names) or a mix of the two.

NORMA automatically verbalizes whatever part of the ORM model is currently selected. As a simple example, Fig. 1 displays a NORMA screen shot showing the automated verbalization in positive form of three fact types along with seven constraints (four alethic and three deontic). The mandatory and uniqueness constraints on the top binary fact type in Fig. 1 are verbalized in positive form thus: **Each** Person was born in **exactly one** Country.

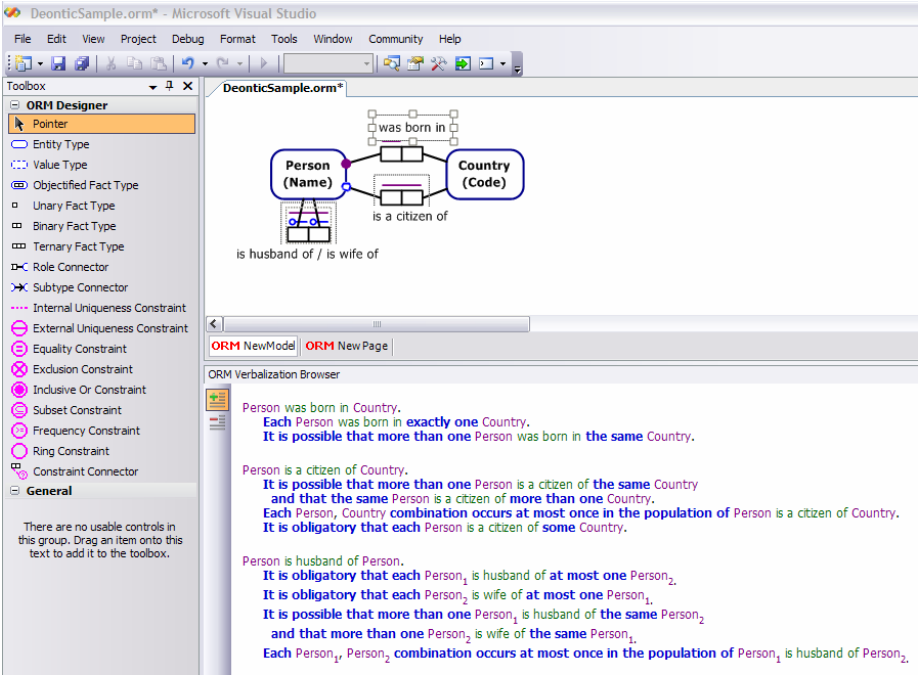

Fig. 1. Screenshot of positive verbalization in NORMA

The *absence of a uniqueness constraint* on the right-hand role is verbalized: **It is possible that more than one** Person was born in **the same** Country. Pressing the "–" button redisplays the constraints in negative form: **For each** Person, **it is impossible that that** Person was born in **more than one** Country; **It is impossible that any** Person was born in **no** Country.

The uniqueness constraint spanning both roles of the citizenship fact type specifies both that the association is many:many, and that its population must be a *set* rather than a bag of facts. These two aspects are verbalized separately: **It is possible that more than one** Person is a citizen of **the same** Country **and that the same** Person is a citizen of **more than one** Country; **Each** Person, Country **combination occurs at most once in the population of** Person is a citizen of Country. The constraints discussed so far are *alethic* (they are logical or physical necessities and hence cannot be violated).

Deontic constraints (marked graphically with "o") indicate rules that ought to be obeyed but may possibly be violated. As a simple example of deontic verbalization, the left deontic uniqueness constraint in Fig. 1 is verbalized in positive form thus: **It is obligatory that each** Person is a husband of **at most one** Person. In negative form, we have: **For each** Person_1, **it is forbidden that that** Person_1 is a husband of **more than one** Person_2.

Forward hyphen binding treats the word before the hyphen as an adjective (e.g. a uniqueness constraint on the first role of Person has first- GivenName verbalizes as "**Each** Person has **at most one** first GivenName."). As new features, NORMA also caters for *reverse hyphen binding* (e.g. Student has Preference -1 may be used instead of Student has first- Preference) and predicates with *front text* (text before the first object placeholder).

For example, the uniqueness constraint on the first role of the birth of Person occurred in Country verbalizes: **For each** Person, the birth of **that** Person occurred in **at most one** Country.

NORMA also caters for constraint types that are new in ORM 2, such as *value constraints on roles* (e.g. **The possible values of** Person.height(cm) **are** [20..270].) and value constraints involving open, continuous ranges (e.g. **The possible values of** NegativeTemperature(Celsius) **are at least** -273.15 **and below** 0.).

3 Sample Verbalization Patterns

In specifying the verbalization patterns for ORM 2, great care was taken to cover all possible cases. In this section, we illustrate the style of high level specification provided as input to developers. The actual specification documents are vast, and will be released as technical reports, e.g. [13]. Here we include just a tiny fragment from the specification for verbalization of inclusive-or (ior, i.e. disjunctive mandatory) constraints for the sub-case of unary and/or binary and/or *n*-ary predicates.

The *positive, alethic relational* pattern for the case where each constrained role starts a predicate reading is shown in Fig. 2. The object types are not necessarily distinct. Each predicate R_i has n_i roles ($n_i \geq 0$) following the role played by A. So the predicates may all be of different arity (unary upwards), and the object types may or may not be the same. We verbalize all the binaries before all the unaries. The *deontic* version prepends "**it is obligatory that**" to the positive form.

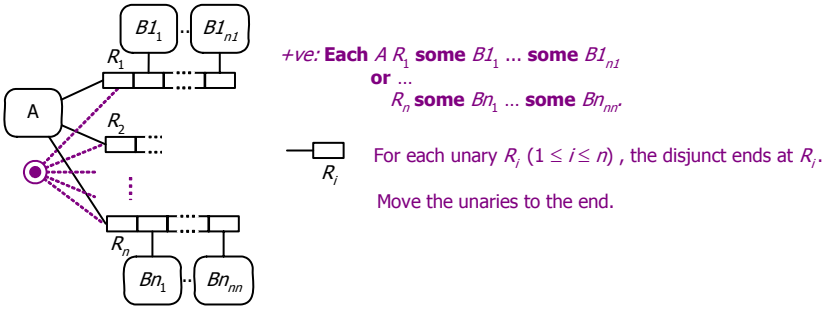

Fig. 2. Inclusive-or verbalization pattern when each constrained role starts a predicate reading

Examples: **Each** Partner became the husband of **some** Partner on **some** Date
or became the wife of **some** Partner on **some** Date.

It is obligatory that
each Vehicle was purchased from **some** BranchNr of **some** AutoRetailer **or** is rented.

If even one predicate in the previous case has *front text*, the pattern in Fig. 3 (some constrained roles do not start a predicate reading) is used instead. If any R_i predicate contains front text, this is included as part of the predicate reading. The *positive, alethic relational* pattern is shown. The object types are not necessarily distinct. If A plays more than one role in at least one of the R_i predicates, we subscript its instances to distinguish them. Each predicate R_i may have ni roles ($ni \geq 0$) plus the role played by A.

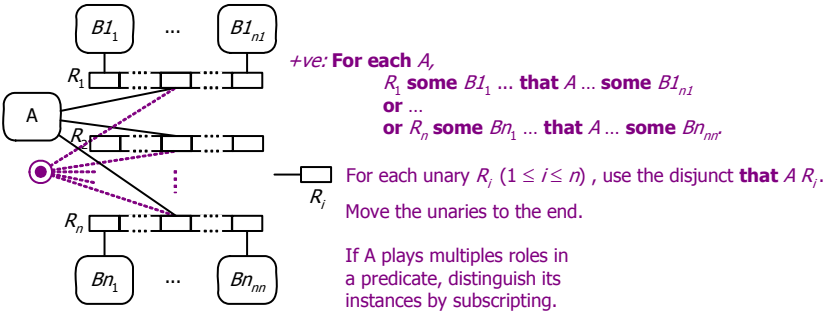

Fig. 3. Ior verbalization pattern when some constrained roles don't start a predicate reading

Examples: **For each** Partner₁,
on **some** Date **that** Partner₁ became the husband of **some** Partner₂
or on **some** Date **that** Partner₁ became the wife of **some** Partner₂.

It is obligatory that for each Vehicle,
some BranchNr of **some** AutoRetailer sold **that** Vehicle
or that Vehicle is rented.

4 Implementation in NORMA

This section briefly outlines how verbalization support is implemented in NORMA. Implementing a verbalization pattern needs care because the number of potential variations is very high. A conservative estimate is that a full ORM verbalization implementation coded by hand requires 10,000-15,000 lines of code, or about 6 person months. Incremental maintenance costs would also be extremely high due to the size of the code. To succeed both short term and long term, we decided to use a pattern-driven generative approach to implement the code whenever possible.

The rules for verbalization of a constraint pattern are constant, but the actual text used for different parts of the verbalization depends on environment-specific factors.

1. Although our reference implementation uses English, verbalization in other languages should incur only incremental implementation costs.

2. The same verbalization engine should be able to render different output formats. NORMA's verbalization window will display html, but we may want different html for a report view, and plain text in other views.

3. Personal verbalization preferences are also an environment factor. For example, by default we do not show the implied "It is necessary that" before positive alethic constraints. However, any skilled user should be able to choose to see the explicit form, or rephrase it (e.g. "The following condition is necessary: ").

4. A skilled user should be able to easily adapt the verbalization output to the current target audience (e.g. replace the default deontic "it is obligatory that" with "It ought to be that" or even a personalized '*CompanyName* policy requires that').

The goal of *dynamic verbalization* is to output verbalization that is easily validated by the reader. Readers typically prefer verbalization in their native language. If the reader wants a full report instead of individual diagram selections, then the same verbalization engine should be able to produce both a standalone printed report and a website of mini-reports for each object type, fact type, and constraint.

Two approaches may be used to *generate* a verbalization phrase. Both combine user-provided predicate text and object type names or instances with text particles provided by the verbalization engine. The first approach uses concatenation, where pieces of a phrase are constructed by combining particles in a specific order. The second approach uses field replacement, where the particles specify the location and order of the particles surrounding them. Let's break down the verbalized phrase "**Each** Person was born in **some** Country" using both approaches.

To use *concatenation* to verbalize this phrase requires seven strings to be combined in the correct order. Arbitrary predicate text requires arity+1 strings. In this case, the three predicate strings are {"", " was born in ", ""}. In addition to the predicate strings, the user-provides the object type names {"Person", "Country"}. The verbalization engine then provides the universal quantifier "**each** " and the existential quantifier "**some** ". The specified pattern (a simple mandatory constraint on a binary predicate where the mandatory role begins a predicate reading) is now built as follows. Though not shown here, each verbalization starts with a capital letter and ends in a period ".".

"" + "**each** " + "Person" + " was born in " + "**some** " + "Country" + ""

In practice, generating formatted text is significantly more complicated because each particle must include formatting specifications before and after the text, thus tripling the number of text particles necessary to complete the phrase.

The *field replacement* approach uses numbered replacement fields. We'll show these in the format required for the .NET System.String.Format function, which uses (regular expression) "\{\d+\}" to denote a zero-based replacement field in a format string. For example, "{0}" in a format string is the placeholder for the first replacement field. The Format function takes a format string as the first argument, followed by arguments for the replacement fields. For our current phrase, the predicate text is "{0} was born in {1}" and the quantifiers become "**each** {0}" and "**some** {0}". The equation now looks like:

REPLACE("{0} was born in {1}", REPLACE("**each** {0}", "Person"), REPLACE("**some** {0}", "Country"))

Note three immediate advantages to this approach. First, the model stores a single predicate text with replacement fields instead of arity+1 strings. Second, adding formatting specifications to verbalization-provided quantifiers does not increase the number of snippets or increase the algorithmic complexity. Third, order dependency is eliminated: "**each**" may come before "Person" in English, but other languages may have some quantifiers either after or around a given replacement field. Using the concatenation approach would require new code for each language or before/after specification for every phrase. The replacement approach removes all ordering and formatting considerations from the code, placing the onus for ordering on the snippets.

Using field replacement exclusively in the ORM2 verbalization engine allows all *snippets* to be specified as user-modifiable data. Concatenation is used only for list generation and incorporates user-specified list separators. Using field replacements

simplifies the verbalization engine and enables data-driven snippet sets to be specified according to both language and user preferences. Snippets are considered dynamic data; how they are combined is specified statically for each verbalized model element.

Given the decision to employ field replacement, the next step was to identify components and determine which pieces should be generated and which ones hand coded. Two primary factors weighed in this decision: First, how often is the code reused? Code generation is generally cost effective only if the generator is applied to more than one piece of data. Second, how complex is the mapping from the verbalization specification to the generated code? The more complex the pattern, the more vital it is to concisely represent that pattern as data, allowing the resulting verbalization implementation to be indirectly modified by changing the input data to the generator.

The implementation was eventually broken down into the following components:

1) The *selection manager* determines which elements are to be verbalized, applies all phrase delimiters such as capitalization and punctuation of sentences, adds lines between selected elements, and indents verbalization phrases for aggregated elements. A good selection management routine allows individual elements to concentrate on self-verbalizing without worrying about the context in which they are verbalized. Selection management routines reference dynamic snippets for document header and footer information, punctuation, and white space, but are otherwise hand coded. The engine recognizes the IVerbalize interface implemented by individual elements.

2) The *snippet manager* determines which snippet variations are available on the user's machine. Snippets can be specified by the core model and any other extension model (the core is not given preferential treatment). User-authored XML files provide customization. The snippet manager presents the user with the available snippets (in the options page), then validates and loads the files. The reference implementation of each snippet set is coded into the application to protect the engine from rogue customizations. The snippet manager is hand-coded, while the snippet set implementation is generated. XML for the reference snippet set is installed but never loaded.

The selection and snippet management components provide the necessary framework for individual elements to verbalize themselves. From the selection manager perspective, an element can be verbalized if it implements the IVerbalize interface. This paper focuses on the XML patterns used to represent the complex patterns in the verbalization specification. Individual element verbalization is difficult and involves precise translation from specification to code. Representing the specification in XML provides a formal, unambiguous representation of the expected verbalization in a form that can be easily verified and modified. Given the XML, generating IVerbalize interface implementations is an XSLT exercise that is beyond the scope of this paper.

Various XML constructs are needed to fully reflect the verbalization specification. The Snippet element, referencing a snippet name, is the only obvious construct. The number of children inside a Snippet tag corresponds to the number of replacement fields expected. The XML schema then revolves around different ways to specify replacement fields, often recursively. The difficulty is to specify conditions to determine which snippet combinations to use. Conditions come in many different forms.

1) *Differences in modality* (alethic or deontic) *and sign* (positive and negative verbalization) are primarily handled by providing different snippets with the same name. Retrieving a snippet by {name, modality, sign} instead of just {name} takes care of the largest variability in specifying which snippet to use. It is possible to specify

radically different verbalizations for sign and modality, but the majority of the cases, especially with modality, have the same pattern with minor variations in the snippet. For the common case, no additional XML is required to switch modality and sign.

2) *Differences in the shape of constraints*. These differences revolve around the number and arrangement of roles and are represented by the *ConstrainedRoles* tag. Possible attributes of *ConstrainedRoles* include factArity (the number of roles in the fact being constrained, used for internal uniqueness and simple mandatory constraints), factCount (the number of fact types being constrained), maxFactArity and minFactArity (similar to factArity, but used for multi-fact constraints), and sign (set to either positive or negative, to use different patterns on sign).

3) *Differences in the availability of reading text* for a given lead role or reading order. ORM verbalization uses the most natural reading available, falling back on a more complicated form if the optimal reading is not available. The *ConditionalReading* tag, occurring directly inside *ConstrainedRoles*, contains *ReadingChoice* tags, each of which specifies a match attribute with reading conditions. Reading precedence corresponds to the order of the ReadingChoice tags. The last ReadingChoice tag can omit the match attribute, indicating the lowest priority fallback condition.

4) Once a pattern and reading are selected, decisions still need to be made based on *how a given role is used in the constraint*. The most common classification is whether a role in the FactType is *included* or *excluded* in the constraint. Additional classifications occur when roles are iterated (with the *IterateRoles* tag). Inside an iteration, a given role can be *included* (a constrained role), *excluded* (not a constrained role), *primary* (the current role in the set being iterated), and *secondary* (a role in the set being iterated that is not the current role). Specifying sets of roles in this fashion gives us a flexible, set-based algorithm that works for both single-role and multi-role sets.

The following sample, which is part of the MandatoryConstraint specification, illustrates uses of these tags and conditions. <!-- Comments --> are in the XML.

```
< !-- Specify the type of constraint. This will implement IVerbalize on the MandatoryConstraint class. -->
<Constraint type="MandatoryConstraint" patternGroup="SetConstraint">
<!-- Positive verbalization of a mandatory constraint on a unary fact type --><!-- Each A R -->
      <ConstrainedRoles constraintArity="1" factArity="1" sign="positive">
<!-- ImpliedModalNecessityOperator: '{0}' (positive alethic), 'it is obligatory that {0}' (positive deontic) -->
            <Snippet ref="ImpliedModalNecessityOperator">
<!-- UniversalQuantifier is 'each {0}' -->
                  <Snippet ref="UniversalQuantifier">
<!-- Fill the default predicate text replacement fields with the role player names -->
                        <Fact/></Snippet></Snippet></ConstrainedRoles>
<!-- A single-role simple mandatory constraint on a binary fact type --><!-- Each A R some B -->
      <ConstrainedRoles constraintArity="1" factArity="2">
<!-- The pattern will change based on the available predicate text -->
            <ConditionalReading>
<!-- A reading is available that begins with the constrained role -->
                  <ReadingChoice match="RequireLeadReading">
                        <Snippet ref="ImpliedModalNecessityOperator">
<!-- Populate the predicate text using the reading from the current context -->
                              <Fact readingChoice="Context">
<!-- Qualify all roles included in the constraint with the UniversalQuantifier 'each {0}' -->
                                    <PredicateReplacement match="included">
                                          <Snippet ref="UniversalQuantifier"/></PredicateReplacement>
```

```
<!-- Qualify all remaining roles included in the constraint with the ExistentialQuantifier 'some {0}' -->
                                    <PredicateReplacement>
                                        <Snippet ref="ExistentialQuantifier"/>
                                    </PredicateReplacement></Fact></Snippet></ReadingChoice>
<!-- The negative form of the snippets changes 'each' to 'any' and 'some' to 'no'. Combining these two
snippet variations with the ImpliedModalNecessityOperator changes 'Each A R some B' to 'It is impossible
that any A R no B'. Given that the pattern change is limited to the snippet variations only, there is no rea-
son to provide an alternate pattern for negative or deontic forms of the verbalization phrase -->

<!-- Move onto the fallback form (no lead reading is available) --><!-- For each A, some B S that A -->
                <ReadingChoice>
<!—The ForEachCompactQuantifier snippet has two replacement fields on the same line -->
                    <Snippet ref="ForEachCompactQuantifier">
<!-- List all included roles. A listStyle is not needed here as the ConstrainedRoles conditions ensure there
will only be one item in the set. Otherwise, listStyle could be SimpleList. The hyphenBind attribute indi-
cates that any hyphen binding in the predicate text should also be applied when the roles are listed -->
                        <IterateRoles match="included" listStyle="null" hyphenBind="true"/>
<!-- The rest is similar to the previous case, except 'each A' becomes 'that A' -->
                        <Snippet ref="ImpliedModalNecessityOperator">
                            <Fact>
                                <PredicateReplacement match="included">
                                    <Snippet ref="DefiniteArticle"/></PredicateReplacement>
                                <PredicateReplacement>
                                    <Snippet ref="ExistentialQuantifier"/>
                                </PredicateReplacement></Fact>
</Snippet></Snippet></ReadingChoice></ConditionalReading></ConstrainedRoles></Constraint>
```

This style of XML is used to verify that we have an exact match with the specification, and forms the input to the code generator. There are other conditional constructs that are not shown. For example, *ConditionalSnippet* allows us to use the same replacement fields while varying the snippets and *ConditionalReplacement* allows us to use different replacement contents inside a single snippet. Additional conditional and iteration constructs are being added as needed to formalize the verbalization requirements. Including standard code to verify error conditions that is generated with all constraints, the sample XML above (~30 lines of data, comprising 2 of the 6 *ConstrainedRoles* elements on MandatoryConstraint) produces ~350 lines of code. In general, we're getting at least a 1/10 ratio between XML and generated code. In addition to being able to easily verify the implementation against the spec, we also have the advantage that a minor change in the code generator can produce widespread changes in the code base. For example, ~50 lines of new XSLT and some new hand-coded support functions added hyphen binding support to all constraint verbalizations.

5 Conclusion

This paper discussed the automated support for verbalization of ORM 2 models provided by the open source NORMA tool, which caters for verbalization patterns that go well beyond previous verbalization work in ORM, and uses a generation process based on application of XSLT transforms to an XML file that succinctly identifies different verbalization patterns and describes how phrases are combined to produce a readable verbalization. At the time of writing, most ORM constraint patterns have

been specified and implemented. Future work will extend the verbalization to cover all aspects of ORM 2 models (schemas plus populations).

References

1. Atkins C. and Patrick J. P., 'NaLER: A natural language method for interpreting entity-relationship models', *Campus-Wide Information Systems* 17(3), 2000, pp. 85-93.
2. Bakema, G., Zwart, J. & van der Lek, H. 2000, *Fully Communication Oriented Information Modelling*, Ten Hagen Stam, The Netherlands.
3. Barker, R. 1990, *CASE*Method: Entity Relationship Modeling*, Addison-Wesley, Wokingham.
4. Bloesch, A. & Halpin, T. 1997, 'Conceptual queries using ConQuer-II', *Proc. ER'97: 16th Int. Conf. on conceptual modeling*, Springer LNCS, no. 1331, pp. 113-26.
5. Chen, P. P. 1976, 'The entity-relationship model—towards a unified view of data'. *ACM Transactions on Database Systems*, 1(1), pp. 9–36.
6. Embley, D. 1998, *Object Database Management*, Addison-Wesley, Reading, MA.
7. Halpin T. & Harding J. 1993, 'Automated support for verbalization of conceptual schemas', *Proc. 4th Workshop on Next Generation CASE Tools*, eds S. Brinkkemper & F. Harmsen, Univ. Twente Memoranda Informatica 93-32, Paris, pp. 151-161.
8. Halpin, T. 2001, *Information Modeling and Relational Databases*, Morgan Kaufmann, San Francisco.
9. Halpin, T. 2004, 'Business Rule Verbalization', *Information Systems Technology and its Applications*, Proc. ISTA-2004, (eds Doroshenko, A., Halpin, T., Liddle, S. & Mayr, H.), Salt Lake City, Lec. Notes in Informatics, vol. P-48, pp. 39-52.
10. Halpin T. 2005, 'ORM 2', *OTM 2005 Workshops*, eds R. Meersman, Z. Tari, P. Herrero et al., Springer LNCS 3762, Cyprus, 2005, pp. 676-87.
11. Halpin, T. 2006, 'Verbalizing Business Rules: Part 14', *Business Rules Journal*, Vol. 7, No. 4. URL: http://www.BRCommunity.com/a2006/b283.html.
12. Halpin, T. 2006, 'Business Rule Modality', *Proc. CAiSE'06 Workshops,* eds T, Latour & M. Petit, Namur University Press, pp. 383-94.
13. Halpin, T., Curland, M. & CS445 Class 2006, 'ORM 2 Constraint Verbalization: Part 1', *Technical Report ORM2-02*, Neumont University. Available online at http://www.orm.net/pdf/ORM2_TechReport2.pdf.
14. Halpin, T., Evans, K, Hallock, P. & MacLean, W. 2003, *Database Modeling with Microsoft® Visio for Enterprise Architects*, Morgan Kaufmann, San Francisco.
15. ter Hofstede, A. H. M., Proper, H. A. & Weide, th. P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
16. Meersman, R. M. 1982, *The RIDL conceptual language*. Research report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.
17. NORMA URL: https://sourceforge.net/projects/orm.
18. Object Management Group 2003, *UML 2.0 Superstructure Specification*. Online: www.omg.org/uml.
19. Ross, R., Lam, G. 2001, 'RuleSpeak Sentence Templates: Developing Rules Statements Using Sentence Patterns', Business Rule Solutions, Online at www.BRCommunity.com.
20. Sowa, J. F. 2004, 'Common Logic Controlled English', Draft available online at http://www.jfsowa.com/clce/specs.htm.
21. Warmer, J. & Kleppe, A. 2003, *The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn.,* Addison-Wesley.