Logical Data Modeling: Part 13

Terry Halpin
INTI International University

This is the thirteenth article in a series on logic-based approaches to data modeling. The first article [5] briefly overviewed deductive databases, showing how simple data models with asserted and derived facts may be declared and queried in LogiQL [2, 20, 22], a leading edge deductive database language based on extended datalog [1]. The second article [6] discussed how to declare inverse predicates, simple mandatory role constraints and internal uniqueness constraints on binary fact types. The third article [7] explained how to declare *n*-ary predicates and apply simple mandatory role constraints and internal uniqueness constraints to them. The fourth article [8] discussed external uniqueness constraints. The fifth article [9] covered derivation rules in some more detail, and inclusive-or constraints. The sixth article [10] discussed simple set-comparison constraints (i.e. subset, exclusion, and equality constraints), and exclusive-or constraints. The seventh article [11] covered subset constraints between compound role sequences, including cases involving join paths. The eighth article [12] discussed exclusion and equality constraints between compound role sequences. The ninth article [13] explained how to declare basic subtyping. The tenth article [14] discussed how to declare relationships to be irreflexive (using a ring constraint) and/or symmetric (using a ring constraint or a derivation rule). The eleventh article [15] showed how to constrain a relationship to be asymmetric and/or intransitive. The twelfth article [16] discussed recursive derivation rules, and how to constrain ring relationships to be acyclic and/or strongly intransitive. The current article shows how to declare internal and external frequency constraints. The LogiQL code examples are implemented using the free cloud-based REPL (Read-Eval-Print-Loop) tool for LogiQL accessible at https://developer.logicblox.com/playground/.

A Parenthood Chart from Egyptian Mythology

Figure 1 pictures the nine Egyptian gods collectively known as the Great Ennead of Heliopolis. Each god is identified by its name, and is a parent of zero or more gods in the same Ennead. A downwards arrow depicts the parent to child relationship. This example was discussed in the previous two articles [15, 16] to illustrate asymmetric, intransitive, acyclic, and strongly intransitive ring constraints.

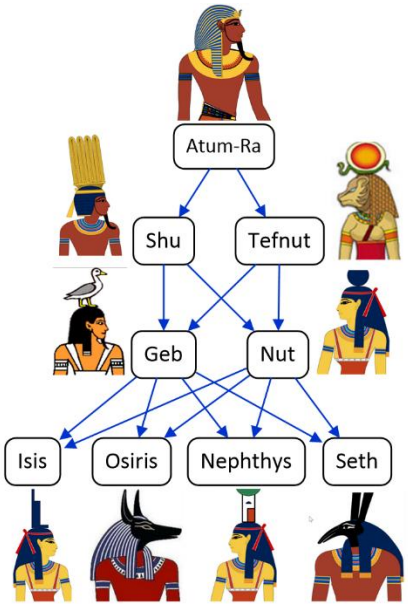
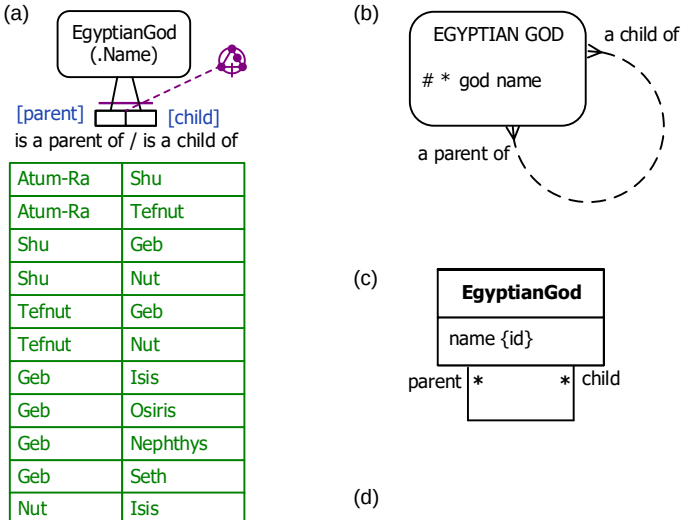


Figure 1 Parenthood relationships of the Great Ennead of Heliopolis.

Figure 2(a) provides a simplified data model for this example using a populated Object-Role Modeling (ORM) [17, 18, 19] diagram, with the parenthood relationship constrained to be acyclic (which implies asymmetry) and strongly intransitive (which implies intransitivity). The example is schematized in Figure 2(b) as an Entity Relationship diagram in Barker notation (Barker ER) [3], in Figure 2(c) as a class diagram in the Unified Modeling Language (UML) [23], and in Figure 2(d) as a relational database (RDB) schema. For detailed discussion of these schemas see the previous two articles [15, 16]. The UML, Barker ER, and RDB schemas have no graphical way to depict ORM ring constraints.



Atum-Ra	Shu
Atum-Ra	Tefnut
Shu	Geb
Shu	Nut
Tefnut	Geb
Tefnut	Nut
Geb	Isis
Geb	Osiris
Geb	Nephthys
Geb	Seth
Nut	Isis
Nut	Osiris
Nut	Nephthys
Nut	Seth

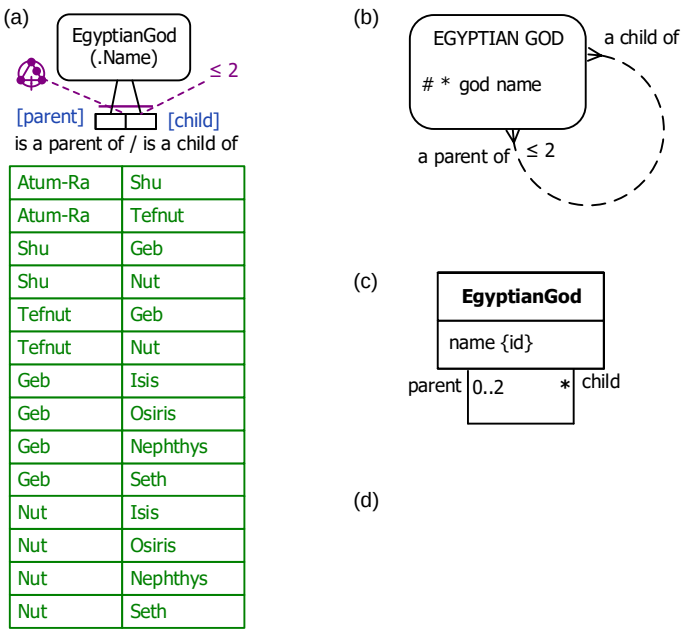
Figure 2 Simplified data model for Figure 1 in (a) ORM, (b) Barker ER, (c) UML, and (d) RDB notation.

A complete data model for this example requires an internal frequency constraint. The next section shows how to declare internal frequency constraints in ORM and LogiQL, and the section after that discusses how to declare external frequency constraints.

Internal Frequency Constraints

Looking back at Figure 1 it's easy to see that Atum-Ra has no parent, Shu and Tefnut each have exactly one parent, and the other gods each have two parents. In other words, each Egyptian god in the Ennead has *at most two* parents. Since only whole instances are allowed, here "at most 2" means "0, 1, or 2". The 0 case is captured by declaring the child role in the parenthood fact type to be optional for an Egyptian god, as indicated by the lack of a mandatory role constraint on the child role. Atum-Ra has no parents (i.e. 0 parents) so does not appear in the fact column for the child role in Figure 2(a).

For each population of the parenthood fact type, those Egyptian gods who appear in the fact column for the child role must occur there once or twice (i.e. 1 or 2 times). This restriction is specified in ORM using a *simple internal frequency constraint*. In ORM, a simple internal frequency constraint applies to a single fact role, and restricts the frequency or number of times an instance in any specific population of that role may occur there. In our parenthood case, the minimum frequency is 1 and the maximum frequency is 2, as illustrated by the fact population in Figure 2(a). A simple frequency constraint applies only to instances that do play the role, so the lowest possible minimum frequency is 1 (not zero). By treating mandatory role constraints and frequency constraints as separate and orthogonal issues, ORM enables more efficent implementations of those constraints as well as declaration of constraint cases that couldn't be captured if these constraints were combined into a single constraint category such as multiplicity constraints in UML.



Atum-Ra	Shu
Atum-Ra	Tefnut
Shu	Geb
Shu	Nut
Tefnut	Geb
Tefnut	Nut
Geb	Isis
Geb	Osiris
Geb	Nephthys
Geb	Seth
Nut	Isis
Nut	Osiris
Nut	Nephthys
Nut	Seth

Figure 3 Adding a constraint that an Egyptian god has at most two parents in (a) ORM, (b) Barker ER, and (c) UML.

Using the NORMA tool for ORM, entering minimum and maximum frequencies of 1 and 2 respectively for the child role causes the internal frequency constraint to be displayed as the numeric restriction "≤ 2" (for "at most 2") attached by a dotted line to the role being constrained, as illustrated in Figure 3(a). The NORMA tool for ORM verbalizes this constraint as "**Each** EqyptianGod is a child of **at most two instances of** EqyptianGod". This is equivalent to the longer formulation "**Each** God_1 **in the population of** "God_1 is a parent of God_2" **occurs there at least** 1 **and at most** 2 **times**.". Such longer verbalizations are used when the minimum frequency is above 1.

Figure 3(b) shows how to depict the ORM frequency constraint in the Barker ER notation, where it is known as a cardinality constraint. Here the "≤ 2" restriction appears at the parent role end of the parenthood relationship rather than the child role. This differs from ORM's convention where attaching a constraint to a role constrains the population of that same role. The optionality of the child role is shown by using a dashed line for that end of the relationship.

The UML class diagram in Figure 3(c) depicts the optionality of the child role as well as the ORM frequency constraint using a multiplicity constraint. Unlike both ORM and Barker ER, UML has no constraint notation to simply declare whether a role is optional or mandatory, so uses a multiplicity constraint of "0..2" (i.e. 0, 1, or 2) to indicate that each Egyptian god is a child of at most 2 parents. Like Barker ER, UML displays the constraint on the parent role rather than the child role.

Figure 3(d) depicts the example in relational database notation. Unlike ORM, Barker ER, and UML, this RDB notation has no graphic way to depict the frequency constraint.

As discussed in the previous two articles [15, 16], the ORM schema in Figure 3(a) minus the frequency constraint may be coded in LogiQL as shown below. The acyclic and strong intransitivity constraints on the parenthood predicate are enforced by first recursively deriving the ancestorhood predicate then declaring it to be irreflexive and ensuring that no Egyptian god is a parent of a descendant of one of its children.

```
EgyptianGod(g), hasGodName(g:gn) -> string(gn).
isaParentOf(g1, g2) -> EgyptianGod(g1), EgyptianGod(g2).
isaChildOf(g1, g2) <- isaParentOf(g2, g1).
isaParentOf(g1, g2) -> !isaParentOf(g2, g1). // asymmetric
isaParentOf(g1, g2), isaParentOf(g2, g3) -> !isaParentOf(g1, g3). // intransitive
isanAncestorOf(g1, g2) <-
    isaParentOf(g1, g2) ;
    isaParentOf(g1, g3), isanAncestorOf(g3, g2).
```

```
!isanAncestorOf(g, g).    // No Egyptian God is an ancestor of itself
// No EgyptianGod is a parent of a descendant of one of its children
isaParentOf(g1, g2), isanAncestorOf(g2, g3)  ->  !isaParentOf(g1, g3).
```

To implement the "≤ 2" frequency constraint, we need to count the number of times each Egyptian god plays the child role and ensure that this number is 0, 1 or 2. In LogiQL, this is done using the *count* function, which is one of several aggregate functions (e.g. count, total, min, max) that may be applied to a collection of facts to return a single number. For counting, LogiQL uses a special syntax "agg<<n = count()>>*condition*", meaning "n is the count of all the facts satisfying *condition*". For our example, we can derive the number of parents of each Egyptian god using the following LogiQL derivation rule. Recall that an underscore "_" denotes the anonymous variable (read as "something"), so the following rule says that the number of parents of an Egyptian god *g* equals the count of all the facts where something is the parent of *g*.

```
// The number of parents of g is the count of all the facts where something is a parent of g
nrParentsOf[g] = n  <-
      agg<<n = count()>> isaParentOf(_, g).
```

If a god (e.g. Atum-Ra) has no parents, you might expect this count function to return 0. However, in this case the LogiQL the count function returns no value at all if it doesn't find any facts satisfying the condition. To ensure that 0 is returned for such empty set cases, we can simply assign 0 as the default value for the count function. This is coded as follows using the lang:defaultValue metapredicate followed by square brackets that enclose the grave accent character "`" followed by function name.

```
// The default value of the nrParentsOf function is 0.
lang:defaultValue[`nrParentsOf] = 0.
```

The frequency constraint may now be coded simply as follows, using two characters "<=" for "≤".

```
// Each Egyptian god has at most 2 parents
nrParentsOf[_] = n  ->  n <= 2.
```

For your convenience in entering code to the REPL tool, the full schema code is repeated below.

```
EgyptianGod(g), hasGodName(g:gn)  ->  string(gn).
isaParentOf(g1, g2)  ->  EgyptianGod(g1), EgyptianGod(g2).
isaChildOf(g1, g2)  <-  isaParentOf(g2, g1).
isaParentOf(g1, g2) -> !isaParentOf(g2, g1).  // asymmetric
isaParentOf(g1, g2), isaParentOf(g2, g3) ->  !isaParentOf(g1, g3).  // intransitive
isanAncestorOf(g1, g2)  <-
      isaParentOf(g1, g2) ;
      isaParentOf(g1, g3), isanAncestorOf(g3, g2).
!isanAncestorOf(g, g).    // No Egyptian God is an ancestor of itself
// No EgyptianGod is a parent of a descendant of one of its children
isaParentOf(g1, g2), isanAncestorOf(g2, g3)  ->  !isaParentOf(g1, g3).
// The number of parents of g is the count of all the facts where something is a parent of g
nrParentsOf[g] = n  <-
      agg<<n = count()>> isaParentOf(_, g).
// The default value of the nrParentsOf function is 0.
lang:defaultValue[`nrParentsOf] = 0.
// Each Egyptian god has at most 2 parents
nrParentsOf[_] = n  ->  n <= 2.
```

To enter the schema in the free, cloud-based REPL tool, use a supported browser to access the website https://repl.logicblox.com. The latest version of the REPL tool now appears to require you to *explicitly create a workspace before adding code blocks* to it. To do this, you may simply enter the following code:

```
create --unique
```

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", and click the addblock option that then appears. This causes the addblock command (followed by a space) to be 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.

Now copy the full schema 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. You are now notified that the block was successfully added, and a new prompt awaits your next command (see Figure 4). 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
/>   create --unique
=>   Successfully created workspace
/>   addblock 'EgyptianGod(g), hasGodName(g:gn)  ->  string(gn).
..   isaParentOf(g1, g2)  ->  EgyptianGod(g1), EgyptianGod(g2).
..   isaChildOf(g1, g2)  <-  isaParentOf(g2, g1).
..   isaParentOf(g1, g2) -> !isaParentOf(g2, g1).  // asymmetric
..   isaParentOf(g1, g2), isaParentOf(g2, g3) ->  !isaParentOf(g1, g3).  // intransitive
..   isanAncestorOf(g1, g2)  <-
..   isaParentOf(g1, g2) ;
..   isaParentOf(g1, g3), isanAncestorOf(g3, g2).
..   !isanAncestorOf(g, g).  // No Egyptian God is an ancestor of itself
..   // No EgyptianGod is a parent of a descendant of one of its children
..   isaParentOf(g1, g2), isanAncestorOf(g2, g3)  ->  !isaParentOf(g1, g3).
..   // The number of parents of g is the count of all the facts where something is a parent of g
..   nrParentsOf[g] = n  <-
..       agg<<n = count()>> isaParentOf(_, g).
..   // The default value of the nrParentsOf function is 0.
..   lang:defaultValue[`nrParentsOf] = 0.
..       // Each Egyptian god has at most 2 parents
..   nrParentsOf[_] = n  ->  n <= 2.
..   '
=>
     Successfully added block 'block_1Z331J1X'
/>
```

Figure 4 Creating a workspace and then adding a block of schema code.

The data in Figure 3(a) may be entered in LogiQL using the following delta rules. A delta rule of the form *+fact* inserts that fact. 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.

```
+EgyptianGod(g1), +hasGodName(g1:"Atum-Ra"), +EgyptianGod(g2), +hasGodName(g2:"Shu"),
+EgyptianGod(g3), +hasGodName(g3:"Tefnut"), +EgyptianGod(g4), +hasGodName(g4:"Geb"),
+EgyptianGod(g5), +hasGodName(g5:"Nut"), +EgyptianGod(g6), +hasGodName(g6:"Isis"),
+EgyptianGod(g7), +hasGodName(g7:"Osiris"), +EgyptianGod(g8), +hasGodName(g8:"Nephthys"),
+EgyptianGod(g9), +hasGodName(g9:"Seth"),
+isaParentOf(g1, g2), +isaParentOf(g1, g3),
+isaParentOf(g2, g4), +isaParentOf(g2, g5),
+isaParentOf(g3, g4), +isaParentOf(g3, g5),
+isaParentOf(g4, g6), +isaParentOf(g4, g7), +isaParentOf(g4, g8), +isaParentOf(g4, g9),
+isaParentOf(g5, g6), +isaParentOf(g5, g7), +isaParentOf(g5, g8), +isaParentOf(g5, g9).
```

Delta rules to add or modify data are entered using the exec (for 'execute') 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 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 (see Figure 5).

```
/> exec '+EgyptianGod(g1), +hasGodName(g1:"Atum-Ra"), +EgyptianGod(g2), +hasGodName(g2:"Shu"),
..  +EgyptianGod(g3), +hasGodName(g3:"Tefnut"), +EgyptianGod(g4), +hasGodName(g4:"Geb"),
..  +EgyptianGod(g5), +hasGodName(g5:"Nut"), +EgyptianGod(g6), +hasGodName(g6:"Isis"),
..  +EgyptianGod(g7), +hasGodName(g7:"Osiris"), +EgyptianGod(g8), +hasGodName(g8:"Nephthys"),
..  +EgyptianGod(g9), +hasGodName(g9:"Seth"),
..  +isaParentOf(g1, g2), +isaParentOf(g1, g3),
..  +isaParentOf(g2, g4), +isaParentOf(g2, g5),
..  +isaParentOf(g3, g4), +isaParentOf(g3, g5),
..  +isaParentOf(g4, g6), +isaParentOf(g4, g7), +isaParentOf(g4, g8), +isaParentOf(g4, g9),
..  +isaParentOf(g5, g6), +isaParentOf(g5, g7), +isaParentOf(g5, g8), +isaParentOf(g5, g9).
..  '
/>
```

Figure 5 Adding the data.

Now that the data model is stored, you can use the *print* command to inspect the contents of any predicate. For example, to list the names of the Ennead gods, type "p" then select print from the drop-down list, then type a space followed by "E", then select EgyptianGod from the drop-down list and press Enter. The print command for the nrParentsOf predicate fails to include the 0 result for Atum-Ra, but this value can be returned by using a query, as discussed below.

To perform a query, you specify a derivation rule to compute the facts requested by the query. For example, the following query may be used to list the god name and number of parents of each Egyptian god. The rule's head uses an anonymous predicate to capture the result derived from the rule's body. The head variables *gn* and *n* are implicitly universally quantified. The variable *g* introduced in the body is implicitly existentially quantified.

```
_(gn, n) <- hasGodName(g:gn), nrParentsOf[g] = n.
```

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.

```
/> query '_(gn, n) <- hasGodName(g:gn), nrParentsOf[g] = n.'
=>
```

Atum-Ra	0
Geb	2
Isis	2
Nephthys	2
Nut	2
Osiris	2
Seth	2
Shu	1
Tefnut	1

```
/>
```

Figure 6 A query to list the god name and the number of parents of each of the Ennead gods.

The numeric expression in an ORM frequency constraint may take various forms, such as *n* (exactly *n*), ≤ *n* (at most *n*), ≥ *n* (at least *n*), *n..m* (at least *n* and at most *m*). These may be coded in LogiQL in a similar way to that discussed above.

In addition to simple frequency constraints, *compound frequency constraints* may be specified. Each compound uniqueness constraint applies to a sequence of two or more roles. For example, in Figure 7 the frequency constraint of 2 is attached to the join of the two roles hosted by Department and Year. This constraint ensures that each department and year that has staff numbers recorded in the fact type Department in Year had staff of Gender in Quantity appears there twice (once for each gender). The NORMA tool verbalizes this constraint thus: **Each** Department, Year **combination in the population of** "Department in Year had staff of Gender in Quantity" **occurs there exactly** 2 **times**. The coding of this example in LogiQL is left as an optional exercise.

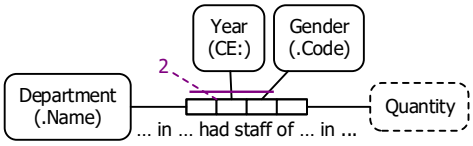


Figure 7 An ORM schema with a compound frequency constraint.

External Frequency Constraints

As you may have realized, a frequency constraint in ORM is a generalization of a uniqueness constraint (which is equivalent to a frequency constraint of 1). Just as ORM supports external uniqueness constraints, it also supports external frequency constraints. An *external frequency constraint* applies to a sequence of two or more roles from different predicates. For example, consider the following report extract from a university where *each student may enroll at most twice in the same course*. Enrollments are identified by enrollment numbers, students by student numbers, and courses by course codes. Note that student 120 has enrolled twice in course CS100. The frequency constraint just mentioned would be violated if that student enrolled a third time in that course.

EnrollmentNr	*StudentNr*	*Course Code*
1001	120	CS100
1002	120	CS135
1003	501	CS135
3001	120	CS100

An ORM schema for this example is shown in Figure 8. The circled "≤ 2" external frequency constraint applied to the roles hosted by Student and Course indicates that for each state of the database, each (student, course) pair that populates those roles does so at most twice. This constraint verbalizes thus: **For each** Student **and** Course, **there are at most** 2 Enrolment **instances where that** Enrollment is by **that** Student **and** is in **that** Course.

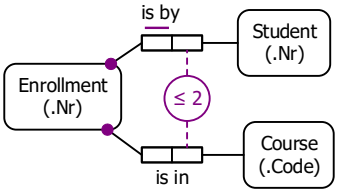


Figure 8 An ORM schema with an external frequency constraint.

This ORM schema may be coded in LogiQL as follows. The external frequency constraint is enforced by using the count function to derive the number of enrollments of a given student in a given course, and then constraining that number to be at most 2.

```
Enrollment(e), hasEnrollmentNr(e:n)  ->  int(n).
Student(s), hasStudentNr(s:n)  ->  int(n).
Course(c), hasCourseCode(c:cc)  ->  string(cc).
studentInvolvedIn[e] = s  ->  Enrollment(e), Student(s).
courseInvolvedIn[e] = c  ->  Enrollment(e), Course(c).
// Each Enrollment involves both a student and a course
Enrollment(e) -> studentInvolvedIn[e] = _, courseInvolvedIn[e] = _.

/* For each student s who enrolled in course c,
   n is the number of cases where there exists an
   enrollment e involving that s and that c */
nrEnrollmentsFor[s, c] = n  <-
   agg<<n = count()>>studentInvolvedIn[e] = s, courseInvolvedIn[e] = c.

// Each student enrolls at most twice in the same course
nrEnrollmentsFor[_, _] = n  ->  n <= 2.
```

Conclusion

The current article discussed how to declare internal and external frequency constraints in ORM and LogiQL. Various other constraints (e.g. value-comparison constraints, object and role cardinality constraints) may be specified graphically in ORM and coded in LogiQL. The core reference manual for LogiQL is accessible at https://developer.logicblox.com/content/docs4/core-reference/. An introductory tutorial for LogiQL and the REPL tool is available at https://developer.logicblox.com/content/docs4/tutorial/repl/section/split.html. Further coverage of LogiQL may be found in [20].

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley, Reading, MA.
2. Aref, M., Cate, B., Green T., Kimefeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. & Washburn, G. 2015, 'Design and Implementation of the LogicBlox System', *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, ACM, New York. http://dx.doi.org/10.1145/2723372.2742796.
3. Barker, R. 1990, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.
4. Curland, M. & Halpin, T. 2010, 'The NORMA Software Tool for ORM 2', P. Soffer & E. Proper (Eds.): *CAiSE Forum 2010*, LNBIP 72, pp. 190-204, Springer-Verlag Berlin Heidelberg 2010.
5. 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.
6. Halpin, T. 2014, 'Logical Data Modeling: Part 2', *Business Rules Journal*, Vol. 15, No. 10 (Oct., 2014), URL: http://www.BRCommunity.com/a2014/b780.html.
7. Halpin, T. 2015, 'Logical Data Modeling: Part 3', *Business Rules Journal*, Vol. 16, No. 1 (Jan., 2015), URL: http://www.BRCommunity.com/a2015/b795.html.
8. Halpin, T. 2015, 'Logical Data Modeling: Part 4', *Business Rules Journal*, Vol. 16, No. 7 (July, 2015), URL: http://www.BRCommunity.com/a2015/b820.html.
9. Halpin, T. 2015, 'Logical Data Modeling: Part 5', *Business Rules Journal*, Vol. 16, No. 10 (Oct., 2015), URL: http://www.BRCommunity.com/a2015/b832.html.
10. Halpin, T. 2016, 'Logical Data Modeling: Part 6', *Business Rules Journal*, Vol. 17, No. 3 Mar., 2016), URL: http://www.BRCommunity.com/a2016/b852.html.
11. Halpin, T. 2016, 'Logical Data Modeling: Part 7', *Business Rules Journal*, Vol. 17, No. 7 (July, 2016), URL: http://www.brcommunity.com/a2016/b866.html.
12. Halpin, T. 2016, 'Logical Data Modeling: Part 8', *Business Rules Journal*, Vol. 17, No. 11 (Nov, 2016), URL: http://www.brcommunity.com/a2016/b883.html.
13. Halpin, T. 2017, 'Logical Data Modeling: Part 9', *Business Rules Journal*, Vol. 18, No. 5 (May, 2017), URL: http://www.brcommunity.com/a2017/b906.html.

14. Halpin, T. 2017, 'Logical Data Modeling: Part 10', *Business Rules Journal*, Vol. 18, No. 11 (Nov, 2017), URL: http://www.brcommunity.com/a2017/b929.html.
15. Halpin, T. 2018, 'Logical Data Modeling: Part 11', *Business Rules Journal*, Vol. 19, No. 4 (April, 2018), URL: http://www.brcommunity.com/a2018/b949.html.
16. Halpin, T. 2018, 'Logical Data Modeling: Part 12', *Business Rules Journal*, Vol. 19, No. 7 (July, 2018), URL: http://www.brcommunity.com/a2018/b960.html.
17. Halpin, T. 2015, *Object-Role Modeling Fundamentals*, Technics Publications, New Jersey.
18. Halpin, T. 2016, *Object-Role Modeling Workbook*, Technics Publications, New Jersey.
19. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, *2nd edition*, Morgan Kaufmann, San Francisco.
20. Halpin, T. & Rugaber, S. 2014, *LogiQL: A Query language for Smart Databases*, CRC Press, Boca Raton. http://www.crcpress.com/product/isbn/9781482244939#.
21. 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.
22. 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.
23. Object Management Group 2017, *OMG Unified Modeling Language (OMG UML)*, version 2.5.1. Retrieved from http://www.omg.org/spec/UML/2.5.1/.
24. OMG, 2012, *OMG Object Constraint Language (OCL), version 2.3.1*. Retrieved from http://www.omg.org/spec/OCL/2.3.1/.