Modeling Collections in UML and ORM

Terry Halpin

Microsoft Corporation, USA
email: TerryHa@microsoft.com

[This paper first appeared in Halpin, T.A. 2000, 'Modeling collections in UML and ORM', *Proc. EMMSAD'00: 5th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, ed. K. Siau, Kista, Sweden (June), and is reproduced here by permission.]

Abstract: Collection types such as sets, bags and arrays have been used as data structures in both traditional and object oriented programming. Although sets were used as record components in early database work, this practice was largely discontinued with the widespread adoption of relational databases. Object-relational and object databases once again allow database designers to embed collections as database fields. Should collections be specified directly on the conceptual schema, as mapping annotations to the conceptual schema, or only on the logical database schema? This paper discusses the pros and cons of different approaches to modeling collections. Overall it favors the annotation approach, whereby collection types are specified as adornments to the pure conceptual schema to guide the mapping process from conceptual to lower levels. The ideas are illustrated using notations from both object-oriented (Unified Modeling Language) and fact-oriented (Object-Role Modeling) approaches.

1 INTRODUCTION

Information systems may be modeled at various levels. For analysis purposes, a conceptual schema of the application domain should first be developed. At this stage, modelers communicate with domain experts in their own terms, making it easier to get a clear, correct and complete picture of the requirements. Once the business model is understood, it can be implemented by mapping the conceptual schema to logical, physical, and external schemas. During this mapping process, further implementation details may be added. For database applications, the mapping from a conceptual schema to a logical database schema often involves a transformation in data structures. For example, a conceptual association may map to a relational table attribute. Though commonly used in programming, the practice of using *collection types* (e.g. sets, bags and arrays) as record components in database schemas largely disappeared with the widespread adoption of relational databases, where each table column is based on an atomic domain. However, the recent emergence of object-relational and object databases once again allows collections as database fields.

How should collection types be specified within the conceptual analysis and logical design of data? Some approaches use collections directly within the conceptual schema, some specify them as annotations to the conceptual schema, while others relegate them to the logical schema. This paper examines these alternative approaches by way of examples, highlighting the conceptual and pragmatic issues underlying such choices. Its focus is on how and where to specify collection types in the modeling process, not on the advisability or otherwise of using collection types in a database implementation. The ideas are illustrated using notations from both object-oriented (Unified Modeling Language (UML)) and fact-oriented (Object-Role Modeling (ORM)) approaches. The discussion is also relevant to some extended Entity Relationship (ER) approaches.

Procedures for mapping from conceptual models to logical models are readily available (e.g. [3, 11, 13, 27]). For relational database systems, denormalization and fragmentation are often used to improve performance. Object-relational and object databases allow further denormalization, since table fields may contain collections rather than just atomic values. In practice, these performance-based design options are usually specified on the logical or internal model, and often the conceptual model is ignored after these changes are made. This practice can lead to weak coupling or inconsistencies between the conceptual and lower level models, making it difficult to leverage the conceptual model to facilitate schema evolution as the business changes, or to provide a conceptual means of accessing the data. One solution to this problem is to include collection types within the conceptual schema itself, where these collections map directly to

similar structures in the implementation. Another solution is to exclude collections from the pure conceptual schema us ed to validate the business model with the domain expert, but afterwards allow collections to be specified as annotations to the conceptual model to guide the mapping to and from the implementation model. Both these solutions allow tools to synchronize the conceptual and the lower level models. Sometimes a mixture of these two solutions is used.

The notion of annotating conceptual models is not new. Berglas proposed conceptual annotations for various rules, meta-authorization protocols and inheritance control [2], and both UML and ORM include constructs that can be viewed as conceptual annotations. UML class diagrams may include conceptual features (e.g. classes, associations, multiplicity constraints) as well as object-oriented implementation features (e.g. navigation directions, visibility settings). When stripped of implementation features, UML class diagrams may be regarded as an extended version of ER, for use in conceptual data modeling. Although UML has no sharp dividing line between conceptual and lower level features, implementation details may be thought of as annotations to the conceptual or analysis model. The UML standard may be accessed on [28]. Detailed discussions of UML may be found in [6, 29] and (with some notational differences) in [3].

ORM is a conceptual modeling approach that makes no use of attributes as a base concept, always using a relationship instead. For example, the attribute Employee.birthdate is modeled in ORM as the fact type: Employee was born on Date. Object types in ORM are semantic domains, hence the connectedness of the model is always apparent. While this approach leads to larger diagrams, it promotes simplicity, semantic stability, verbalizability and populatability. ORM includes rich graphical and textual languages for capturing a wide range of business rules. Although such detail is needed to fully capture and transform models [19], various abstraction mechanisms allow the modeler to hide unwanted detail. One abstraction displays minor fact types as attributes of major object types [7], enabling ER-models and OO-models to be derived as views. An overview of ORM may be found in [14], a detailed discussion in [13], and comparisons between ORM and UML data modeling in [15, 16, 17].

The next section discusses whether set collections should be modeled conceptually as first class types or instead as annotations. Though the focus here is on sets, most of the discussion applies to collections in general. Collection types other than sets are then considered. The conclusion summarizes the main points and indicates areas of further research.

2 SETS AS COLLECTION TYPES

Conceptual data modeling has two basic data structures: relationship types or associations, and object types or classes (and data types). In ER and UML, an object type may have attributes, and in a sense is a named collection or aggregation of attributes (cf. a relational table as a named set of attributes, populated by sets of tuples). Unless specified as a collection however, object types are treated as simple (e.g. an instance of Employee is a single employee, not a set of employees). UML includes three ways in which collection types may effectively be introduced: stereotypes; multi-valued attributes; and constraints.

UML does not predefine any data types, leaving it up to modelers to define their own type systems. For example, collection types may be specified directly, as stereotypes of classes, as shown in Figure 1. This is similar to the way collection types were specified in version 1.0 of the Open Information Model [21]. Collection types may be realized by implementation classes [e.g. 29, p. 485], and may be homogeneous (all its elements are of the same type, as in Figure 1) or heterogeneous (elements of different type).

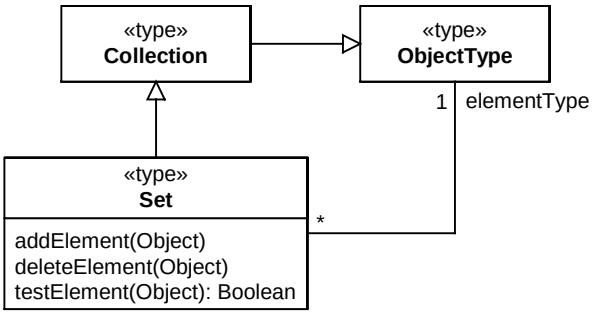

Figure 1 In UML, collection types may be specified as stereotypes of class

Some target systems support no collections, many support homogenous collections only, and some support homogeneous and heterogeneous collections (e.g. Object Store). However such specifications are typically used for program code design rather than conceptual modeling, so our discussion of UML will focus on the other options: multi-valued attributes and constraints.

Figure 2(a) shows a simple ORM model. Object types are displayed as named ellipses, with solid lines for entity (non-lexical) types and dashed lines for value (lexical) types. Associations are named sequences of roles, where each role appears as a box connected to the object type playing it. Injective associations providing a reference scheme may be abbreviated in parentheses (e.g. empNr). A black dot indicates that a role is mandatory (e.g. each Employee has an EmpName). Arrow-tipped bars over one or more roles are uniqueness constraints, indicating that each instance populating that role sequence is unique. This example includes the four possible uniqueness patterns for binary associations: *n:1* (each employee has at most one employee name); *1:1* (each employee heads at most one department, and vice versa); *m:n* (it is possible that the same employee drives many cars, and that the same car is driven by many employees); and *1:n* (an employee may have many nicknames, but each nickname refers to at most one employee).

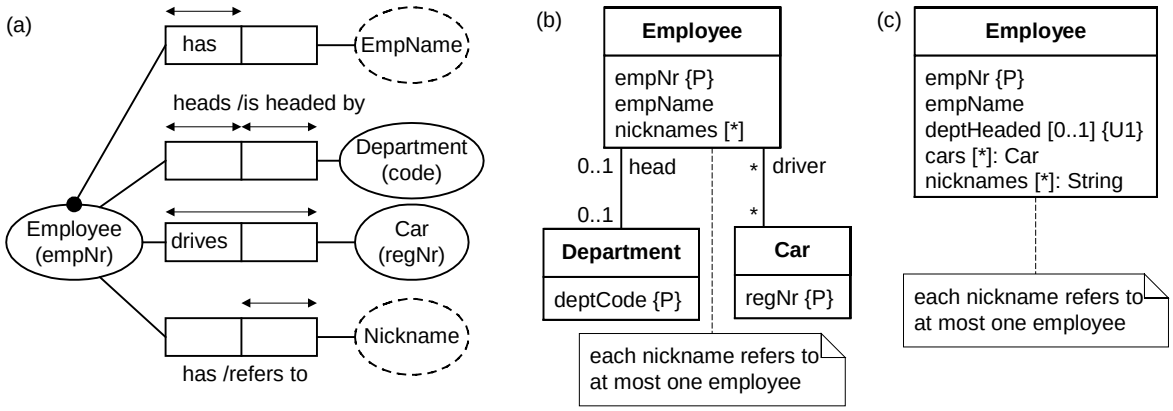

Figure 2 Four binary associations in ORM modeled as associations or attributes in UML

Figure 2(b) shows a typical way of modeling this in UML. Department and Car are non-lexical object types, for which we probably want to record various properties not shown here, so they are modeled as classes. The heads and drives information relates non-lexical types, so is modeled using associations. Since employee number, employee name and nickname and lexical, they would usually be modeled as attributes, as shown. The empNr and empName attributes are single-valued as in traditional ER. In UML the default multiplicity of an attribute is 1 (exactly one). So empNr and empName are mandatory (at least one) and single valued (at most one). Attribute multiplicities may be shown in square brackets after the attribute name. UML allows standard and user-defined constraints to be added in braces. In the absence of standard UML notation for primary reference, we use our own symbol {P}.

Figure 2(c) shows an alternative way of modeling this situation that uses attributes only. If we decided for some performance reason to implement all the information in a single object-relation, this choice would be good for displaying the actual implementation model. The 0..1 multiplicity for deptHeaded indicates it is optional and single-valued. In the absence of standard UML notation for uniqueness of attributes, we use our own symbol {U1}.

UML allows *multi-valued attributes* as an alternative to m:n and 1:n associations. In Figure 2(c), both cars and nicknames are multi-valued attributes whose instances are sets. This is one way of specifying set collections on a conceptual schema. The * (or 0..*) multiplicity indicates “zero or more”. An additional constraint is required to ensure that each nickname refers to at most one employee. A simple attribute uniqueness constraint (e.g. {U2}) is not enough, since nicknames is set-valued. Not only must each nicknames-set be unique for each employee, but each element in each set must be unique (the second condition implies the former). In both the UML models, this more complex constraint is specified informally in an attached note. From an implementation standpoint, this constraint is expensive if the structure is actually mapped to a set-valued attribute.

In conceptual analysis, the model should be easily communicated and validated with the domain expert. For this purpose, explicit associations are typically more suitable than attributes, especially multi-valued attributes. Why? Associations and related constraints are easily verbalized as sentences readily understood by non-technicians. Moreover, constraints can more easily be checked using populations. Each association in Figure 2(a) can be populated with data illustrating which uniqueness pattern applies, and counter-examples can be added for acceptance or rejection by the domain expert. ORM was specifically designed to facilitate this process, by having a role box associated with each column in the fact table. These benefits of ORM are exploited in Microsoft Visio Enterprise, which provides automated support for both verbalization and population of ORM models. This model validation could be done manually in UML, assuming extensions to disambiguate ordering within non-binary associations. UML provides object diagrams where each instance may be displayed separately with its own attribute values, but these are awkward for checking constraints against multiple instantiations [16, 17]. Associations allow various constraints on the "role played by the attribute" to be expressed in standard notation, rather than resorting to non-standard or complex extensions (as in our braced comments).

Associations tend to be more stable than attributes. Consider the association: Employee drives Car. To record how many accidents an employee had when driving a given car, in UML we make a Drives class out of the drives association (the same name must be used for both association and class) and add the attribute Drives.nrAccidents. In ORM we objectify the drives association as Driving, and add the association Driving had NrAccidents. If instead we modeled the drives feature as an attribute, we could not add new details without first changing our original schema to replace the attribute by an association or class. Populating attributed classes leads to null values, with all their attendant problems, and the connectedness between an attribute and its domain may not be obvious. This is one reason to use attributes only when their domain is a simple data type rather than an object class. Finally, queries involving multi-valued attributes often require some way of extracting their components, and hence complicate the query process for users.

For these reasons, multi-valued attributes should be avoided in conceptual models, which need to be validated with the domain expert. If attributes are used to show developers that an *m:n* or *1:n* association will be implemented as a multi-valued attribute, this could be done as a sub-conceptual view of the association version of the model. For example, assuming some way of cross-referencing, we could use Figure 2(b) as an analysis model and Figure 2(c) as a design model. This objective can also be achieved by annotations, as discussed later.

Although UML allows collection types to be introduced directly as stereotypes, base ORM, as supported in industry tools, has no such feature. However various ORM extensions, such as the Predicator Set Model (PSM) [22, 23, 24, 25] do support collection types (sets, sequences, unique sequences, bags and schemas). These extensions often model such *collections as first class types*. The schema type may be used to provide an argument on which operations may be defined ([10], [9]) in a similar way to UML's encapsulation of operations within classes. Although relationship objectification and composite identification do not involve collections, these features along with collection types are sometimes regarded as abstraction mechanisms [9]. Abstraction and view mechanisms help manage not just *information quantity* but also *information complexity*. As the underlying DBMS structures grow more complex, so do languages that deal directly with them. For example, the addition of constructors for building sets, bags and lists requires additional language features to not only declare such structures but also manipulate them. The language user now has *many more choices* as to how to formulate a model, update or query.

ORM supports some object-oriented features directly (e.g. subtyping [18] and composite domains), and can be extended with other object-oriented features. If these extensions are provided as view mechanisms or annotations rather than by adding them to the base ORM framework, the simplicity and stability of ORM models and queries can be retained at the same time as performance is improved by use of non-relational implementation structures. In both UML and ORM, use of object-identifiers (oids) and complex object constructors (e.g. set or array constructors) seems best relegated to the logical level rather than the conceptual level.

An analysis of how collection types have been used in practice indicates that such constructors should be used with extreme care, if at all, in developing a base conceptual model. Undisciplined use of constructors often leads to incorrect or incomplete solutions, or to overlooking a simpler solution that does not use collections. However, there are three main arguments in favor of conceptual constructor usage. Firstly, they lead to more compact models. Secondly they explicate the connection between conceptual and physical models (for a non-relational implementation). Thirdly they allow *unnamed structures* to be modeled directly. Let's examine these claims briefly, using set collections.

Collection types can reduce the size of a model diagram, but often at the expense of hiding other detail. The compactness advantage disappears if the collection-view can be generated as an abstraction, which is certainly possible. Collection types explicate the connection to an implementation model that uses those collections as storage structures. However this can also be achieved by specifying the collections as annotations to the conceptual model rather than as first class types.

This leaves us with the advantage (or otherwise) of modeling unnamed structures as first class types. There are several examples in the literature used in support of this proposal. As a classic example [20], consider a convoy as an unnamed *set* of ships. This may be modeled in ORM using a set collection type, as shown in Figure 3(a). Various notations exist for collection types. In this paper, ORM collection types are depicted as soft rectangles, with the collection kind (set, bag etc.) indicated in bold.

Figure 3(b) shows an alternative flat model in both ORM and UML using an association, and a name to identify each convoy. In practice, convoy names are very useful, making it easy to verbalize facts about convoys, and to check constraints. The uniqueness constraint on the ORM model asserts that each Ship is in at most one Convoy. This constraint is captured in the UML diagram also, since the solid diamond indicates composition (strong aggregation). This constraint is missing from Figure 3(a). In most cases, use of collection types makes it harder to express constraints on collection members. The association in the flat model can also be conveniently populated with fact instances, to help validate the constraint.

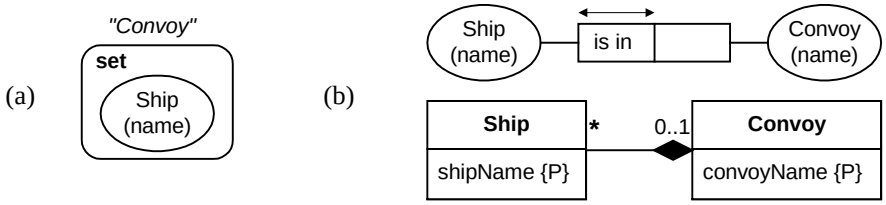

Figure 3 Convoy modeled as: (a) an unnamed set type (b) a named simple object type

For snapshot purposes, it is convenient to identify a convoy by its name (or by its flagship). The convoy membership fact type can be temporalized to maintain an historical record of membership, directly in temporal ORM, or in non-temporal ORM either by using the ternaries Ship joined Convoy on Date and Ship left Convoy on Date, or by nesting the historical m:n association (Ship was in Convoy) as ConvoyMembership, and adding ConvoyMembership began on Date; ConvoyMembership ended on Date. In UML, this history may be modeled using ConvoyMembership as an association class with appropriate date attributes (possibly multi-valued) or associations. Whether it is philosophically correct to use the word "convoy" to allow a convoy to be the same convoy after a ship leaves is of little practical interest.

As a related example, consider the notion of team membership, where it is possible for a person to be a member of many teams and vice versa. Figure 4 shows how to model this in ORM with and without the set constructor. In Figure 4(b) the spanning uniqueness constraint indicates the association is m:n. The circled "eu" is an "extensional uniqueness" constraint [25], to declare that teams may be defined by their membership (i.e. different teams cannot have exactly the same set of members). This constraint is implicit in Figure 4(a). Its use in Figure 4(b) allows unnamed structures without introducing collection types.

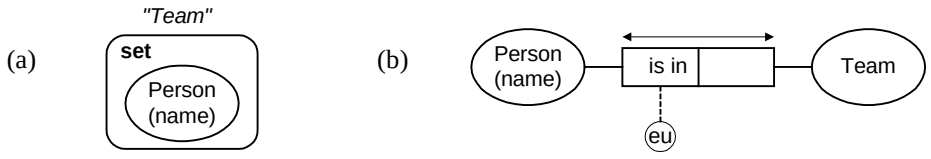

Figure 4 Two ways of modeling team membership

An alternative notation for this constraint is discussed in [1], where it is argued that if teams are unnamed in the real world, this must be reflected in the conceptual model, which must obey the conceptualization principle [26] by modeling only conceptually relevant features. We adopt a relaxed reading of this principle, encouraging the introduction of identifiers that improve model usability. Here it is

advisable to add a simple identifier for teams (e.g. a team name) since this facilitates talk about teams. If Figure 4(b) depicts the as-is model, the to-be model would add "(name)" as a reference scheme for Team. The extensional uniqueness constraint is still required if we definitely don't want two different teams to have the same membership. This constraint is very expensive to enforce, so should only be used when necessary.

A complex example cited in the ORM literature to justify the use of constructors is the chemical reaction model [12]. This models reactions such as: $2H_2 + O_2 \leftrightarrows 2H_2O$. One way of modeling this with a set collection type is shown in Figure 5(a). This model, based on [22] treats each side of a reaction equation as a set of chemical quantities, e.g. $\{(H_2, 2), (O_2, 1)\}$. The reaction itself is then captured in the binary input/output association. An alternative flat model is shown in Figure 5(b). Here a reaction number is introduced as a convenient way of referring to reactions. This model also shows a complex example of the extensional uniqueness constraint.

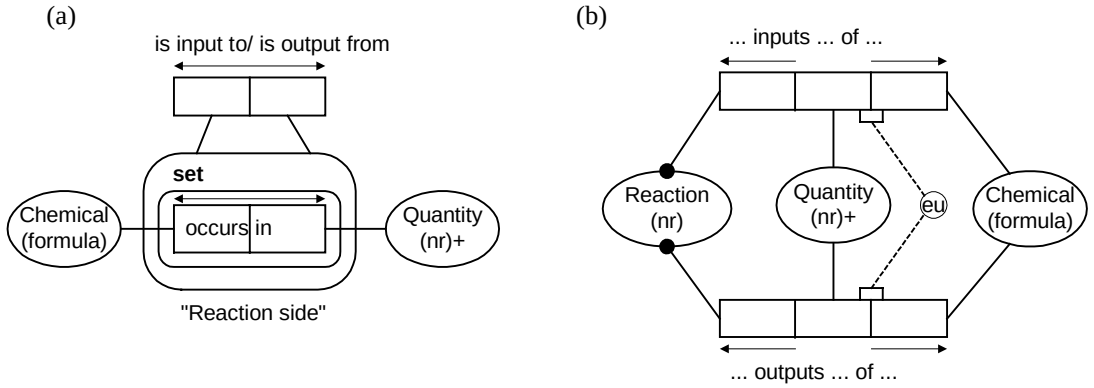

Figure 5 Chemical reactions modeled (a) with and (b) without sets

Although Figure 5(a) is more compact, its use of a set constructor makes it harder to think about. As evidence of this, it lacks a simple constraint captured in Figure 5(b). Can you spot it? The uniqueness constraints in Figure 5(b) ensure that each chemical occurs at most once in each reaction side (e.g. to exclude equations such as $5H \leftrightarrows 3H + 2H$). This is not enforced in Figure 5(a). For example, we may populate the input/output association with the tuple $\langle\{(H, 5)\}, \{(H, 3),(H, 2)\}\rangle$. The set-based model makes this constraint harder to conceive and express.

In Figure 5(a) the reaction association is m:n (presumably other conditions such as catalysts and temperature could be varied to have the same input reagents produce different results). If we instead demand that the same input always gives the same output, this association becomes 1:1. This is easy to model in Figure 5(a), but harder to model in Figure 5(b). This is one of those rare cases when we encounter a constraint on a collection that cannot be easily modeled as a constraint on the members. The opposite situation (a constraint on members that is hard to model on collections) is far more common.

An alternative way to model chemical reactions using collections is discussed in [12], which models a reaction as a set of sets of object-role pairs. Although the structure of this model is difficult to comprehend, even for modelers much less the domain experts, it is argued to be the only correct way of modeling the situation, based on a deterministic principle that one birth transition instance must correspond to exactly one object instance. Our viewpoint is that communication is more important than determinism, and there are often many correct ways to model the same application.

Hence there seems to be no compelling case to introduce collection types as first class citizens in the conceptual model. We still need some way of being able to specify collections for implementation in object-relational and object databases. While this can be done directly on the logical schema (as supported in some tools), it is desirable that there be a two-way transformation between conceptual and logical levels, so that changes in one can be communicated to the other to keep the models synchronized. The simplest way to do this is to specify collection type mapping as *annotations* to the pure conceptual schema, resulting in an annotated conceptual schema used only by the developer, not the domain expert, for the sole purpose of controlling the mapping between conceptual and lower levels.

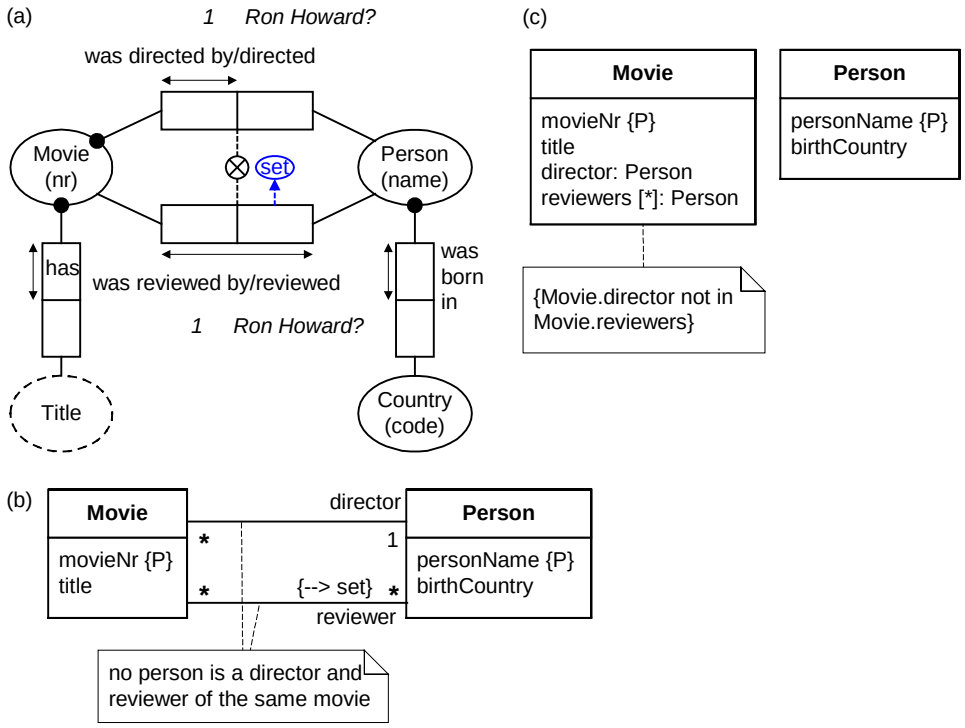

Figure 6 A set annotation as an extension to (a) ORM and (b) UML, leading to implementation (c)

As a simple example, see Figure 6(a) and (b). Here the reviewer role played by Person is annotated to indicate that its instances will be mapped as a set-valued attribute of Movie in the actual implementation, as shown more directly in Figure 6(c). For ORM, this already has tool support, though the current notation differs from the one shown here. For UML, a {--> set} constraint might be used for this purpose; though not standard, this is analogous to UML's {ordered} constraint (see next section).

Figure 6(a) includes an exclusion constraint between the director and reviewer associations. Depicted by "⊗", this constraint verbalizes as: **no** Person directed **and** reviewed **the same** Movie. ORM tools treat this constraint as formal, and generate DDL code to enforce it. A counterexample has been added to test this constraint with the domain expert: Can Ron Howard both direct and review movie 1? UML has no graphic symbol for this constraint, so it is added informally as a note. It could be specified formally in UML's Object Constraint Language (OCL) [30], but rules in OCL are typically too mathematical for the average domain expert to understand.

For a relational implementation, if the set annotation is removed from Figure 6(a) or (b), the default mapping generates three tables (*Movie*(movieNr, title, director), *Person*(personName, birthCountry), *Reviewed*(personName, movieNr)), three foreign key references (Movie.director references Person, Reviewed.personName references Person, Reviewed.movieNr references Person) and the exclusion constraint assertion **check** (**not exists** (**select** * **from** Movie **natural join** Reviewed **where** director = personName).

For an object-relational implementation, if the set annotation is included, the review facts are grouped as a set-valued field in the Movie table, leading to two tables (*Person*(personName, birthCountry), *Movie* (movieNr, title, director, reviewers: **set of** Person)), one foreign key reference (Movie.director references Person) and appropriate to code to enforce the constraint director **not in** reviewers. Figure 6(c) shows this implementation more directly, with one constraint captured informally as a note. Although this figure might assist the developer once the implementation choice has been made, it should not be used for the conceptual model. The review facts could be grouped into a separate table, a Movie table or even a Person table. The modeler needs to consider carefully the impact on queries and updates (e.g. component access) before deciding which is the best choice. These considerations are not relevant to conceptual analysis, where a non-annotated version of Figure 6(a) or (b) should be used.

3 OTHER KINDS OF COLLECTION TYPES

Although a number of collection types were slated for inclusion in the object-relational database standard SQL:1999, the only one that made it was *array* (a one dimensional array with a maximum number of elements). It is anticipated that three further collection types will be added in SQL200n: *set* (unordered collection with no duplicates); *multiset* (bag, i.e. unordered collection that allows duplicates); and *list* (sequence, i.e. an ordered bag). Some commercial systems already support these. Our experience with these systems is that little performance gain is actually achieved by use of collection types. This may change as the technology matures. Array, set, bag and list are also included as collection types in the object database standard ODMG 2.0 [8]. Currently UML includes none of these as standard constraint annotations, but does include the constraint {ordered} to indicate mapping to an *ordered set* (i.e. a sequence with no duplicates).

A simple example of a flat model for the bag (multiset) constructor is shown in Figure 7. Again, an identifier for bags may be introduced.

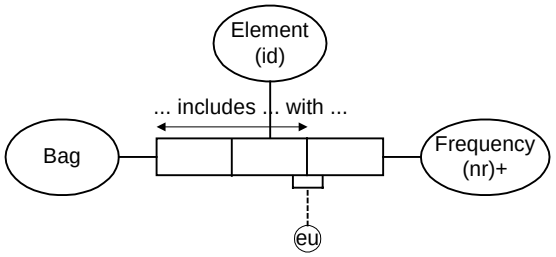

Figure 7 Flat metamodel of a bag

A sequence is an ordered bag, and in ORM its collection type is marked "seq". If the sequence cannot have duplicates, it is a "unique sequence" and is marked "seq". As an example of the unique sequence (or ordered set) constructor, see Figure 8(a). Here an author list is a sequence of authors, each of whom may appear at most once on the list. This may be modeled in flat ORM by introducing a Position object type to store the sequential position of any author on the list, as shown in Figure 8(b). The uniqueness constraint on the first two roles declares that for each paper an author occupies at most one position; the constraint covering the first and third roles indicates that for any paper, each position is occupied by at most one author. The textual constraint indicates that the positions in any list are numbered sequentially from 1. Although this ternary representation may appear awkward, it is easy to populate and it facilitates any discussion involving position (e.g. who is the second author for paper 21?). From an implementation perspective, a sequence structure could still be chosen: this can simplify updates by localizing their impact. However the update overhead of the positional structure is not onerous, given set-at-a-time processing (e.g. to delete author *n*, simply set position to position –1 for position > *n*).

Though not shown here, the ternary solution can also be modeled in UML. If the ternary model is chosen as the base model, it would be useful to support the annotated binary shown in Figure 8(c) or Figure 8(d) as a view of the base model. In ORM a unique sequence annotation is connected to the relevant role. This representation is equivalent to the {ordered} constraint in UML, as shown in Figure 8(d), indicating that the authors are to be stored as a unique sequence.

As an example of a pure sequence (allowing duplicates), a paragraph may be modeled as a sequence of sentences. Arrays may be handled in base ORM by using a numeric object type that explicitly denotes the ordering. Use of arrays in mapping may be indicated by a square brackets marker, optionally including array size. For example, for the association Athlete holds Rank in Sport a "[10]"annotation at the right end of the pair of roles played by Athlete and Rank indicates that this information will be stored within Sport in an array of size 10 encoding Rank as the array index.

When it comes to querying models as opposed to building them, users are typically far less skilled than modelers, and it seems unreasonable to burden users with the complexity of accessing information via constructors. At least at the query level, users should be able to formulate their queries in plain language without having to know how facts are grouped into structures in the underlying database or how particular constructors may be used in the query. An ORM conceptual query tool has been developed with just this in mind [4, 5]. The mapping of a conceptual query to an internal query is a job for an expert, not the average user. A query tool should perform this mapping automatically.

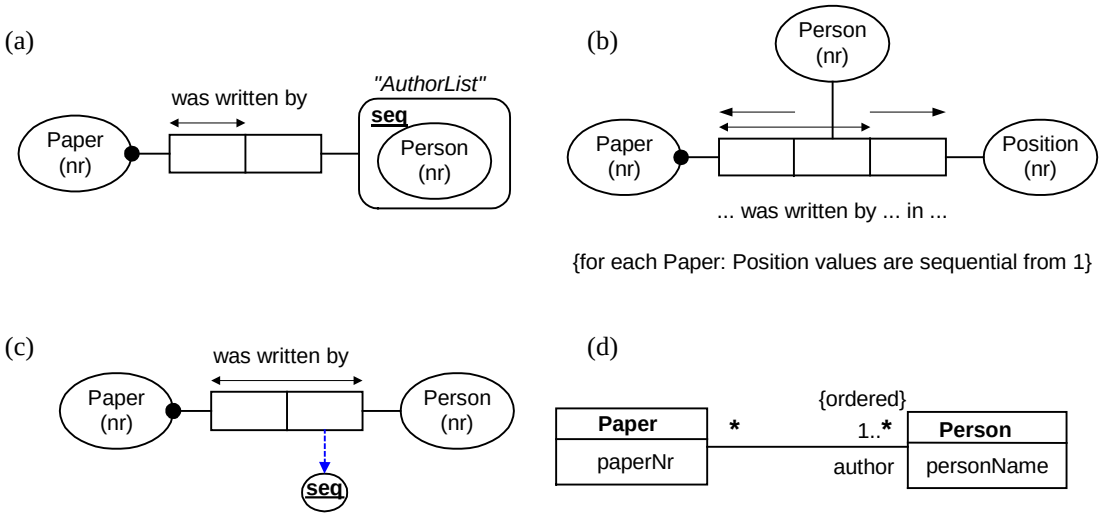

Figure 8 Unique sequences modeled in different ways

4 CONCLUSION

Although direct use of collection types in a conceptual model can lead to compactness, an alternative flat model (with no constructors) of comparable convenience or only minor inconvenience is generally available. Use of constructors often makes it harder to express constraints (which typically occur on members, not collections). Moreover, because constructors give rise to compound facts, they are awkward to populate or check for validation. To make matters worse, use of constructors requires more sophisticated modeling skills (even good modelers are prone to make subtle mistakes with them). Constructors also tend to be over-used (i.e. they are used even when there is there is a simpler modeling solution without them). It seems better to free the conceptual modeler from this added complexity, instead relegating constructor usage to annotations that guide the later mapping phase under automated support.

If collection types are used in the implementation, the use of collection-valued attributes or classes may have some use in a design version of the data model, in order to show the implementation structures directly. In this case, tool support should be provided to enable the modeler to move between analysis and design models at will.

Clearly, if constructors are to be used at all in conceptual modeling, some strong modeling guidelines need to be identified so that the appropriate constructors are judiciously chosen. This applies even more so for choosing collection types in the implementation model. The development of such design guidelines, especially for logical/physical models, is recommended as a practical area of future research.

The current release of our Visio Enterprise product includes support for object-relational databases, including collection types (and row types, installable types etc.) as well as basic annotations to ORM models. It also includes support for UML for object-oriented code design. More flexible support for collection types within both ORM and UML is currently being investigated.

References

1. Bakema, G.P., Zwart, J.P.C. & van der Lek, H. 1994, 'Fully communication oriented NIAM', *Proc. NIAM-ISDM Conf.*, Albuquerque, NM., pp. L1-35.
2. Berglas, A. 1994, 'Using annotated conceptual models to derive information system implementations', *Australian Journal of Information Systems*, vol. 1, no. 2.
3. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.
4. Bloesch, A. & Halpin, T. 1996, 'ConQuer: a conceptual query language', *Proceedings of the 15th International Conference on Conceptual Modeling ER'96* (Cottbus, Germany), B. Thalheim ed., Springer LNCS 1157 (Oct.) 121-133.

5. Bloesch, A. & Halpin, T. 1997, 'Conceptual queries using ConQuer-II', *Proceedings of the 16th International Conference on Conceptual Modeling ER'97* (Los Angeles), D. Embley, R. Goldstein eds, Springer LNCS 1331 (Nov.) 113-126.
6. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
7. Campbell, L.J, Halpin, T.A. & Proper, H.A. 1996, 'Conceptual schemas with abstractions: making flat conceptual schemas more comprehensible', *Data and Knowledge Engineering*, vol. 20, Elsevier Science, pp. 39-85.
8. Cattell, R.G.G. (ed.) 1997, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, San Francisco.
9. Creasy, P.N. & Proper, H.A. 1996, 'A generic model for 3-dimensional conceptual modelling', *Data & Knowledge Engineering*, vol. 20, no. 2, pp. 119-62.
10. De Troyer, O. & Meersman, R. 1995, 'A logic framework for a semantics of object oriented data modeling', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 238-49.
11. Elmasri, R. & Navathe, S. 1994, *Fundamentals of Database Systems, 2nd edn*, Addison-Wesley, Menlo Park CA.
12. Falkenberg, E.D. 1993, 'DETERM: deterministic event-tuned entity-relationship modeling', *Entity-Relationship Approach – ER'93*, Springer LNCS no. 823, pp. 230-41.
13. Halpin, T.A. 1995, *Conceptual Schema and Relational Database Design, revised 2nd edn*, WytLytPub, Bellevue WA, USA.
14. Halpin, T.A. 1998, 'ORM/NIAM Object-Role Modeling', *Handbook on Information Systems Architectures*, eds P.Bernus, K. Mertins & G. Schmidt, Springer-Verlag, Berlin, pp. 81-101.
15. Halpin, T.A. 1999, 'Data modeling in ORM and UML revisited', *Proc. EMMSAD99: 4th IFIP WG8.1 Int. Workshop on evaluation of modeling methods in systems analysis and design*, Heidelberg (June).
16. Halpin, T.A. & Bloesch, A.C. 1998, 'A comparison of UML and ORM for data modeling', *Proc. EMMSAD98: 3rd IFIP WG8.1 Int. Workshop on evaluation of modeling methods in systems analysis and design*, Pisa (June).
17. Halpin, T.A. & Bloesch, A.C. 1999, 'Data modeling in UML and ORM: a comparison', *Journal of Database Management*, Idea Group Publishing, Hershey PA, pp. 4-13.
18. Halpin, T.A. & Proper, H.A. 1995, 'Subtyping and polymorphism in Object-Role Modeling', *Data and Knowledge Engineering*, vol. 15, Elsevier Science, pp. 251-81.
19. Halpin, T.A. & Proper, H. A. 1995, 'Database schema transformation and optimization', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 191-203.
20. Hammer, M. & McLeod, D. 1981, 'Database description with SDM: a semantic database model', *ACM Transactions on Database Systems*, vol. 6, pp. 351-86.
21. Meta Data Coalition 1999, *Meta Data Coalition Open Information Model*, online at www.mdcinfo.com.
22. ter Hofstede, A.H.M., Proper, H.A. & Weide, th.P. van der 1992, 'Data modeling in complex application domains', *Proc. CAiSE'92: Fourth Int. Conference on Advanced Information Systems Engineering*, ed. P. Loucopoulos, Springer LNCS, no. 593, pp. 364-77.
23. 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.
24. ter Hofstede, A.H.M., Proper, H.A. & Weide, th.P. van der 1993, 'Expressiveness in conceptual data modelling', *Data & Knowledge Engineering* 10, 1 (Feb.), pp. 65-100.
25. ter Hofstede, A.H.M.. & Weide, th.P. van der 1994, 'Fact orientation in complex object role modelling techniques', *Proc. 1st Int. Conference on Object-Role Modeling (ORM-1)*, eds T. A. Halpin & R. Meersman, Magnetic Island, Australia, July 1994, pp. 45-59.
26. ISO 1982, *Concepts and Terminology for the Conceptual Schema and the Information Base*, J. van Griethuysen ed., ISO/TC97/SC5/WG3-N695 Report, ANSI, New York.
27. Mok, W.Y. & Embley, D.W. 1996, 'Transforming conceptual models to object-oriented database designs: practicalities, properties and peculiarities', *Conceptual Modeling – ER'96*, Springer LNCS, no. 1157, pp. 309-24.
28. OMG UML Revision Task Force, UML 1.3 specification, online at www.omg.org.
29. Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.
30. Warmer, J. & Kleppe, A. 1999, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, Reading MA, USA.