Logical Data Modeling: Part 2

Terry Halpin
INTI International University

This is the second article in a series on logic-based approaches to data modeling. The first article [3] provided a brief overview of deductive databases, and illustrated how simple data models may be declared and queried in LogiQL [6], a leading edge example of deductive database system based on extended datalog [1]. The current article discusses how to declare inverse predicates, simple mandatory role constraints and internal uniqueness constraints on binary fact types in LogiQL, using the free, cloud-based version of LogiQL that is accessible at https://developer.logicblox.com/playground/.

Inverse Predicates and Simple Mandatory Role Constraints on Binary Fact Types

Figure 1 depicts a modified version of the Country data model discussed in the previous article [3]. Please review that article for basic background on the example as well as the graphical notations used. The example differs from that in the previous article mainly by allowing that some languages are not currently used in official documents of any country. Figure 1(a) depicts the data model in Object-Role Modeling (ORM) [4] notation. The exclamation mark "!" appended to the name of the Language object type indicates that Language is *independent*, so a language may be recorded to exist independently of playing any role other than its reference role (in this case, its role of being named). For example, the sample data includes the Akkadian language, which was popular in ancient Mesopotamia but is no longer used in any official capacity.

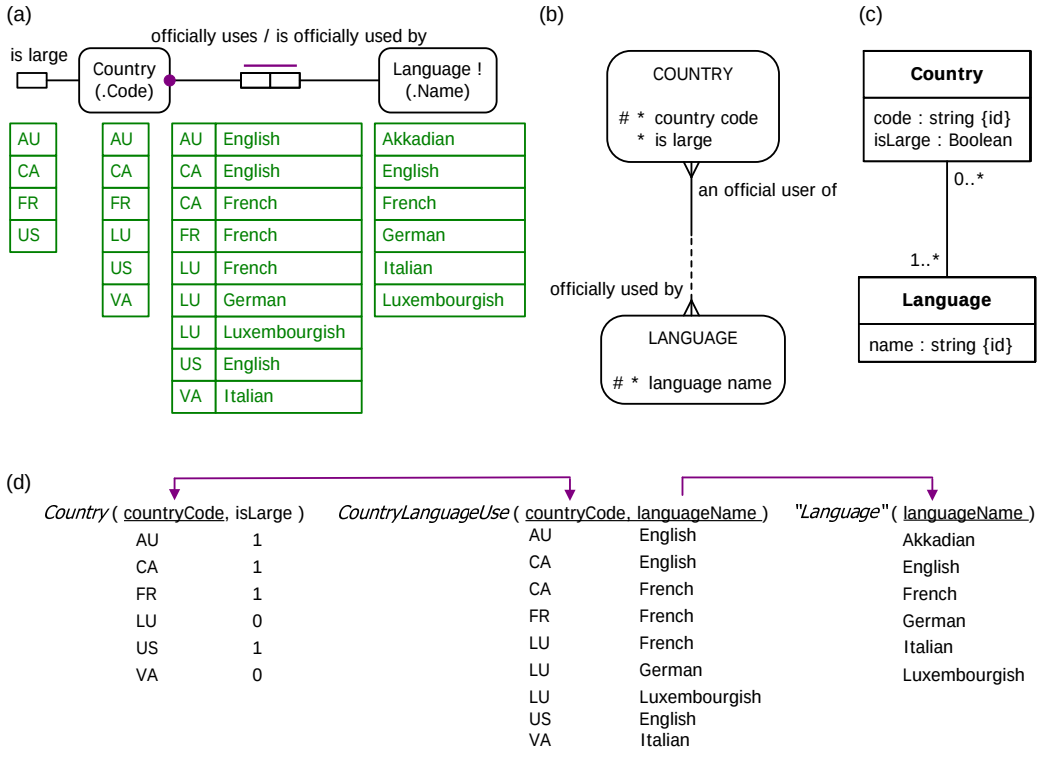

Figure 1 Sample data model in (a) ORM, (b) Barker ER, (c) UML, and (d) relational database notation.

The inverse predicate reading "is officially used by" is now included for the fact type Country officially uses Language (in ORM, forward and inverse readings are separated by a slash). For the fact type Country officially uses Language, the role connection to Country has a large dot, indicating that for each state of the database, that role must be played by each instance in the population of Country. This *mandatory role constraint* may be verbalized as "Each Country officially uses some Language". In this fact type, the role connected to Language has no such dot, so is *optional* for Language. This optionality may be verbalized as "It is possible that some Language is officially used by no Country". The closed world assumption is assumed for the unary fact type Country is large, so sample countries that are not included in the population of this fact type are assumed to be not large (e.g. Luxembourg and the Vatican City State).

Figure 1(b) depicts the same data model in the Barker notation [2] for Entity Relationship (ER) modeling, but without the sample data. Figure 1(c) depicts the same data model as a class diagram in the Unified Modeling Language (UML) [7], again without the sample data. Finally, Figure 1(d) depicts a populated relational database model for the same example. There are three relational tables, with their primary keys underlined. The arrowed lines between attributes depict subset constraints: the arrowed lines from CountryLanguageUse.countryCode to Country.countryCode and from CountryLanguageUse.languageName to Language.languageName denote foreign key references; the arrow from Country.countryCode to CountryLanguageUse.countryCode depicts the subset constraint that each country officially uses some language. The attribute Country.isLarge uses the bit data type, so here 1 denotes True and 0 denotes False.

The previous article [3] discussed how to code most of this Country model in LogiQL using the free cloud-based REPL (Read-Eval-Print-Loop) "playground" provided at the website mentioned earlier. We now discuss how to write LogiQL code to declare the inverse predicate "is officially used by" and the mandatory role constraint that each Country officially uses some language. If you wish to execute LogiQL code yourself on the LogiQL playground, please note that currently the website supports only Chrome and Firefox as web browsers. It is anticipated that other web browsers will be supported in the future.

As discussed in [3], the reference schemes and fact types for Country and Language may be declared in LogiQL as follows. Predicates are written in prefix notation with their arguments in parentheses. The right arrow symbol "->" stands for the material implication operator "→" of logic, and is read as "implies". The colon ":" in hasCountryCode(c:cc) and hasLanguageName(l:ln) distinguishes hasCountryCode and hasLanguageName as refmode predicates, so these predicates are injective (mandatory, 1:1). Recall that LogiQL is case-sensitive, and each formula must end with a period.

```
Country(c), hasCountryCode(c:cc) -> string(cc).
Language(l), hasLanguageName(l:ln) -> string(ln).
isLarge(c) -> Country(c).
officiallyUses(c, l) -> Country(c), Language(l).
```

The ORM model in Figure 1(a) provides both forward and inverse predicate readings for the fact type Country officially uses Language. The forward reading is declared in the above code using the officiallyUses predicate. The *inverse predicate* may be declared using the following derivation rule, where the left-arrow symbol "<-" denotes the inverse material implication operator "←" of logic, read as "if".

```
isOfficiallyUsedBy(l, c) <- officiallyUses(c, l).
```

LogiQL formulae assume that variables that occur on both sides of an arrow are implicitly universally quantified, so the above formula is equivalent to the following formula in predicate logic, where "∀" denotes the universal quantifier, read as "for each".

$\forall l,c$ (l isOfficiallyUsedBy c $\leftarrow$ c officiallyUses l)

The *mandatory role constraint* that each country officially uses some language may be coded in LogiQL as follows. Here the *underscore* "_" denotes the *anonymous variable*, read as "something".

```
Country(c) -> officiallyUses(c, _).
```

The individual variable *c* is implicitly universal quantified, so this formula may be read as follows: Given any individual thing *c*, if *c* is a country then *c* officially uses something. Because of the typing constraint declared earlier on officiallyUses we know that the "something" mentioned here is some country.

The above LogiQL formula is equivalent to the following formula in predicate logic, where "∃" denotes the existential quantifier, read as "there exists some". LogiQL uses the anonymous variable in this context to avoid typing the symbol for the existential quantifier (which is not available on standard keyboards).

$$\forall c \, (\text{Country } c \rightarrow \exists l \;\; c \text{ officiallyUses } l \,)$$

In Figure 1(a), Language is declared independent (shown by appending an exclamation mark). In LogiQL an object type with no mandatory role constraint is implicitly independent, so no code is needed to declare that Language is independent for this example. In both ORM and LogiQL, all predicates are set-based, and have a spanning uniqueness constraint unless constrained otherwise. Hence there is no need to add any code for the many-to-many uniqueness constraint on the Country officially uses Language fact type. The full LogiQL code for the schema in Figure 1 is set out below.

```
Country(c), hasCountryCode(c:cc) -> string(cc).
Language(l), hasLanguageName(l:ln) -> string(ln).
isLarge(c) -> Country(c).
officiallyUses(c, l) -> Country(c), Language(l).
isOfficiallyUsedBy(l, c) <- officiallyUses(c, l).
Country(c) -> officiallyUses(c, _).
```

To enter the schema in the free, cloud-based REPL tool use a supported browser such as Chrome or Firefox to access the website (https://developer.logicblox.com/playground/), then click the "Open in new window" link to show a full screen for entering the code. Schema code is entered in one or more *blocks* of one or more lines of code, using the *addblock* command to enter each block. After the "/>" prompt, type the letter "a". A drop-down list of available commands starting with the letter "a" now appears. Click the addblock option to have the addblock command(followed by a space) added to the code window. Typing a single quote after the addblock command causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your block of code (see Figure 2).

Figure 2 Invoking the addblock command in the REPL tool.

Now copy the six lines of schema code provided above in this article to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. You are now notified that the block was successfully added, and a new prompt awaits your next command (see Figure 3). By default, the REPL tool also appends an automatically generated identifier for the code block Alternatively, you can enter each line of code directly, using a separate addblock command for each line.

```
      1  Welcome to the LogicBlox playground!
      2
9-33 />  addblock 'Country(c), hasCountryCode(c:cc) -> string(cc).
     ..  Language(l), hasLanguageName(l:ln) -> string(ln).
     ..  isLarge(c) -> Country(c).
     ..  officiallyUses(c, l) -> Country(c), Language(l).
     ..  isOfficiallyUsedBy(l, c)  <-  officiallyUses(c, l).
     ..  Country(c)  ->  officiallyUses(c, _).
     ..  '
     =>
         Successfully added block 'block_1Z1C1G7M'
9-33 />
```

Figure 3 Adding a block of schema code.

The data in Figure 1(a) may be entered in LogiQL using the following delta rules. A delta rule of the form *+fact* inserts that fact. For example, the delta rule "+isLarge("AU")." inserts the fact that Australia is a large country. Recall that *plain, double quotes* (i.e. ",") are needed here, not single quotes or smart double quotes. Hence it's best to use a basic text editor such as WordPad or NotePad to enter code that will later be copied into a LogiQL tool.

```
+isLarge("AU"), +isLarge("CA"), +isLarge("FR"), +isLarge("US").
+officiallyUses("AU", "English"), +officiallyUses("CA", "English").
+officiallyUses("CA", "French"), +officiallyUses("FR", "French").
+officiallyUses("LU", "French"),  +officiallyUses("LU", "German").
+officiallyUses("LU", "Luxembourgish").
+officiallyUses("US", "English"), +officiallyUses("VA", "Italian").
+Language("Akkadian").
```

Note that there is no need to add delta rules to insert countries and languages (other than Akkadian) because those facts are implied by the typing and mandatory role constraints on the officiallyUses predicate. For example, the assertion +officiallyUses("AU", "English") implies the assertions +Country("AU") and +Language("English"). However, there is no harm in adding those facts separately if you wish.

Delta rules to add or modify data are entered using the exec (for 'execute') command rather than the addblock command. To invoke the exec command in the REPL tool, type "e" and then select exec from the drop-down list. A space character is automatically appended. Typing a single quote after the exec command and space causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your delta rules.

Now copy the seven lines of data code provided above to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. A new prompt awaits your next command. Both schema and data have now been entered (see Figure 4). Alternatively, you can enter each line of data yourself directly, using a separate exec command for each line.

```
9-33 />  exec '+isLarge("AU"), +isLarge("CA"), +isLarge("FR"), +isLarge("US").
     ..  +officiallyUses("AU", "English"), +officiallyUses("CA", "English").
     ..  +officiallyUses("CA", "French"), +officiallyUses("FR", "French").
     ..  +officiallyUses("LU", "French"),  +officiallyUses("LU", "German").
     ..  +officiallyUses("LU", "Luxembourgish").
     ..  +officiallyUses("US", "English"), +officiallyUses("VA", "Italian").
     ..  +Language("Akkadian").
     ..  '
9-33 />
```

Figure 4 Adding the data below the program code.

Now that the data model (schema plus data) is stored, you can use the *print* command to inspect the contents of any predicate. For example, to list all the recorded languages, type "p" then select print from the drop-down list, and then type a space followed by "L", then select Language from the drop-down list and press Enter. Alternatively, you can type all of "print Language" yourself and then press Enter. Figure 5 shows the relevant result. By default, the REPL tool also prepends a column listing automatically generated, internal identifiers for the returned entities.

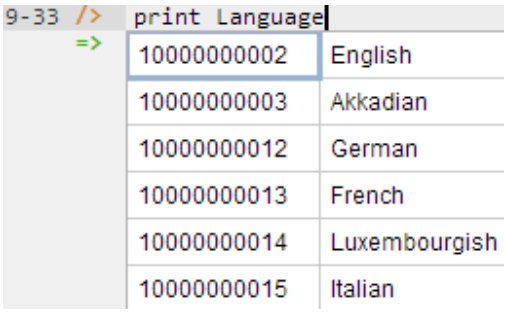

Figure 5 Using the print command to list the extension of a predicate.

As discussed in the previous article [3], to perform a general query, you need to specify a derivation rule to compute the facts requested by the query. If you simply want to issue a query to derive some facts without needing to reference them later, there is no need to name the derived predicate, so we just use an anonymous predicate instead to capture the query result. For example, using a comma "," for the logical "and" operator and an exclamation mark "!" for the logical not operator, the following query may be used to derive those languages that are not officially used by any country.

```
_(l) <- Language(l), !officiallyUses( _, l).
```

Here the rule's head _(*l*) uses an anonymous predicate to capture the result derived from the rule's body. Since the variable *l* is a head variable, it is implicitly universally quantified. Given the type declaration for the officiallyUses predicate, this query lists each language where it is not the case that some country officially uses that language. So the above query is understood as shorthand for the following logic formula, where the dummy isReturned predicate is satisfied by what is returned by the query, thus standing for an anonymous predicate.

$$\forall l \, [\, l \text{ isReturned} \leftarrow (\text{ Language } l \;\&\; \sim\exists c(c \text{ officiallyUses } l\,)\,]$$

In LogiQL, queries are executed by appending their code in single quotes to the *query* command. To do this in the REPL tool, type "q", choose "query" from the drop-down list, type a single quote, then copy and paste the above LogiQL query code between the quotes and press Enter. The relevant query result is now displayed as shown in Figure 6. By default, the REPL tool also prepends a column listing automatically generated, internal identifiers for the returned entities.

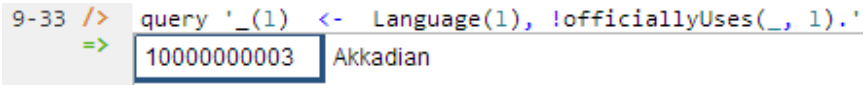

Figure 6 Using the query command to execute a sample query.

Internal Uniqueness Constraints on Binary Fact Types

As indicated earlier, predicates in LogiQL are by default assumed to have a spanning uniqueness constraint, so binary predicates are many-to-many unless a stronger uniqueness pattern is specified for them. The ORM data model in Figure 7(a) includes three binary fact types with non-spanning uniqueness constraints. The simple uniqueness constraints on each role of the fact type Country has CountryName indicate that this is a *one-to-one* relationship, so no duplicates are allowed in the columns of its sample fact table. In other words, each country has at most one name, and each country name applies to at most one country.

The uniqueness constraint on the scientist role of the fact type Scientist was born in Country indicates that each scientist was born in at most one country, while the lack of a uniqueness constraint on the other role indicates that some country may be the birth country of more than one scientist. So the fact type Scientist was born in Country is *many-to-one*. Similarly the fact type Country has Population is also many-to-one. In ORM, arrow-tips on predicate readings reverse the usual right-to-left or top-to-bottom reading direction, and role names may optionally be provided and displayed in square brackets besides beside their role box. The solid dots on the three fact types displayed in Figure 7 depict mandatory role constraints, so each country has a name and population, and each scientist was born in some country.

Figure 7(b) and Figure 7(c) model the same example in Barker ER and UML respectively, but have no graphical notation to constrain each country name to refer to at most one country. The relational database model shown in Figure 7(d) captures both uniqueness constraints for Country, using a double-underline and single-underline to mark the Country table's primary key and secondary key respectively. The *n*:1 and mandatory-to-optional nature of the birth country relationship is depicted in Barker ER using a crows-foot for "many", a solid line for mandatory and a dashed line for optional, as shown. In UML, this is captured by the multiplicity constraints indicated ("1" for exactly one, and "*" for zero or more).

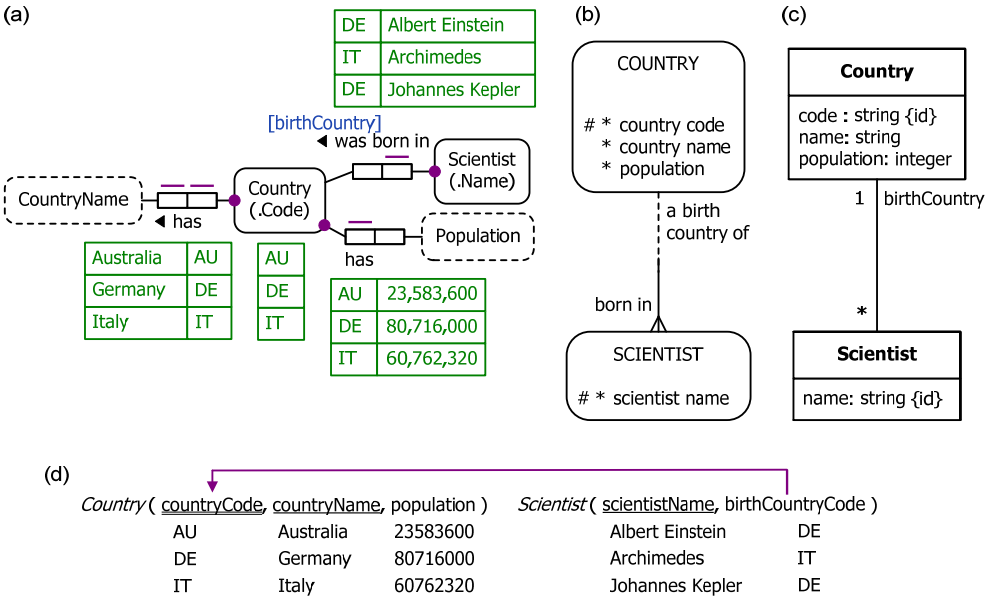

Figure 7 Another sample data model in (a) ORM, (b) Barker ER, (c) UML, and (d) relational database notation.

In LogiQL, the reference schemes for Country and Scientist may be declared in the usual way as follows:

```
Country(c), hasCountryCode(c:cc) -> string(cc).
Scientist(s), hasScientistName(s:sn) -> string(sn).
```

The three fact types discussed earlier have a non-spanning uniqueness constraint so they are *functional fact types*. For example, country name and population are each functionally dependent on country, and birth country is a function of scientist. In LogiQL, functional fact types are best declared using a *square bracket notation* where the arguments that functionally determine the final argument are placed in square brackets , followed by the equals operator "=" and the final argument. For readability, the predicate name is usually best formed by appending a preposition such as "Of" to a name for the relevant role. For example, the three fact types under discussion may be declared as follows using "int" for an integer datatype used to store population numbers.

```
birthCountryOf[s] = c  ->  Scientist(s), Country(c).
countryNameOf[c] = cn  ->  Country(c), string(cn).
populationOf[c] = n  ->  Country(c), int(n).
```

These fact types could alternatively be declared using the parentheses notation used for non-functional fact types. However, the square-bracket notation is more efficient, since it declares not only the types of the arguments but also the functional nature of the predicate. If we use the parentheses notation we would have to declare the functional nature of the predicate in a separate constraint. For example, the formula "birthCountryOf[s] = c -> Scientist(s), Country(c)." in square bracket notation is equivalent to the combination of the following two formulae in parentheses notation, where the second formula constrains each scientist to be born in at most one country:

```
wasBornIn(s, c)  ->  Scientist(s), Country(c).
wasBornIn(s, c1), wasBornIn(s, c2)  ->  c1 = c2.
```

If a predicate is one-to-one, the square bracket notation captures only one of its uniqueness constraints, so the other uniqueness constraint must be declared separately. For example, to ensure that each country name applies to only one country, the following constraint must be added:

```
countryNameOf[c1] = cn, countryNameOf[c2] = cn  ->  c1 = c2.
```

The three mandatory role constraints on the fact types in Figure 7(a) are specified using anonymous variables in the usual way, so the full schema for Figure 7(a) may now be declared as shown below.

```
Country(c), hasCountryCode(c:cc) -> string(cc).
Scientist(s), hasScientistName(s:sn) -> string(sn).
birthCountryOf[s] = c  ->  Scientist(s), Country(c).
countryNameOf[c] = cn  ->  Country(c), string(cn).
populationOf[c] = n  ->  Country(c), int(n).
countryNameOf[c1] = cn, countryNameOf[c2] = cn  ->  c1 = c2.
Country(c)  ->  countryNameOf[c] = _.
Country(c)  ->  populationOf[c] = _.
Scientist(s)  ->  birthCountryOf[s] = _.
```

To implement this in the REPL tool, use the add block command and copy the above code in the usual way, as shown in Figure 8 .

```
addblock 'Country(c), hasCountryCode(c:cc) -> string(cc).
Scientist(s), hasScientistName(s:sn) -> string(sn).
birthCountryOf[s] = c  ->  Scientist(s), Country(c).
countryNameOf[c] = cn  ->  Country(c), string(cn).
populationOf[c] = n  ->  Country(c), int(n).
countryNameOf[c1] = cn, countryNameOf[c2] = cn  ->  c1 = c2.
Country(c)  ->  countryNameOf[c] = _.
Country(c)  ->  populationOf[c] = _.
Scientist(s)  ->  birthCountryOf[s] = _.
'

Successfully added block 'block_1Z1C1HCY'
```

Figure 8 Adding the schema for Figure 7 in the REPL tool.

The sample data in Figure 7 may be added using the following delta rules:

```
+birthCountryOf["Albert Einstein"] = "DE".
+birthCountryOf["Archimedes"] = "IT".
+birthCountryOf["Johannes Kepler"] = "DE".
+countryNameOf["AU"] = "Australia".
+countryNameOf["DE"] = "Germany".
+countryNameOf["IT"] = "Italy".
+populationOf["AU"] = 23583600.
+populationOf["DE"] = 80716000.
+populationOf["IT"] = 60762320.
```

To implement this in the REPL tool, use the exec command and copy the above code in the usual way, as shown in Figure 9.

```
exec '+birthCountryOf["Albert Einstein"] = "DE".
+birthCountryOf["Archimedes"] = "IT".
+birthCountryOf["Johannes Kepler"] = "DE".
+countryNameOf["AU"] = "Australia".
+countryNameOf["DE"] = "Germany".
+countryNameOf["IT"] = "Italy".
+populationOf["AU"] = 23583600.
+populationOf["DE"] = 80716000.
+populationOf["IT"] = 60762320.
```

Figure 9 Adding the data for Figure 7 in the REPL tool.

You can now print and query the model in the usual way. For example, the following query lists each scientist and the name and population of his/her birthcountry where the country has a population over 70000000.

```
_(s, cn, p) <- birthCountryOf[s] = c, countryNameOf[c] = cn, populationOf[c] = p,
populationOf[c] > 70000000.
```

Running this query in REPL tool yields the result shown in Figure 10. The first column may be ignored as it merely displays internal identifiers for the returned entities (only the scientists are modeled in LogiQL as entities, since country names are coded as strings and populations as integers).

```
query '_(s, cn, p) <- birthCountryOf[s] = c, countryNameOf[c] = cn, populationOf[c] = p, populationOf[c] > 70000000.'
```

10000000000	Johannes K...	Germany	80716000
10000000006	Albert Einstein	Germany	80716000

Figure 10 Using the query command to execute a sample query on the second data model.

Conclusion

The current article discussed some further, basic aspects of LogiQL, focusing on how to declare inverse predicates, simple mandatory role constraints, and internal uniqueness constraints on binary fact types. Future articles in this series will examine how LogiQL can be used to specify business constraints and rules of a more advanced nature.

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley, Reading, MA.
2. Barker, R. 1990, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.
3. Halpin, T. 2014, 'Logical Data Modeling: Part 1', *Business Rules Journal*, Vol. 15, No. 5 (May, 2014), URL: http://www.BRCommunity.com/a2014/b760.html.

4. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, *2nd edition*, Morgan Kaufmann, San Francisco.
5. Halpin, T. & Wijbenga, J. 2010, 'FORML 2', *Enterprise, Business-Process and Information Systems Modeling*, eds. I. Bider et al., LNBIP 50, Springer-Verlag, Berlin Heidelberg, pp. 247–260.
6. Huang, S., Green, T. & Loo, B. 2011, 'Datalog and emerging applications: an interactive tutorial', *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, ACM, New York. http://dl.acm.org/citation.cfm?id=1989456.
7. Object Management Group 2013, *OMG Unified Modeling Language (OMG UML)*, version 2.5 RTF Beta 2. Available online at: http://w ww.omg.org/spec/UML/2.5/Beta2/PDF/.