Mapping ORM to Datalog: An Overview

Terry Halpin[1], Matthew Curland[2], Kurt Stirewalt[2], Navin Viswanath[2], Matthew McGill[2], Steven Beck[2]

[1]LogicBlox, Australia and INTI International University, Malaysia
[2]LogicBlox, USA
e-mail: {terry.halpin, matt.curland, kurt.stirewalt, navin.viswanath, matt.mcgill, steven.beck}@logicblox.com

Abstract: Optimization of modern businesses is becoming increasingly dependent on business intelligence and rule-based software to perform predictive analytics over massive data sets and enforce complex business rules. This has led to a resurgence of interest in datalog, because of its powerful capability for processing complex rules, especially those involving recursion, and the exploitation of novel data structures that provide performance advantages over relational database systems. ORM 2 is a conceptual approach for fact oriented modeling that provides a high level graphical and textual syntax to facilitate validation of data models and complex rules with nontechnical domain experts. Datalog^{LB} is an extended form of typed datalog that exploits fact-oriented data structures to provide deep and highly performant support for complex rules with guaranteed decidability. This paper provides an overview of recent research and development efforts to extend the Natural ORM Architect (NORMA) software tool to map ORM models to Datalog^{LB}.

1 Introduction

In order to compete effectively in the information age, many businesses are exploiting information technology as a way to promote efficiency and reduce costs. For example, business intelligence tools and rule-based software are being increasingly used to perform predictive analytics over massive data sets and enforce complex business rules. This has led to a resurgence of interest in *datalog*, because of its powerful capability for processing complex rules, especially those involving recursion. Moreover, *novel data structures* such as column-oriented data stores are being exploited to provide performance advantages over relational database systems for complex analytics and data warehousing tasks (http://en.wikipedia.org/wiki/Column-oriented_DBMS).

While datalog and related technologies are powerful, the effective use of them typically requires a considerable level of mathematical sophistication. This often results in a communication gap when the business experts, who best understand the complex business rules and queries needed for their business, attempt to validate that the technical rules and queries used in the implementation actually conform to their requirements. This problem is best addressed by first formulating the models, rules and queries at a conceptual level where they can be reliably validated with the busi-

ness domain experts, and then automatically transforming these high level constructs into equivalent, lower level constructs (e.g. datalog code) for implementation.

This paper provides an overview of our recent research and development efforts to support such a model-driven engineering approach to business analytics. For the conceptual level, we use second generation *Object-Role Modeling* (*ORM 2*) [10]. Unlike attribute-based approaches such as Entity-Relationship (ER) modeling [5] and class diagramming within the Unified Modeling Language (UML) [23], ORM is *fact-oriented*, where all facts, constraints, and derivation rules may be verbalized naturally in sentences easily understood and validated by nontechnical business users using concrete examples. ORM's graphical notation for data modeling is far more expressive than that of industrial ER diagrams or UML class diagrams, and its attribute-free nature makes it more stable and adaptable to changing business requirements. Brief introductions to ORM may be found in [12, 15], a detailed introduction in [16], a thorough treatment in [18], and a comparison with UML in [14]. An overview of fact-oriented modeling approaches, including ORM and others such as RIDL [22], NIAM [24], and PSM [20], as well as history and research directions, may be found in [13].

For the datalog platform, we use *datalog*^{LB}, a vastly extended version of datalog developed by LogicBlox. Datalog^{LB} is a typed datalog [24] that employs fact-oriented data structures with performance benefits similar to those of column stores when processing very complex rules over vast data sets. For detailed coverage of traditional datalog, see [1, 6, 9]. Datalog^{LB} extends basic datalog with stratified negation, types, functions (including aggregate functions), transactions, modules (called "blocks"), constraints, default values, ordered predicates, metalevel support, and other features.

Early tool support for ORM introduced two textual languages. Formal ORM Language (FORML) was supported as an output verbalization language in InfoModeler and the ORM solution within Microsoft Visio for Enterprise Architects. Conceptual Query Language (ConQuer) enabled ORM models to be queried, and was implemented in the InfoAssistant and ActiveQuery tools [3, 4]. However, the ConQuer language was used only for formulating non-recursive ORM queries, not constraints or derivation rules, and tool support for it is no longer available.

Recently, ORM was extended to ORM 2, with tool support provided by Natural ORM Architect (*NORMA*) [8], including improved constraint verbalization in FORML 2 [17, 19] as well as further rule options such as semiderived types, deontic rules, and deep support for conceptual outer joins [18]. More recently, we developed a *role calculus* to formally capture derivation rules in ORM [7], and the VisualBlox team at LogicBlox has extended the NORMA tool to capture derivation rules and have also developed a VisualBlox tool to map ORM models to Datalog^{LB}.

Extensions to the NORMA tool allow ORM 2 derivation rules to be entered by clicking options in a Model Browser, and are then stored in a structure based on the role calculus, which offers a high level of semantic stability [7]. Compared to the ActiveQuery tool for ConQuer, NORMA's derivation support covers a wider range of rules (including recursion), has much better rule verbalization, and generates datalog code for implementation instead of SQL. While it is planned to add SQL generation for derivation rules at a later stage, our current efforts are focused on completing the datalog generation. NORMA's derivation language is designed to be relationally complete, and at the time of writing, about 90% of its constructs have been automatically transformed into Datalog^{LB}.

While the role calculus offers advantages such as compactness and semantic stability, its internal metamodel is technically challenging and its structures differ significantly from those of datalog or SQL. To simplify the task of transforming role calculus structures into target languages such as datalog and SQL, we first map the role calculus version of derivation rules to an intermediate structure based on the *domain relational calculus*, and then transform this second structure into the target code.

This paper provides a high level overview of some of this work, illustrating some of the mapping patterns by concrete examples. Discussion of the relevant metamodels and detailed transformation algorithms is beyond the scope of this paper, but portions of an early version of the role calculus metamodel may be found in [7].

The rest of this paper is structured as follows. Section 2 briefly illustrates how ORM object types, fact types, and constraints map to Datalog^{LB}. Section 3 discusses the basics of mapping ORM derivation rules map to Datalog^{LB}, including a rule for placing existential quantifiers. Section 4 considers some derivation rule examples involving use of scalar and aggregate functions. Section 5 summarizes the main results, outlines future research directions, and lists references.

2 Mapping ORM Object Types, Fact Types, and Constraints

In logic, an individual is a single thing of interest (e.g. a specific person, country, name, or number). An object in ORM corresponds to an individual in this sense. In first-order logic (FOL), predication is allowed only over individuals, not predicates, and quantification is allowed only over individual variables. First-order logic is undecidable, so there are some first-order formulas whose truth value can't be established by any algorithm. An algorithm to map ORM models into unsorted, first-order logic was provided by one of the authors in the late 1980s [10].

In the 1990s, the ConQuer query language for ORM was formalized in terms of sorted FOL, extended by a special operator for outer joins as well as set and bag comprehension [4]. Later, ORM 2 added modal operators to distinguish between alethic and deontic rules. Currently, deontic rules are ignored in mapping to datalog. While outer joins can be captured in NORMA derivation rules, their transformation to Datalog^{LB} awaits further work.

Datalog is designed for database work, and is a decidable fragment of first-order logic with powerful capabilities for storing, constraining, and deriving facts. As a logic programming language, datalog's support for recursive rules is more elegant and efficient than that provided by relational database systems. Unlike other logic programming languages such as Prolog, datalog programs are guaranteed to terminate.

Standard datalog uses prefix notation, with individual terms (individual variables or constants) listed in parentheses after the predicate name. In basic datalog, a *rule* is an expression of the form

$$q(\tau_1, \ldots, \tau_n) \leftarrow p_1(x_1, \ldots), \ldots, p_m(y_1, \ldots).$$

where the head predicate q has as argument an ordered list of individual terms $\tau_1, \ldots \tau_n$ ($n \geq 0$), each variable of which must occur in at least one argument of the body predicates $p_1 \ldots p_m$ ($m \geq 0$), the main propositional operator "$\leftarrow$" (read as "**if**") is the con-

verse implication operator from logic, and a comma "," (read as "**and**") between predications is the logical conjunction operator. A predication (the application of a predicate to a list of variables or constants) is also known as an *atom* or *positive literal*. The head or the body may be empty (but not both). A rule is treated as shorthand for a formula where the head variables are universally quantified at the top level, and any other variables introduced in the body are existentially quantified, with the existential quantifiers placed at the start of the body [1, p. 279]. For example, the following datalog rule

grandparentOf(*x, y*) ← parentOf(*x, z*), parentOf(*z, y*).

is equivalent to the following FOL formula (using mixfix predicates)

∀*x*∀*y*[*x* is a grandparent of *y* ← ∃*z*(*x* is a parent of *z* & *z* is a parent of *y*)].

Datalog adopts the closed world assumption (CWA), so if the same atom appears as the head of exactly *n* rules, the logical disjunction of the *n* rule bodies provides an if-and-only-if (iff) condition for the head. For example, the logical rule ∀*x*∀*y*[*x* is a parent of *y* ← (*x* is a father of *y* ∨ *x* is a mother of *y*)] may be set out in datalog as

parentOf(*x, y*) ← fatherOf(*x, y*).
parentOf(*x, y*) ← motherOf(*x, y*).

Datalog^{LB}, allows such *disjunctions* to be captured as a single rule, using a semicolon ";" for the inclusive-or operator. In datalog^{LB}, "←" is rendered as "<-" and no italics are used. So the above parenthood rule may be set out in Datalog^{LB} thus:

```
parentOf(x, y) <- fatherOf(x, y) ; motherOf(x, y).
```

Datalog extended with *negation* allows negated atoms (negative literals) in the body. Datalog^{LB} uses an exclamation mark "!" for the logical negation operator. An *anonymous variable* (denoted by an underscore "_" and read as "something") is used to existentially quantify a variable that is not referenced elsewhere in the formula (in which case the implicit existential quantifier has scope over only the atom in which the underscore occurs). For example, the derivation rule for living parents expressed as the FOL formula ∀*x*[*x* is a living parent ← (∃*y*(*x* is a parent of *y*) & ~ *x* died)] may be formulated thus:

```
livingParent(x) <- parentOf(x, _), !died(x).
```

Datalog^{LB} is a typed datalog, so each of its predicates is constrained to apply to a sequence of zero or more types. Object types are modeled in datalog^{LB} as unary predicates. Entity types are directly supported, but value types are currently handled as implicit subtypes of the associated data type. Type declarations are specified as constraints, or "right-arrow" formulas, using "->" (read as "**implies**" or "**only if**") for the material implication operator. Entity types that are identified using reference modes are declared along with their reference modes, using a colon ":" in the variable list of the reference predicate. For example, Country(.code) maps to:

```
Country(x), country:code(x:y) -> string(y).
```

An ORM fact type corresponds to a set of one or more typed predicates. A Datalog^{LB} predicate represents exactly one ORM fact type, so qualified predicate names are often used to distinguish predicates that have the same predicate reading in ORM.

For example, the *m:n* predicates in Person runs Company and Horse runs Race may be declared respectively as follows:

```
person:company:runs(x, y)  ->  Person(x), Company(y).
horse:race:runs(x, y)  ->  Horse(x), Race(y).
```

If a fact type has a uniqueness constraint spanning *n*-1 roles, a square-bracket notation is used to indicate the functional nature of the predicate. For example:

```
person:birthcountry[p]=c  ->  Person(p), Country(c).
```

Additional uniqueness constraints need to be declared separately. Variable names may include letters and digits. For example, the ORM schema in Figure 1 may be declared in Datalog^{LB} as follows, using the functional predicate declaration style to capture the left-hand uniqueness constraint on the head of government predicate and a separate clause to capture the right-hand uniqueness constraint.

```
Politician(p), politician:name(p:n)  ->  Politician(p), string(n).
Country(c), country:code(c:cc)  ->  Country(c), string(cc).
politician:countryGoverned[p]=c  ->  Politician(p), Country(c).
politician:countryGoverned[p1]=c, politician:countryGoverned[p2]=c  ->  p1=p2.
```

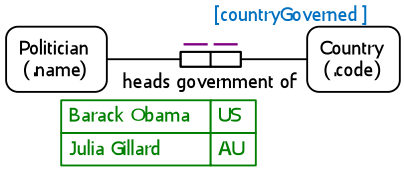

Fig. 1. A populated 1:1 ORM fact type.

The above code is an example of a Datalog^{LB} program. Data files are declared separately using delta predicates. For example, the data population in Figure 1 may be declared using the following assertions, where the "+" indicates insertion (addition of a fact to a predicate's population). Facts may be retracted (using "–") or modified using other options.

```
+politician:countryGoverned["Barack Obama"] = "US".
+politician:countryGoverned["Julia Gillard"] = "AU".
```

To illustrate the benefits of Datalog^{LB} for capturing ORM constraints, consider the ORM schema shown in Figure 2(a), which is fragment of a larger schema discussed elsewhere [16]. The equivalent Datalog^{LB} code shown in Figure 2(b). For discussion purposes, comments are inserted above the code for three constraints.

The mandatory role constraint that each book has a title is neatly expressed using an anonymous variable. The exclusion constraint that no book may be written and reviewed by the same person is also easily captured using negation. Finally, the acyclic constraint on the book translation predicate is enforced by introducing a recursively derived ancestor predicate and then declaring that to be irreflexive. This is much simpler than the equivalent SQL code, and also offers better performance.

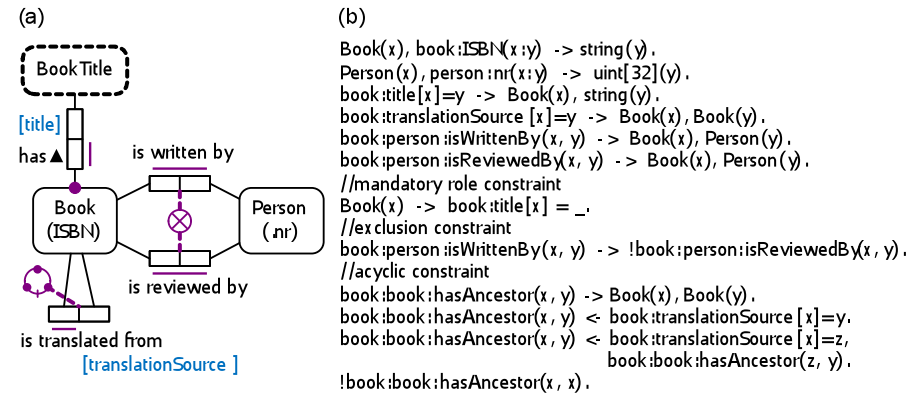

Fig. 2. (a) An ORM schema mapped to (b) Datalog^{LB}.

3 Mapping Derivation Rules

The above acyclic constraint enforcement introduced a derived fact type under the covers. ORM users may also introduce derived fact types of their own, and have NORMA map these to Datalog^{LB}. For implementation, we first capture the derivation rules in a role-calculus based structure, and then transform this to an intermediate, domain relational calculus structure, from which the Datalog^{LB} code is generated.

Derivation rules may be used to derive either subtypes or fact types. The NORMA screenshot in Figure 3(a) includes two derived subtypes and one derived fact type. Figure 3(b) shows how the associated derivation rules are displayed in the Model Browser after being entered by selecting and clicking the relevant options.

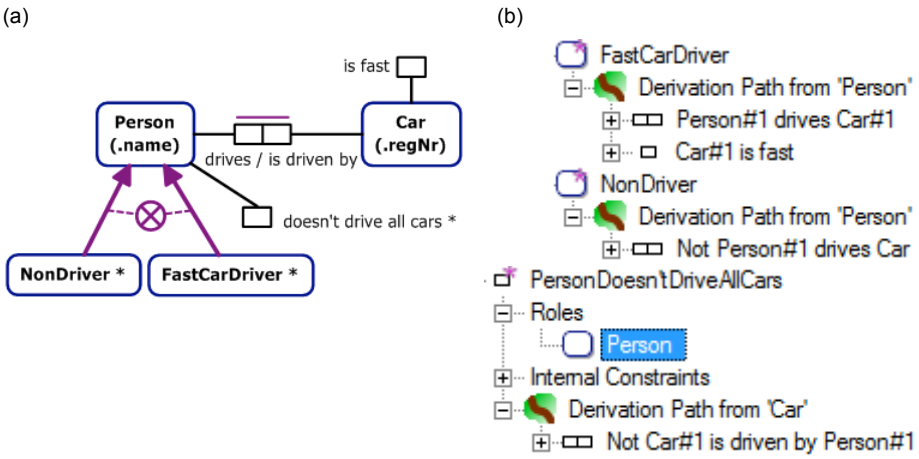

Fig. 3. NORMA screenshot of an ORM schema and its derivation rules.

The derivation path for the subtype FastCarDriver starts with Person (the path root) and navigates via the drives predicate to Car and then onto the isFast predicate, performing a conceptual join on Car. NORMA generates the following verbalization for the derivation rule: ***Each** FastCarDriver **is some** Person **who** drives **some** Car **that** is fast. The role calculus form of the rule is translated to a named tree structure representing the following sorted, relational calculus formula {*x*:Person | ∃*y*:Car (*x* drives *y* & *y* is fast)}. This is then transformed to an equivalent version of the following Datalog^{LB} rule, using standard techniques for reducing sorted to unsorted logic, and employing implicit quantification:

```
FastCarDriver(x) <- Person(x), Car(y), person:car:drives(x, y), car:isFast(y).
```

The derivation path for the NonDriver subtype starts with Person and then negates its entry into the drives predicate. This verbalizes as: ***Each** NonDriver **is some** Person **who** drives **no** Car. This maps to a named structure for the relational calculus formula {*x*:Person | ~∃*y*:Car *x* drives *y*}. For first-time users of datalog, the following rule may seem like an acceptable way to encode this rule:

```
NonDriver(x) <- Person(x), !(Car(y), person:car:drives(x, y)).          -- error!
```

However, the implicit existential quantification ∃*y* is before the negation rather than inside it, so the body is satisfied if *x* is a person, and there is anything that is not a car or is not driven by *x*. The range of *y* is unrestricted, so the rule is unsafe. In basic datalog, negands in the rule body are restricted to atoms, but in the above example the negand is a conjunction. Datalog^{LB} allows negated conjunctions if the variables in the negand are range restricted outside the negation (which is not true of the *y* variable in the above example). One solution is to generate the code in two steps, first deriving the opposite predicate and then negating it as shown below. It can also be done in one rule simply as NonDriver(x) <- Person(x), !person:car:drives(x,_).

```
Driver(x) <- Person(x), Car(y), person:car:drives(x, y).
NonDriver(x) <- Person(x), !Driver(x).
```

Figure 3 also includes the derived fact type in Person doesn't drive all cars. This is intended to return each person where there is at least one car not driven by that person. In this case, the derivation path starts with a car variable, and then uses negation to navigate to the person(s) who don't drive that car, and finally the derived role of Person is bound to that person variable. This verbalizes as: *Person doesn't drive all cars **if and only if for some** Car **it is not true that that** Person drives **that** Car.

A key aspect of generating the relational calculus version of the rule is knowing where to place existential quantifiers. Unprojected root variables are existentially quantified. Hence the derivation rule currently being discussed leads to the following relational calculus formula: {*x*:Person | ∃*y*:Car ~*x* drives *y*}.

ORM is essentially a sugared, visual version of sorted logic, hence in ORM each variable that is projected is a typed variable. The act of projecting on a typed variable in the scope of a negation ensures that the type declaration for that variable is lifted outside the negation. As a more general approach that works also with unsorted relational calculus, we introduce the following *Existential Placement Rule* (*EP*).

For each variable ν that occurs only in the rule body, place ∃ν immediately before the minimal wff that contains all the ν occurrences. Hence, if ν occurs only inside a negation then place ∃ν immediately after the negation symbol.

For the current derivation rule, the unsorted relational calculus formula with implicit quantification is {*x* | Person *x* & Car *y* & ~*x* drives *y*}. The only variable introduced in the rule body is *y*, and the minimal wff containing all its occurrences is Car *y* & ~*x* drives *y*. Applying EP now yields {*x* | Person *x* & ∃*y*(Car *y* & ~*x* drives *y*)}, which is equivalent to the sorted version given earlier. Using the Change of Scope rule ∃ν(*p* & Φν).≡. *p* & ∃νΦν where *p* is any wff in which ν does not occur free, this now maps to the Datalog^{LB} code shown below. In contrast to our NORMA and VisualBlox implementation, the ActiveQuery tool [4], although dealing well with many tasks, fails to provide correct semantics for this rule when formulated as a query.

```
doesntDriveAllCars(x) <- Person(x), Car(y), !drives(x, y).
```

4 Functions

Figure 4 shows an ORM schema with two derived fact types. The FORML derivation rules involve a multiply operator and a sum function (both are treated as functions in NORMA). An earlier paper discussed how to capture these two rules in the role calculus [7]. We now discuss their transformation to Datalog^{LB}. The asserted fact types map in the usual way. Assuming a float32 data type for AUDValue, the subtotal derivation rule maps to a named version of the relational calculus expression: {*li*:LineItem, *st*:Float32 | ∃*q*:Quantity ∃*p*:AUDValue (*li* hasQuantity *q* & *li* hasUnitPrice *p* & *st* = *q* * *p*)}. This maps to the following predicate declaration and rule in Datalog^{LB}:

```
lineItem:subtotalValue[x]=y -> LineItem(x), float[32](y).
lineItem:subtotalValue[x]=y <- lineItem:quantity[x]=q, lineItem:unitPrice[x]=p, y=q*p.
```

The invoice total rule uses the subtotal rule, generating a named version of the following relational calculus expression: {*i*:Invoice, *t*:Float32 | *t* = sum{*st*:AUDValue | ∃*li*:LineItem (*li* is on *i* & *li* has subtotal value *st*)}}. Datalog^{LB} includes a function called "total" to sum over sets or bags of numeric values. This function may now be applied to the derived subtotal predicate to derive the invoice total predicate.

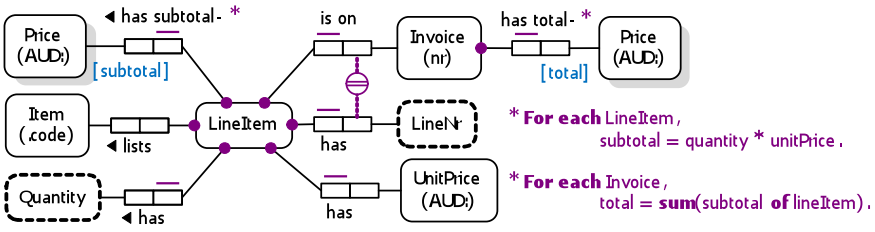

Fig. 4. Derivations involving a mathematical operator and aggregate function.

A special "agg" syntax is used for this as well as other aggregate functions (e.g. counts, minima and maxima). The type declaration and derivation rule for the invoice total is rendered by the following Datalog^{LB} code:

```
invoice:totalValue[x]=t  ->  Invoice(x), float[32](t).
invoice:totalValue[x]=t  <-  agg<< t=total(st) >> lineItem:invoice[li]=x,
                                                  lineItem:subtotalValue[li]=st.
```

5 Conclusion

This paper provided a high level overview of our recent work to automatically transform ORM models, including derivation rules, to an extended version of datalog, using a sorted, relational calculus based structure as an intermediate format between the initial role calculus based source structure and the final datalog code. A general procedure was introduced for placement of existential quantifiers in the intermediate structure to ensure that the desired semantics are achieved. To assist readers unversed in datalog, we have used simple, concrete examples to illustrate the main ideas. In practice, rules of far greater complexity are supported.

As future research, we plan to cater for Datalog^{LB} derivation rules that are not yet supported in NORMA and VisualBlox. For example, in Datalog^{LB} heights may be ordered using a meta-predicate and the top ranking function may then be used to return the top *r* height values via the following rule. Adding high level support for such rules will empower more users to exploit the expressive power of Datalog^{LB}.

```
heightRank:heightVal[r]=hv  <-  agg<<hv = top[r](y)>> height:cmValue(_:y).
```

Derivation rules and queries are *safe* only if they are guaranteed to execute in a finite time. Most versions of datalog implement safety using syntactic checks proposed by Ullman (e.g. all head variables must occur in the rule body, and any variables in an arithmetic or relational subgoal must also appear in a positive relational subgoal) [9]. While efficient to implement, these safety rules are in fact too strong (e.g. see [21]), and we are researching ways to accept rules in a more convenient format that can be transformed into Ullman-safe rules. Apart from more sophisticated support for safety, we plan to extend our ORM-to-Datalog^{LB} conversion to 100% coverage, add support for dynamic rules [2], and extend both ORM and our mapping procedures to exploit new features being added to Datalog^{LB} (e.g. existential variables in rule heads).

Acknowledgment: The assistance of our LogicBlox colleague Martin Bravenboer in providing helpful feedback on related work is greatly appreciated.

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley.
2. Balsters, H. & Halpin, T. 2008, 'Formal Semantics of Dynamic Rules in ORM', *On the Move to Meaningful Internet Systems 2008: OTM 2008 Workshops*, eds. R. Meersman, Z. Tari, P. Herrero et al., Monterrey, Mexico, Springer LNCS 5333, pp. 699-708.
3. Bloesch, A. & Halpin, T. 1996, 'ConQuer: a conceptual query language', *Proc. ER'96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, no. 1157, pp. 121-133.

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. 113-126.
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. Colomb, R. 1998, *Deductive Databases and their Applications*, Taylor & Francis Ltd, London.
7. Curland, M., Halpin, T. & Stirewalt, K. 2009, 'A Role Calculus for ORM', *On the Move to Meaningful Internet Systems 2009: OTM 2009 Workshops*, eds. R. Meersman, P. Herrero & T. Dillon, Springer LNCS 5872, pp. 692–703.
8. Curland, M. & Halpin, T. 2010, 'The NORMA Tool for ORM 2', *Proc. CAiSE-2010 Forum*, Tunisia.
9. Garcia-Molina, T., Ullman, J. & Widom, J. 2009, *Database Systems: The Complete Book, 2nd edition*, Pearson.
10. Halpin, T. 1989, 'A Logical Analysis of Information Systems: static aspects of the data-oriented perspective', doctoral dissertation, University of Queensland. Available as an 18 MB bitmap pdf file at http://www.orm.net/Halpin_PhDthesis.pdf.
11. Halpin, T. 2005, 'ORM 2', *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, eds R. Meersman, Z. Tari, et al., Cyprus. Springer LNCS 3762, pp 676-87.
12. Halpin, T. 2006, 'ORM/NIAM Object-Role Modeling', *Handbook on Information Systems Architectures*, *2nd edn*, eds P. Bernus, K. Mertins & G. Schmidt, Springer, Heidelberg, pp. 81-103.
13. Halpin, T. 2007, 'Fact-Oriented Modeling: Past, Present and Future', *Conceptual Modelling in Information Systems Engineering*, eds. J. Krogstie, A. Opdahl & S. Brinkkemper, Springer, Berlin, pp. 19-38.
14. Halpin, T. 2008, 'A Comparison of Data Modeling in UML and ORM', *Encyclopedia of Information Science and Technology*, *2nd edn*, vol. II, ed. M. Khosrow-Pour, Information Science Reference, Hershey PA, USA, pp. 613-618.
15. Halpin, T. 2009, 'Object-Role Modeling', *Encyclopedia of Database Systems*, ed. L, Liu & M. Tamer Ozsu, Springer-Verlag, Berlin.
16. Halpin, T. 2010, 'Object-Role Modeling: Principles and Benefits', *International Journal of Information Systems Modeling and Design*, Vol. 1, No. 1, IGI Global, pp. 32-54.
17. Halpin, T. & Curland, M. 2006, 'Automated Verbalization for ORM 2', On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, eds. R. Meersman, Z. Tari, P. Herrero et al., Montpellier. Springer LNCS 4278, pp. 1181-90.
18. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, *Second Edition*, Morgan Kaufmann, San Francisco.
19. Halpin, T. & Wijbenga, J. 2010, 'FORML 2', *Enterprise, Business-Process and Information Systems Modeling*, eds. I. Bider et al., LNBIP 50, Springer, Heidelberg, pp. 247–260.
20. ter Hofstede, A., Proper, H. & van der Weide, T. 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
21. Hull, R. & Su, J. 1993, 'Domain Independence and the Relational Calculus', *Technical Report 88-64*, Comp. Science Dept., University of Southern California.
22. Meersman, R. 1982, *The RIDL Conceptual Language*, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.
23. Object Management Group 2003, *UML 2.0 Superstructure Specification*. Online at: www.omg.org/uml.
24. Wintraecken J. 1990, *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, Deventer, The Netherlands.
25. Zook, D., Pasalic, E., & Sarna-Starosta, B. (2009). Typed Datalog. In *Practical Aspects of Declarative Languages (PADL'09)*, LNCS 5418 (pp. 168-182). Berlin: Springer.