2-2008

A Framework for Relationship Pattern Languages

Sudarshan Murthy
Portland State University

David Maier
Portland State University, maier@cs.pdx.edu

A Framework for Relationship Pattern Languages

Technical Report# TR-08-03. Created: May 15, 2006. Last Modified: Feb. 08, 2008

Sudarshan Murthy, David Maier
Department of Computer Science, Portland State University
PO Box 751 Portland, OR 97207 USA
{smurthy, maier}@cs.pdx.edu

Abstract. A *relationship pattern* is an abstraction of a recurring need when establishing relationships among information elements in specific contexts. By developing or leveraging a relationship pattern, modelers can solve a class of problems once and describe many relationship types at once. We have developed a framework for specifying relationship patterns and pattern languages (sets of patterns) in both modeling-language-independent and modeling-language-specific ways. We describe this framework both informally and formally. We provide examples of some commonly observed relationship patterns and show how to use them in ER with the help of a relationship pattern language called *Exemplar*. We also provide notes on using relationship patterns in UML.

1 Introduction

Consider the Entity-Relationship (ER) relationship type Assign in Figure 1(a) that represents assignment of aircrafts to routes. Assuming that flights on all routes are one hop, and that an aircraft assigned to a route must be able to fly farther than the distance of the route. Unfortunately, this constraint cannot be expressed in ER, because ER allows only key and cardinality constraints for relationships. One way to address this limitation is to extend the ER relationship-type construct to allow constraint specifications. For example, Figure 1(b) includes a predicate as a part of the relationship type Assignable. This predicate requires that an aircraft must be able to fly at least 25% farther than the distance of the route to which it is assigned.

The kind of problem demonstrated in Figure 1(a) recurs often enough to merit a common solution. Figure 1(b) uses a simple syntactic convention as a common solution. However, syntactic extensions alone do not suffice when providing a common solution; the semantics of the extension and the consequences of using the extension must also be defined. For example, consider another kind of recurring need: relating an entity's non-key attributes with other entities. Such relationships cannot be expressed in ER, because ER relationships are only among entities. One might create a convention of specifying attributes to be related as parameters of a relationship type name (similar to that in Figure 1(b)), but that convention does not clarify the meaning of such a relationship. Nor does it clarify how such relationships are instantiated.

The software design community (for example, Gamma and others [6]) has quite effectively applied the notion of *patterns* (recurring problems or needs in specific contexts) to solve recurring software-design problems. The information-modeling community has also recognized *information patterns* and provided general solutions, although the term information pattern has not been widely used. For example, the Extended ER model [3] defines a *specialization* construct to allow an entity type to *extend* another entity type. By developing a pattern, designers can solve a class of problems once, in a manner appropriate to the contexts in which the solution is required (or is applicable). By matching a problem to a pre-defined pattern, designers can leverage pre-defined solutions.

We use the term *context* (of a pattern) broadly: It can mean the space of the problems a pattern solves, it can mean a specific setting in which the pattern is noticeable, or it can mean a combination of such aspects.

In this paper, we focus on *relationship patterns*, which are abstractions of recurring needs, problems, or solutions when establishing relationships in particular contexts. We have developed a framework to specify relationship patterns and *pattern languages* (sets of related relationship patterns). The framework formally defines *pattern specification*, which describes the contexts in which a pattern applies; the semantics and constraints of the pattern; the consequences of applying the pattern; and the syntax to express relationship types of the pattern.

The contributions of this work are: A framework to define relationship patterns and pattern languages in both modeling-language-neutral and modeling-language-specific ways; the notion of *relationship typespaces* (namespaces for relationship types); and a relationship pattern language called *Exemplar* which defines some commonly observed patterns.

The rest of the paper is organized as follows. Section 2 presents an informal overview of Exemplar. Section 3 introduces the formalism for the framework. Section 4 gives an overview of related work, and Section 5 presents concluding comments and mentions some possible future work.

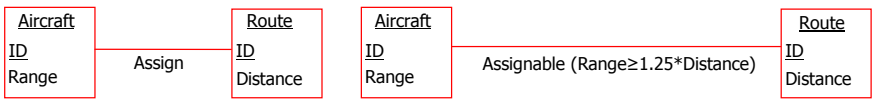

Figure 1. (a) Relationship of aircrafts and routes with only key and cardinality constraints (b) Adding a pre-condition to relationships between aircrafts and routes

2 Exemplar

The *Exemplar Pattern Language of Relationships* (*Exemplar*) is a set of some commonly encountered relationship patterns. In this section, we provide an informal overview of the patterns this language defines for the ER model. For each pattern, we also give an overview of the procedure to translate relationship types and related entity types to the relational model. Where appropriate we also describe operations on resulting relationships.

2.1 The Traditional Pattern

The Traditional pattern models relationships as defined in the ER model.

Name: Traditional.

Context: When entities need to be related.

Syntax: Relationships of this pattern are visually represented using the UML-like syntax Ramakrishnan and Gehrke [14] describe. In summary, a relationship type of this pattern is expressed as a line drawn between entity types to be related. A relationship-type name is a string of the form <type>, written at or about the center of the line. A role name is a string of the form <role>, written adjacent to the entity playing the role. Cardinality constraints are written adjacent to each participating entity, on or around the relationship line.

Constraints: Entities of any type may be related, and there is no limit on the degree of a relationship type. Relationships can have attributes. There are no constraints on cardinality constraints.

Semantics: The entities are *somehow* related. The user (the person instantiating relationships) provides the rationale for relationships.

Consequences: Any set of entities can be related based on key attributes. Key values of related entities and values of relationship attributes are stored. Relationship type and related entity types are converted to the relational model using the procedure Elmasri and Navathe have defined [3] (called the *traditional* procedure in the rest of this paper; we only discuss conversion to the relational model).

Example: The Assign relationship type in Figure 1(a).

2.2 The Predicated Pattern

In the rest of this section, we omit describing aspects of patterns that are similar to the Traditional pattern.

Name: Predicated.

Context: When a precondition applies to entities to be related.

Syntax: <type> (<predicate>), where <predicate> is a Boolean expression.

Constraints: Same as the Traditional pattern.

Semantics: Entities are related only if the predicate is satisfied.

Consequences: Participation constraint on any of the related entity types can force a pattern on entities. Obviously, if an entity must participate in at least one relationship, it must also help satisfy the precondition for the relationship.

Example: The Assignable relationship type in Figure 1(b).

2.3 The Computed Pattern

In the Traditional pattern, all relationships are stored. However, there are cases when relationships do not need to be stored, instead they can be computed. The Computed pattern models relationships that can be computed.

Name: Computed.

Context: When relationships can be computed.

Syntax: Computed:<type> (<predicate>). The string Computed: is fixed. It denotes that relationships are computed, not stored. The component <predicate> is a Boolean expression.

Constraints: Relationship attributes, if any, must also be computable. Also, cardinality constraints are not constraints at all; they are consequences: The number of permutations of key values found in the computed relationship set depends on the values of key attributes of related entities that match the predicate. Thus, we recommend that cardinality constraints be unspecified for relationship types of this pattern.

Semantics: This pattern models the relational *inner join* operation [3]. Entities are *automatically* related if the predicate is satisfied.

Consequences: Relationships are *not* stored. Conversion to the relational model generates a view definition, not a definition for a stored relation. Relationships *cannot* be deleted.

Example: The Assignable relationship type in Figure 1(b) with the type name changed to read as follows: Computed:Assignable (Range≥1.25 * Distance)

2.4 The Grouped Pattern

Consider scheduling tutorials for a 3-day conference: A tutorial is 1-day long, is offered only once, and is lead by 1 or 2 presenters. A presenter may lead at most one tutorial per day, but a presenter is not required to lead any tutorial. Figure 2(a) shows an attempt to model this information as a traditional relationship type. Unfortunately, this schema restricts a presenter to at most one tutorial over the duration of the conference, not at most one tutorial each day. Changing the cardinality constraint 0..1 to 0..* allows a presenter to present many tutorials over the duration of the conference, but it does not restrict the presenter to one tutorial a day.

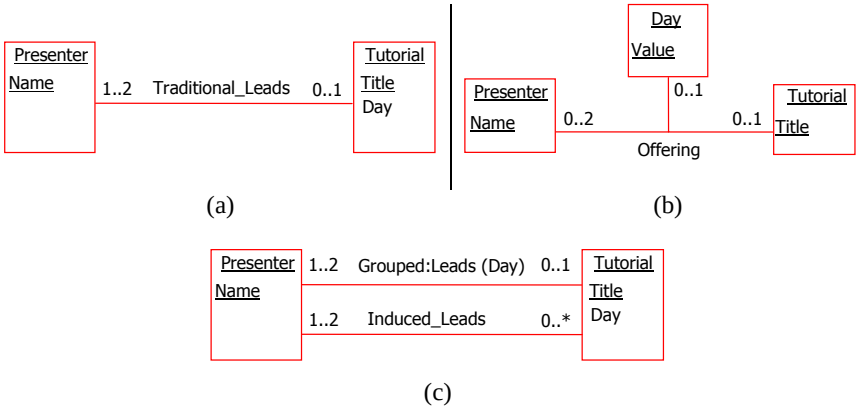

Figure 2. (a) A Traditional relationship that cannot model the tutorial information (b) A Traditional ternary relationship for the tutorial information (c) A Grouped relationship for the tutorial information, and the Traditional relationship it induces

The tutorial information can be modeled using the Traditional ternary relationship type Offering shown in Figure 2(b), (attribute Day is promoted to an entity so it can participate in relationships), but ternary relationships are hard to comprehend, and converting them to the relational model is not straightforward. For example, the relationship type Offering has two candidate keys in the relational model: Presenter.Name + Day.Value and Presenter.Name + Tutorial.Title (Ferg's M:1:1 scenario [4]). The former key restricts a presenter to one tutorial a day, but it does not limit a tutorial to one day, and a tutorial might not be scheduled at all. The latter key additionally does not limit a presenter to one tutorial a day. That is, the modeler must carefully choose the key for the relationship table.

The Grouped pattern provides an alternative way to model tutorial information using only a binary relationship. The pattern is based on the observation that enforcing cardinality constraints in a Traditional relationship amounts to enforcing the cardinality of groups formed by keys (or *exactly* the set of key attributes) of the entity whose cardinality is constrained. The Grouped pattern allows cardinality constraints to be expressed against a set of *any* attributes such that the relationship degree does not change. In addition to providing a procedure to automatically generate a relational schema, it also defines a procedure to assist in enforcing cardinality constraints.

Name: Grouped.

Context: When cardinality constraints are to be expressed with any set of attributes, not just keys.

Syntax: Grouped:<type> (<attributes>). The string Grouped: is fixed; <attributes> is a non-empty set of attributes to be related to other entities.

Constraints: The attributes specified must be from one entity (called the *grouped entity*). We recommend the set of attributes *not* be exactly the set of key attributes of the grouped entity; the Traditional pattern should be used in that case.

Semantics: The cardinality constraints have Look-across semantics [4]. The attributes specified partition relationships, and cardinality constraints are enforced on the partitions. Consequently, the attributes specified are called *grouping attributes*, and their cardinality is called *group cardinality*.

Consequences: A Grouped relationship induces a Traditional relationship if the grouping attributes are not exactly the set of key attributes. In this case, if lower:upper is the cardinality constraint on the grouping attributes in the Grouped relationship, then lower:* is the constraint on the grouped entity in the induced Traditional relationship. The cardinality constraint of the grouping attributes and that of the induced Traditional relationship are enforced. Only the induced Traditional relationships are stored. Consequently, only the induced relationship type is converted to the relational model. If the grouping attributes are exactly the set of key attributes, the Grouped relationship is treated as a Traditional relationship.

Example: The relationship type Grouped:Leads in Figure 2(c) models the tutorial information. Entity Tutorial is the grouped entity because the grouping is on one of its attributes, Day. This relationship type induces the Traditional relationship type Induced_Leads, with the cardinality constraint on the grouped entity Tutorial being 0..*. Clearly, if a presenter leads 0 or 1 tutorial each day, he leads 0 or more tutorials over the duration of the conference. Figure 3 shows an instance of the tutorial information schema in the relational model. The schema of the relationship table matches the induced Traditional relationship type.

Table 1. Comparison of how the relational schemas for the example Grouped and the Traditional ternary relationship types enforce the constraints of scheduling tutorials

Constraint	Grouped	Ternary
A tutorial is scheduled	**Automatic** (non-NULL Day)	Post-processing
A tutorial is scheduled once	**Automatic** (key constraint)	Post-processing
A tutorial has a presenter	Post-processing	Post-processing
A tutorial has no more than two presenters	Precondition	Precondition
A presenter does not have to lead on a day	Automatic (optional participation)	Automatic (optional participation)
A presenter leads no more than one tutorial a day	Precondition	**Automatic** (key constraint)

Table 1 compares how the relational schemas for the Grouped relationship and the Traditional ternary relationship enforce the various constraints of the tutorial information. For the ternary relationship, we assume the key Presenter.Name + Day.Value (the better choice). An entry 'Automatic' in a cell means the constraint is automatically enforced; the reason is mentioned in parentheses. An entry 'Precondition' means that enforcing a precondition at the time a relationship is instantiated (or altered) can help meet the constraint; an entry 'Post-

processing' means the constraint can be satisfied by some form of post-processing after a relationship is instantiated (or altered). Entries are shown in bold when one approach has an edge over the other.

In the Grouped pattern, group cardinality constraints can be enforced by computing the cardinalities of groups that involve the grouping attributes and the key attributes of the non-grouping entities. Figure 3 includes an SQL query to compute group cardinalities for the example relationship type. It also shows the result for that query over the example instance. In this instance, presenter `LMD` can be assigned only to a tutorial on Day 2, because the group count for that presenter is already `1` for Days 1 and 3. However, presenter `DEM` can lead a tutorial on Days 1 and 3.

In general, the cardinality of a group for any Grouped relationship type can be computed using the following SQL query:

```
SELECT <non-group key attrs>, <grouping-attrs>, COUNT(*)
FROM <relationship-type> JOIN <group-entity>
GROUP BY <non-group key attrs>, <grouping-attrs>
```

The `JOIN` clause joins the key attributes of the tables corresponding to the relationship type and the grouped entity; `<relationship-type>` indicates the table for the Grouped relationship; `<group-entity>` is the table for the grouped entity; `<grouping-attrs>` indicates the grouping attributes of the Grouped relationship; and `<non-group key attrs>` indicates the key attributes of the non-grouped entities. The join operation can be omitted if the grouping attributes are a subset of the grouping entity's key attributes, because the relationship table would already contain those attributes. That is, the `FROM` clause can contain only the relationship table.

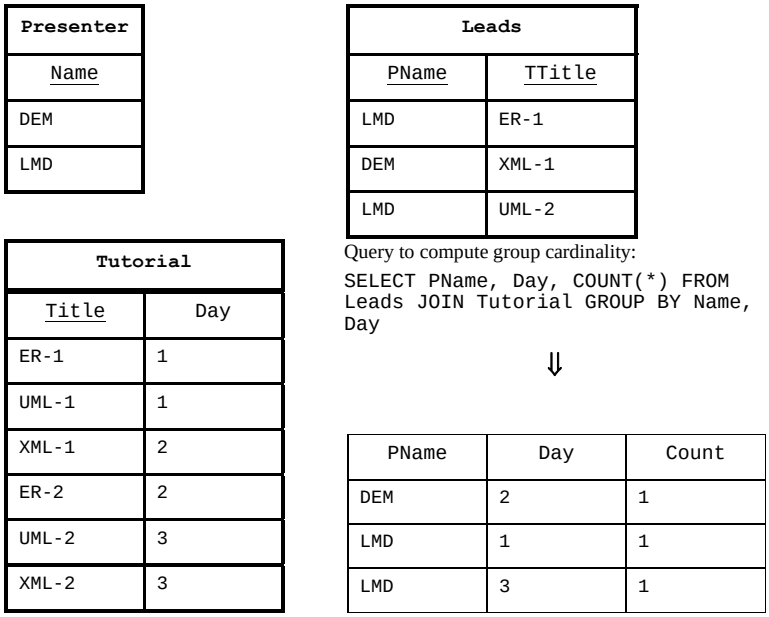

Figure 3. An instance of the tutorial information schema in the relational model, and the result of the SQL query to compute group cardinality

2.5 The History Pattern

The History pattern augments a relationship type of any degree with the ability to capture history of relationships, without increasing the degree of the relationship type. In addition to defining a procedure to automatically generate a relational schema, its specification also defines operations to retrieve the current state of a relationship and the history of a relationship. It also defines operations to update a relationship.

Name: History.

Context: When a record of creation, changes, and deletion of relationships is to be maintained.

Syntax: History:<type>. The string History: is fixed.

Constraints: Same as the Traditional pattern.

Semantics: Same as the Traditional pattern, except a record of *all* updates (including deletes) to an instance is maintained. Timestamps or sequence numbers are used to map the history of a relationship (some applications might require a timestamp).

Consequences: Store a copy of *each state* of a relationship with corresponding timestamp or sequence number. Conversion to the relational model is the same as Traditional pattern, except a relationship table is always created, and a timestamp (or sequence number) attribute is added to the key of that table. An `Action` attribute is also added to the table to record the actual action (insert, change, or delete) performed on the relationship. This attribute can be omitted in some scenarios (when the action performed can be inferred), but we do not discuss those scenarios.

Example: Consider the Traditional relationship type WorksOn in Figure 4(a). Position is a relationship attribute. This relationship type allows an employee to be assigned to any number of projects, but it does not track the history of changes to an employee's association with a single project. For example, assume a relationship that associates Employee `3` with Project `12` as a manager. When the employee is no longer associated with that project, the relationship is deleted. Similarly, if the employee's position on that project changes, the relationship is *altered* to show his new position. In both cases, the record of assignment of the employee as a manager of the project is lost.

Figure 4(b) shows an attempt to model history of relationships as a Traditional ternary relationship type. Position is a relationship attribute. This relationship type allows an employee to be assigned to a project in more than one capacity at the same time. For example, Employee `3` can be assigned a manager of Project `12` as a manager on `May 1st 2006`; and as a consultant on `May 2nd 2006`. The second operation does *not* update the relationship the first one creates; it creates another relationship.

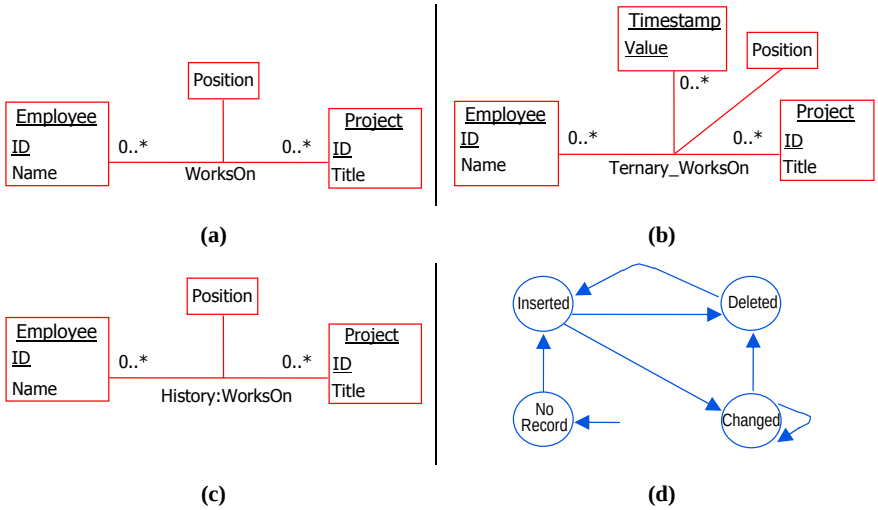

Figure 4. (a) A Traditional binary relationship to model assignment of employees to projects (b) A Traditional ternary relationship that cannot model history of assignment of employees to projects (c) A binary relationship type in the History pattern to model history of assignment of employees to projects (d) States and state-transitions of a History relationship of M:N cardinality with optional participation

No combination of existing ER model constructs can model history of relationships, because the ER model does not have constructs to track changes to (and deletion of) a relationship. The History pattern offers a solution. Figure 4(c) shows a binary relationship type of this pattern to model history of employee assignments to projects. Figure 5 shows an instance of this schema in the relational model. The table `History_WorksOn` captures the history of employee assignments. We use a sequence number attribute `TS` to track changes to relationships. This instance shows the history of relationship of Employee `3` with two projects. At time `1` the employee is assigned as a manager to Project `12`, and is disassociated from that project at time sequence `2`. At time sequence `3`, he is assigned as a consultant to Project `19`, but changed to a manager of that project at time sequence `4`.

The history of a relationship can be modeled using a state diagram. Figure 4(d) shows the state diagram for a History relationship type of cardinality M:N (with optional participation). The No Record state is the start state. For example, for the instance shown in Figure 5, the relationship of Employee `3` with Project `12` is in the No Record state before time `1`; at time `1` the relationship transitions to the Inserted state; at time `2` the relationship transitions

to the Deleted state. Later at time 5 the relationship transitions back to the Inserted state because the relationship is reestablished.

In the relational model, the No Record state of a relationship is indicated by the absence of a corresponding tuple in the relationship table. A tuple is created for a relationship when the relationship first enters the Inserted state. The tuple corresponding to a relationship is *not* deleted when the relationship enters the Deleted state. Updates alter the value of attribute Action to denote the current state of a relationship.

Operations: Many of the operations associated with temporal datasets [8] apply to relationships of History pattern (which we outline in the context of the relational model). The operation to *retrieve* a relationship works on the set of tuples concerning the employee and project in question. It returns the tuple with the highest sequence number, provided the Action attribute of that tuple does not contain the value 'Deleted'. No *current* relationship exists if the tuple with the highest sequence number is marked deleted, or if no tuple exists for the employee and project combination. The operation to retrieve the history of a relationship returns all tuples for that relationship in order of sequence number or timestamp.

An *update* operation (insert, change, or delete) *always* inserts a new tuple and writes an appropriate value in attribute Action, and the attribute TS is automatically assigned a *transaction time* or a timestamp. The change and delete operations cause an error if no current relationship exists. The insert operation causes an error if a current relationship does exist. The change and delete operations first clone the current relationship. The change operation then updates the requested attributes; the delete operation marks the relationship as deleted.

The change operation considers changes to key attributes of participating entities specially to address possible conflicts, and to preserve the update semantics of ER and the relational models. (Neither model prevents changes to key attributes, but both models enforce key constraints.) For example, consider the relationship of Employee 3 with Project 12. If at time 7 the employee ID is changed to 4, the current relationship is marked deleted, and a new relationship is inserted for Employee 4 with Project 12 because no current relationship exists between Employee 4 and Project 12 (relationship attributes are copied from the deleted relationship). However, if at time 7 the project ID is changed to 19 for the same relationship (Employee 3, Project 12), the current relationship is marked deleted, and the existing relationship between Employee 3 and Project 19 is marked changed.

Operations such as *vacuuming* [8] that apply to temporal datasets also apply to the History pattern, but we do not describe them in this paper.

Employee

ID	Name
3	DEM
5	LMD

Project

ID	Title
12	FP
19	SW

History_WorksOn

TS	EID	PID	Position	Action
1	3	12	Manager	Inserted
2	3	12	Manager	Deleted
3	3	19	Consultant	Inserted
4	3	19	Manager	Changed
5	3	12	Manager	Inserted
6	3	12	Consultant	Changed

Figure 5. An instance of the employee-project assignment schema in the relational model

2.6 Exemplar Relationship Patterns in UML

The Exemplar relationship patterns can be employed in the Unified Modeling Language (UML) as well, but the syntax of relationship types may be different than that for the ER model. Table 2 provides an overview of UML constructs helpful in representing the Exemplar patterns.

No UML construct readily conveys the History pattern. However, UML does provide two constructs useful in representing patterns: parameterized collaborations and stereotypes. A *parameterized collaboration* can be viewed as a template for creating parts of a schema. A *stereotype* is a means of providing a custom interpretation to a model construct. For example, a stereotype association class called History can be used to represent the History pattern. With that stereotype defined, the relationship type History:WorksOn in Figure 4(a) could be shown as the association class «History» WorksOn in UML. Note that UML does not define the stereotype History; a modeler defines it.

Though UML may provide the syntax (and possibly the semantics) for a pattern, modelers still need to define other aspects such as contexts for the pattern and consequences of using the pattern. For example, the syntax and semantics of constraints on UML associations suffice for the Predicated pattern, but the context for its ap-

plication and the consequences of its applications (such as procedure to convert to the relational model) still need to be defined. *Decorative* stereotypes [2] (stereotypes that just introduce new syntax) rarely suffice when using stereotypes to express patterns: One might need to define descriptive stereotypes (stereotypes that help in expressing more information), and sometimes need to define restrictive stereotypes (stereotypes that introduce new semantics).

Table 2. UML constructs helpful in using Exemplar patterns.

Pattern	UML Construct	Remarks
Predicated	Constraint, OCL precondition	Useful as is
Computed	Derived Association	Semantics of multiplicity of a derived association is unclear
Grouped	Ternary association or OCL invariant	Ternary associations can be hard to comprehend
History	—	No native construct

3 The Framework

We have defined a framework in which relationship patterns and pattern languages can be expressed for use in different modeling languages. The framework establishes the requirements for a pattern, and includes a template for describing patterns.

A *relationship pattern specification* is a description of a relationship pattern. A pattern specification has two aspects: modeling-language neutral and modeling-language specific. The modeling-language-neutral aspect sets up expectations for the pattern in all modeling languages in which the pattern might be used. For example, the expectation for the History pattern is to maintain a history of relationships, regardless of the modeling language. The modeling-language-specific aspect describes the pattern as it applies to each modeling language in which the pattern is used.

Formally, a relationship pattern specification $\mathcal{P}$ is a set of specification parts $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n$, where part $\mathcal{P}_0$ specifies the modeling-language-neutral aspect, and parts $\mathcal{P}_1$ to $\mathcal{P}_n$ specify the modeling-language-specific aspect (one part per modeling language). A pattern specification must include at least the modeling-language-neutral part ($n \geq 0$). It is possible that a specification initially includes no modeling-language-specific part, or it might include parts for only a small number of languages. Specification parts for other modeling languages may be added later as need arises.

A *specification part* is a set of six *sub-parts*: $\{C, S, T, M, N, E\}$, where C describes the contexts in which the pattern can be used; S specifies the syntax; T specifies the constraints; M describes the semantics, N describes the consequences of using the pattern; and E is a set of illustrative example uses of the pattern. The sub-parts may be specified either formally or informally. For example, context may be specified informally, whereas syntax may be specified formally. Section 2 informally describes the specification sub-parts for the ER model.

Some of the sub-parts of a modeling-language-neutral part can be left unspecified, but all sub-parts of a modeling-language-specific part must be defined. For example, there may be no modeling-language-neutral syntax specification, but each modeling-language-specific part must specify a syntax sub-part.

Parts can share sub-part specifications. For example, parts can share the context sub-part specification. We do not define a particular mechanism to reference sub-part specifications for sharing, but we expect that mechanisms (such as section numbers) commonly used in technical writing suffice in many cases.

Modeling-language-specific part specifications should be consistent with the modeling-language-neutral part specification, and corresponding sub-part specifications should be consistent across modeling-language-specific specifications. For example, the context sub-part specification for each modeling language should be consistent with the modeling-language-neutral context sub-part specification.

A pattern specification must include at least one example use of the pattern. Following the *rule of three* for patterns [9] (to demonstrate a recurring need), we recommend a pattern specification also cite three distinct cases where the pattern could have been used, or cite three genuine uses of the pattern.

3.1 Relationship Signatures

We use the term *relationship signature* to mean the specification of the syntax sub-part. A well-designed signature can clearly and concisely express the intent of a pattern.

Depending on the modeling language, a relationship signature may have more than one component. For example, the ER and UML models involve a drawing syntax and text syntax. The text syntax for the ER model includes specifications for relationship type names, role names, and cardinality constraints. We recommend pattern developers define all components of a relationship signature both informally and formally (where possible).

Relationship signatures should be consistent with the syntax of modeling languages, and existing language constructs should be reused where possible. For example, the built-in constraint construct should be used in UML for the Predicated pattern.

3.1.1 Example Relationship Signatures

Section 2 informally specifies signatures for ER relationship patterns in the Exemplar pattern language (under the heading 'Syntax'). Formally, names of Traditional ER relationship types can be specified using the regular expression `c*`, where `c` is a printable character. However, formally specifying names of relationship types of the Predicated, Computed, and Grouped patterns requires a context-free grammar [7] due to the use of balanced parentheses.

```
<type-sign> :: <typespace><typename><params>
<typespace> :: ε | <ncpchar><eorncp><colon>
<eorncp>    :: ε | <ncpchar><eorncp>
<typename>  :: ε | <ncpchar><type>
<params>    :: ε | (<fparams>) | {<fparams>} | 03
<fparams>   :: <chars> | (<fparams>) | {<fparams>} |
               [<fparams>] | <fparams><fparams>
<chars>     :: ε | <npchar><chars>
<colon>     :: << The character : >>
<npchar>    :: << not a bracket char ( ) { } [ ] >>
<ncpchar>   :: << not colon or a bracket char >>
```

Figure 6. A context-free grammar to generate the name of a relationship type for the ER model. Non-terminal symbols are shown in angle brackets; double-angle brackets contain informal definitions. Characters in bold are literals to be used as shown

Figure 6 shows a context-free grammar that defines a super language which can generate the name of any relationship type for Exemplar patterns, for the ER model. We call this language a *super language* because the language of type names for some patterns may be a sub-language of this language. For example, the signature for the Exemplar Predicated pattern restricts the non-terminal symbol `<params>` to allow only Boolean expressions, but the grammar in Figure 6 does not impose that restriction.

The non-terminal symbol `<typespace>` in this grammar denotes a *relationship typespace*, which is a namespace for relationship types. It defines the semantics of the relationships of the included types. It can also define operations that apply to those relationships. That is, a relationship typespace can be viewed as a class from which its types inherit semantics and operations.

Relationship typespaces are useful in classifying and interpreting relationship types. The use of typespaces is not required, and we assume a *global* typespace when no typespace is specified for a relationship type. The Traditional pattern does not use typespaces; the Computed, Grouped, and History patterns of Exemplar use specific typespaces. For example, the History pattern uses the typespace named 'History'.

3.2 Example Pattern Specification: The History Pattern

P_0: Modeling-Language-Neutral Part

Name: History.

Context: When a record of creation, changes, and deletion of relationships is to be maintained.

Syntax: Not defined, but we recommend using the relationship typespace name 'History'.

Constraints: Same as the Traditional pattern.

Semantics: Same as the Traditional pattern, except a record of *all* updates (including deletes) to an instance is maintained. Timestamps or sequence numbers are used to map the history of a relationship.

Consequences: *Effectively* stores a copy of each state of a relationship with corresponding timestamp or sequence number. Operations to retrieve the current state of a relationship and the complete or partial history of a relationship are defined.

P_1: Modeling-Language-Specific Part (ER)

Context: When a record of creation, changes, and deletion of relationships is to be maintained.

Syntax: Formally, the type signature is a regular expression of the form History:cc*, where the string History: is fixed, and c is a printable character.

Constraints: Same as the modeling-language neutral part.

Semantics: Same as the modeling-language neutral part.

Consequences: See paragraph with this heading in Section 2.4.

Example: See Figure 4(b) and associated description.

P_2: Modeling-Language-Specific Part (UML)

Context: When a record of creation, changes, and deletion of relationships is to be maintained.

Syntax: Formally, the signature type name is a regular expression of the form «History»cc*, where the string «History» is fixed, and c is a printable character.

Constraints: Same as the modeling-language neutral part.

Semantics: Same as the modeling-language neutral part. A UML stereotype named is History defined (but not shown in this paper).

Consequences: See paragraph with this heading in Section 2.4.

Example: We omit a complete UML example, but see Figure 4(b).

3.3 A Conceptual Model for Relationship Patterns and Pattern Languages

Figure 7 shows a conceptual model for relationship patterns and pattern languages. Although we use ER terminology in this figure, the conceptual model applies to any modeling language that allows specification of relationships, with appropriate substitution of terminology. For example, the conceptual model applies to UML if the terms *entity* and *relationship* are substituted by the terms *class* and *association* respectively.

The conceptual model provides an overview of our framework. From the model we see that a relationship pattern can relate to zero or more patterns. For example, the Computed pattern is related to the Predicated pattern. A relationship type *always* follows a pattern, even if it is just the traditional pattern in some language (and possibly belongs to the global typespace). The relationship type Illustrates represents the examples sub-part ($\mathcal{E}$) of a pattern specification.

4 Related Work

The History pattern in Exemplar is related to Fowler's *Historic Mapping* pattern [5] which allows one or more temporal attributes such as "valid time" and "knowledge time" to be associated with a relationship, but Fowler's pattern does not track history of updates to relationships. The History pattern associates a *transaction time* with each update to a relationship; a modeler can associate other temporal attributes, if necessary.

Noble [13] has described relationship patterns that correspond to associations, aggregation (both supported in ER), and inheritance (supported in Extended ER). The relationship patterns defined in Exemplar operate at a higher level of abstraction than the patterns Noble describes.

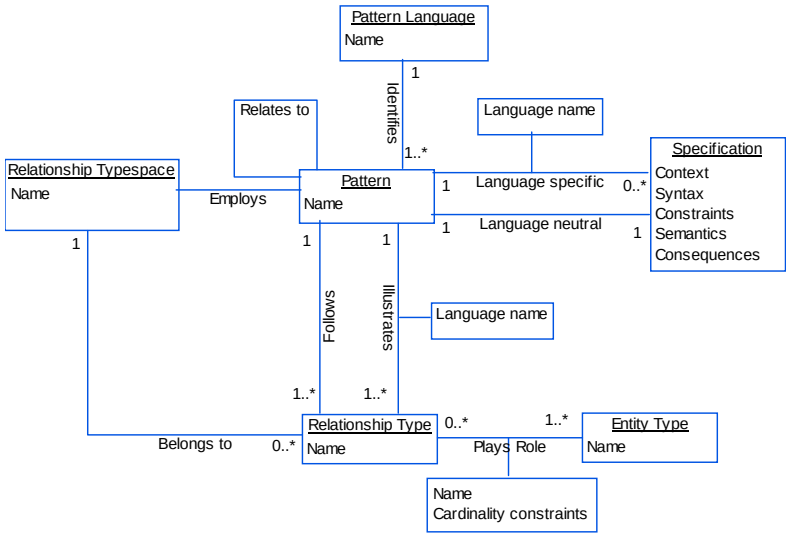

Figure 7. A conceptual model for relationship patterns and pattern languages

Relationship typespaces are namespaces, but they are also different from namespace systems such as XML namespaces [12]. A namespace typically denotes a set of pre-defined names, and only the namespace definer is expected to add new names to a namespace. In contrast, the definer of a relationship typespace may not include any relationship types in that typespace; others can add relationship types to that typespace.

Alexander [1] uses the term *patterns of relationships* in architecture (of cities and buildings) to indicate the relationship of various parts of a building. For example, in the context of a house, a door *connects* the interior of the house with its exterior. Alexander uses the term *pattern language* to mean the set of all patterns one uses to realize a *complete* building or a community. However, we use the term *relationship pattern language* to mean only a set of related relationship patterns, because modeling particular information involves more than just relationship patterns. Gamma and others [6] express a similar view with respect to software design patterns: Design patterns are not guaranteed to generate complete programs.

The software design community has used several means to describe software design patterns. For example, the *Portland Form* [11] prescribes a narrative, essay-like form. Our *pattern specification* is similar to the outline-styled *pattern template* Gamma and others [6] prescribe. However, like Fowler, we believe that a single template or form cannot help describe all patterns. Our framework identifies the components of a minimal pattern specification, and recommends that the components be consistent. Pattern developers can use the examples in Sections 2 and 3 as a starting point to describe new relationship patterns.

5 Summary

Relationship patterns express solutions for common conceptual-design problems. They can assist in customizing actions performed on conceptual relationships. Our framework provides a means to specify both relationship patterns and pattern languages. It also defines a relationship pattern language called Exemplar to describe some common patterns observed. We have also used the framework to define a relationship pattern language (not described in this paper) to represent references to information fragments [10]. The strength of our framework is that it allows patterns to be specified in both modeling-language-neutral and modeling-language-specific ways.

In this paper we have described individual relationship patterns, but patterns can also be composed. For example, the History pattern can be combined with the Predicated pattern. Pattern composition is a possible area of future work. We have thus far specified relationship patterns for ER and UML; and indirectly (via ER and UML) for the relational and XML models. However, it might be beneficial to specify some patterns directly for the relational and XML models. The framework currently does not define a means to distinguish relationship

typespaces (whereas XML namespaces are distinguished using URIs). This and other aspects of managing relationship typespaces is another possible area of future work.

References

[1] Alexander, C. 1979. *The Timeless Way of Building*. Oxford University Press.

[2] Berner, S., Glinz, M., Joos, S. 1999. A Classification of Stereotypes for Object-Oriented Modeling Languages. Paper read at Second International Conference on the Unified Modeling Language, Colorado, USA.

[3] Elmasri, R., Navathe, S.B. 2003. *Fundamentals of Database Systems*. 4th edition. Addison-Wesley.

[4] Ferg, S. 1991. Cardinality Constraints in Entity-Relationship Modeling. Paper read at 10th International Conference on Entity-Relationship Approach (ER'91), October 23-25, San Mateo, California.

[5] Fowler, M. 1996. Analysis Patterns: Reusable Object Models. Addison-Wesley.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7] Hopcroft, J. E., Motwani, R., Ullman, J.D. 2001. *Introduction to Automata Theory, Languages, and Computation*. 2nd edition. Addison-Wesley.

[8] Jensen, C. S. 2000. Temporal Database Management.

[9] Lea, D. 2000. Patterns-Discussion FAQ. Available from http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html. Accessed.

[10] Murthy, S., Delcambre, L., Maier, D. 2006. Explicitly Representing Superimposed Information in a Conceptual Model. In *Proceedings of 25th International Conference on Conceptual Modeling (ER 2006)*, Nov. 6-9, Tucson, Arizona.

[11] Murthy, U., Ahuja, K., Murthy, S., Fox, E.A. 2006. SIMPEL: A Superimposed Multimedia Presentation Editor and Player. In *Proceedings of 6th ACM/IEEE-CS Joint Conference on Digital Libraries*, Jun. 11-15, Chapel Hill, NC.

[12] Namespaces in XML. 1999. W3C. Available from http://www.w3.org/TR/REC-xml-names/. Accessed.

[13] Noble, J. 2000. Basic Relationship Patterns. In *Pattern Languages of Program Design 4*, edited by Harrison, N., Foote, B., Rohnert, H. Addison-Wesley.

[14] Ramakrishnan, R., Gehrke, J. 2003. *Database Management Systems*. Third edition. McGraw Hill.