A Framework for Relationship Pattern Languages

Sudarshan Murthy, David Maier

Department of Computer Science, Portland State University

http://sparce.cs.pdx.edu

mailto:smurthy@cs.pdx.edu

Background

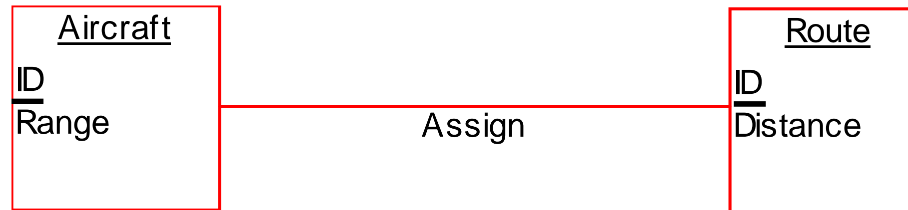
- A conceptual schema provides the basis for understanding entities and relationships; it is typically described in a diagram
- Most modeling techniques provide simple means of indicating relationships
  - Commonly as a line drawn between related entities (ER also uses a diamond)
- We might give the relationship type a name, and indicate cardinality and other constraints (such as *{Ordered}* in UML)

Common Uses for a Conceptual Schema

- A means to convey ideas and requirements
  - E.g., designers often use ER diagrams and UML diagrams to communicate ideas to developers
- A basis for activities later in the information lifecycle
  - *E.g.*, an ER schema is translated to a relational schema

* Assume non-stop flights; source: www.airliners.net, www.webflyer.com

A Problem



- Relationship specifications are at times unclear or incomplete
  - Can reduce comprehension, introduce defects, and cause implementation inefficiencies
- What is 'Assign' really about?
  - What is the basis for this relationship?
  - Assign a DC-8 (range 3748-5996 miles) to the LAX-AKL route (6510 miles)? *

Other Problems

- The ER model has limited expression for relationships
  - Type names, role names, cardinality constraints
- Treats all relationships the same way, regardless of the context of the relationships
  - *E.g.*, Assumes *any* set of instances can be related (only cardinality constraints apply)
- Typically, the same procedure is used to generate logical schemas for all relationship types (problem not due to the ER model)

Observations

- There are 'patterns' of relationships
  - A pattern is a form or model proposed for imitation*
  - Relationship patterns can impose constraints other than cardinality constraints, and they can have consequences
- ER supports *one* relationship pattern: entities related based on key attributes, with cardinality constraints

Some Relationship Patterns

- Non-key or partial key attributes of entities might form the basis of a relationship
- Pre-conditions, post-conditions, and invariants might apply to relationships
- Some relationships can be computed (traditionally, relationships are stored)
- Attributes of an entity might need to be related

Goals

- Improve expression of relationships
  - Express constraints, semantics, and consequences, at both schema and instance levels
- Express patterns of relationships
- We confine our work to the ER model
  - Models such as UML do not have some of the limitations we address
  - Some of the limitations we address do apply to other models
  - The concept of patterns applies to any model

The Proposal

- Use a ‘framework’ that allows expression of both relationship patterns and one-off relationship types
- We describe the framework, its components, and show example applications
  - Exemplar: Exemplar Pattern Language of Relationships

Outline

- The problem and proposed solution
- Relationship patterns and pattern languages
- Exemplar: a relationship pattern language
- The framework
  - Typespaces
  - Signatures
- Related work
- Summary

* Christopher Alexander. The Timeless Way of Building.

Relationship Patterns

- A *relationship pattern* is an abstraction of *recurring* needs or problems when establishing relationships in a context; it can also be a *suggested* solution to the problems identified
- A relationship pattern is similar to a software pattern, except it is focused on relationships
- Like software patterns, inspired by the notion of patterns in architecture*

Why Use Relationship Patterns?

- Solve a kind of problems once
- Describe many relationship types at once
- Understand many relationship types at once
- Customize
  - Define how relationships are treated in various stages of the information life cycle
- Leverage known patterns
  - Following a pattern well-understood can ensure consistency and increase acceptance

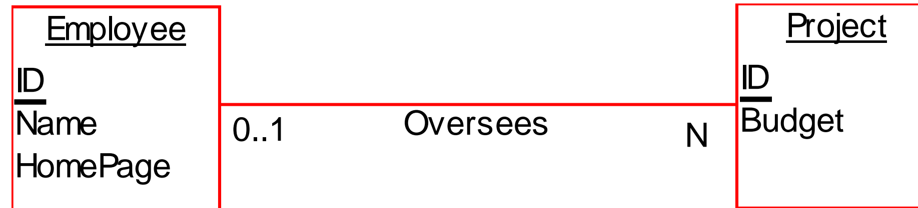
* Riehle, Züllighoven. Understanding and Using Patterns in Software Development. 1996.

Relationship Pattern Languages

- According to Christopher Alexander, a *pattern language* is a set of related patterns
- Riehle and Züllighoven* use the term *Pattern Set*
- Gamma and others (Gang of Four) use the term *Pattern Catalogue*

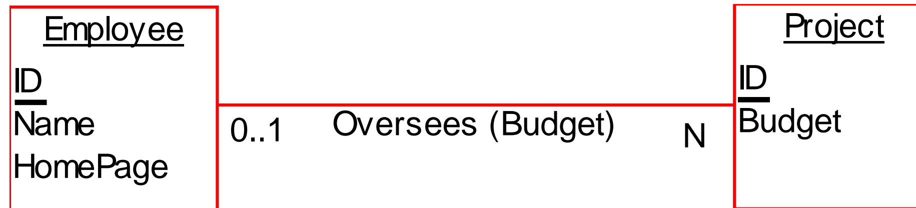
* Terms adapted from cognitive psychology. See works of Gibson, Norman, and others

Affordances and Perceptions*



"An employee oversees projects"

Notation Enrichment



"An employee oversees project budgets, not projects"

- Enriching the notation altered affordance, hence altered the perception
  - We enriched the relationship type name here, but other aspects of ER notation might also be enriched
- We enrich the ER notation to express relationship patterns

Exemplar: Exemplar Pattern Language of Relationships

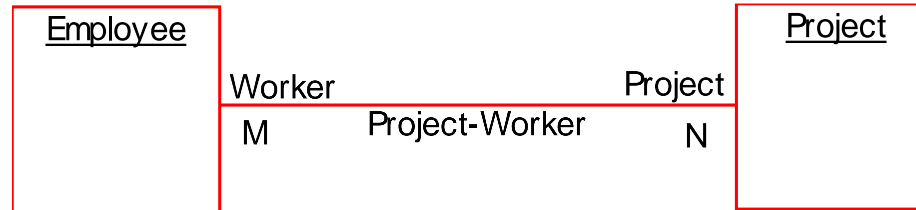
* Ferg. Cardinality Constraints in Entity-Relationship Modeling. ER 91.

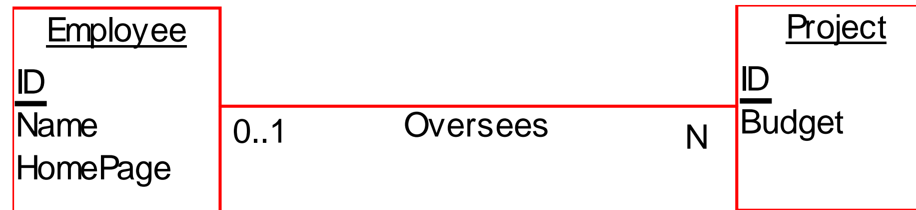
The *Traditional* Pattern: Context, Syntax

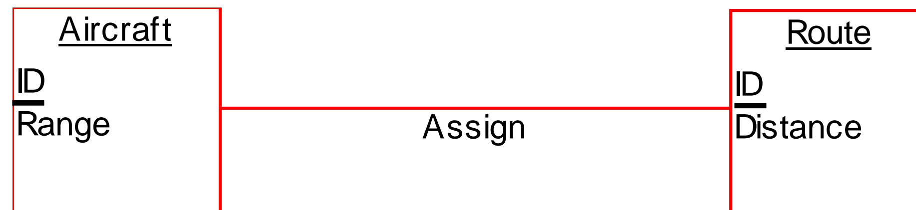
- Context
  - Entities need to be related
- Syntax
  - Draw a line between related entities
  - Relationship name is a string of the form `<type>`, written at/about the center of the line
  - Role name is a string of the form `<role>`, written adjacent to the entity playing the role
  - Cardinality constraints are specified as a number or a range of numbers written adjacent to an entity (*Look-Across* cardinality*)

* First example due to P. Chen

The Traditional Pattern: Examples*







The Traditional Pattern: Constraints, Semantics

- Constraints
  - Entity type: Entities of any type may be related
  - Degree: Any number of entity types may be related
  - Cardinality: 1-1, 1-many, many-many; participation can be optional
- Semantics
  - Instances of entities are *somehow* related

* Elmasri, Navathe. Fundamentals of Database Systems, 4th Ed.; we mention conversion to only one model

The Traditional Pattern: Consequences

- Any set of entity instances can be related based on key attributes, *without* regard to values of other attributes
- Storage: Key values of related entity instances and values of relationship attributes are stored
- Conversion: Relationship type and entity types converted to relational model using the Elmasri-Navathe procedure*

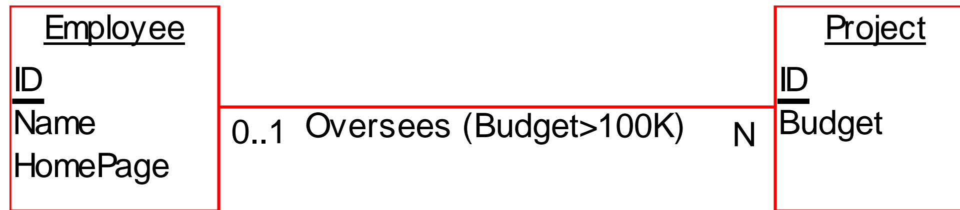
The *Predicated* Pattern: Context, Syntax

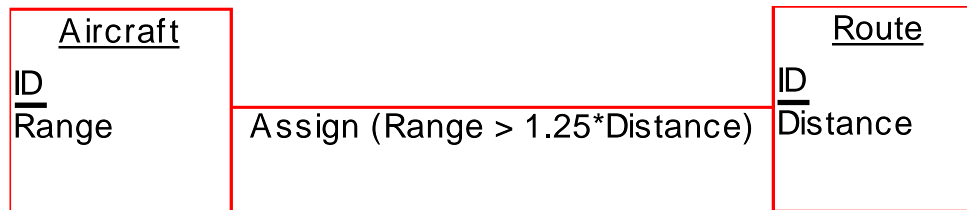
- Context
  - A relationship has preconditions
- Syntax
  - Type name is a string of the form

```
<type>(<predicate>)
```

  - `<predicate>` is a Boolean expression

The Predicated Pattern: Examples





* If an entity must participate in at least one relationship, it must help satisfy the predicate

The Predicated Pattern: Constraints, Semantics, Consequences

- Constraints
  - Same as Traditional pattern
- Semantics
  - A relationship is possible only when the specified condition is met
  - Same as Traditional pattern if predicate is `true`
- Consequences
  - Participation constraint can force a pattern on entity instances*
  - Storage, conversion: same as Traditional pattern

The Predicated Pattern: Relational Schema

Aircraft	
ID	Range
3	1500
5	5000
7	6000

Route	
ID	Distance
2	1000
4	3000
6	7000

*Aircrafts are **explicitly** assigned to routes, but each assignment is tested for pre-condition*

Assign	
AID	RID
3	2
5	4
7	4

```
CREATE TABLE Assign
(AID Integer REFERENCES
  Aircraft(ID),
 RID Integer REFERENCES
  Route(ID),
 PRIMARY KEY (AID, RID)
)
```

The *Computed* Pattern: Context, Syntax

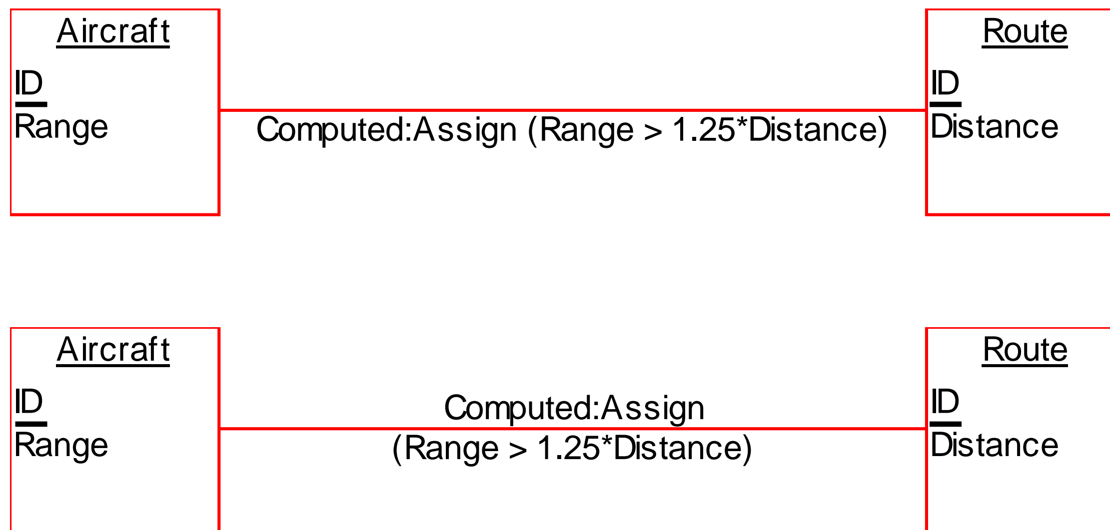
- Context
  - When relationships can be computed (no need to store relationships)
- Syntax
  - Type name is a string of the form

```
Computed:<type>(<predicate>)
```

  - The string '`Computed:`' is fixed

* Syntactic variations shown

The Computed Pattern: Example*



* Cardinality constraints are not interpreted as in traditional relationship type (they are not constraints at all)

The Computed Pattern: Constraints, Semantics, Consequences

- Constraints
  - Relationship attributes must be computable
  - Cardinality can be any*
- Semantics
  - Same as Predicated pattern, except qualifying entity instances are *found* based on the predicate
  - 'Joins' entity sets
- Consequences
  - Storage: Relationships are *not* stored
  - Conversion: Same as Predicated pattern, except a view definition is generated for relationship type

The Computed Pattern: Relational Schema

Aircraft	
ID	Range
3	1500
5	5000
7	6000

Route	
ID	Distance
2	1000
4	3000
6	7000

Aircrafts are ***not*** *explicitly assigned to routes, instead they are found. Table Assign lists candidate aircrafts for each route*

Assign	
AID	RID
3	2
5	2
7	2
5	4
7	4

Definition generated

```
CREATE VIEW Assign
(AID, RID) AS
SELECT AID, RID
FROM Aircraft, Route
WHERE Range > 1.25 *
   Distance
```

* Cardinality M:N with optional participation works for any computed relationship

The Computed Pattern: Cardinality

- Cardinality of a computed relationship is not a constraint; it is a *consequence*
- Cardinality depends on the predicate, the entity sets, and other relationships defined among the entities
  - Equality predicate on keys (if they are joinable) results in 1:1, with optional participation
  - Any predicate produces an empty relationship set when computed over empty entity sets
  - Predicate in the example results in M:N*

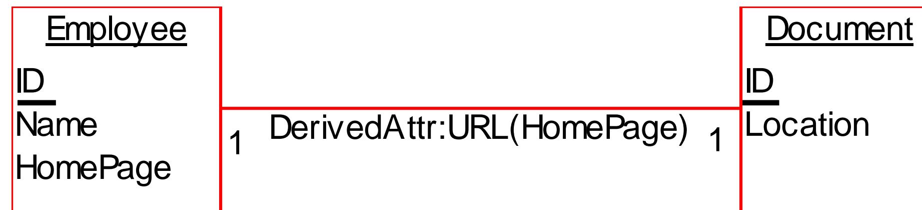
The *Derived-Attribute* Pattern: Context, Syntax

- Context
  - When an attribute′s value is derived (not stored) from other entities (or entities′ attributes)
- Syntax
  - Type name is a string of the form

    `DerivedAttr:<type>(<attribute>)`

  - `<attribute>` is the name of the attribute whose value is derived

The Derived-Attribute Pattern: Example



- The HomePage attribute is *always* the location of a known document
- Its value can be derived from the related Document entity
- Alternative syntax can specify source of value
  - `DerivedAttr:URL(HomePage-Location)`

The Derived-Attribute Pattern: Constraints, Semantics

- Constraints
  - The relationship must be 1-1 or many-1 (we assume single-valued attributes)
  - Participation can be optional
- Semantics
  - Entity instances are related in the traditional way
  - Specified attribute′s value is a function of the attributes of the entities related

The Derived-Attribute Pattern: Consequences

- Storage
  - Same as Traditional pattern, but the value of derived attribute is not stored
- Conversion
  - The derived attribute is excluded from traditional conversion
  - A view definition is generated over the entity whose attribute is derived (to include the derived attribute)

* View definition not needed in some cases when using 'computed columns' available in some RDBMSs

The Derived-Attribute Pattern: Relational Schema

Stored_Employee		
ID	Name	DID
3	DEM	2
5	LMD	4
7	SSM	6

*An employee is **explicitly** associated with a document; attribute HomePage is **not** stored*

Document	
ID	Location
2	dem.jsp
4	lstart.htm
6	ssm.asp

*The schema generated includes the traditional table definitions **and** the view definition**

Employee		
ID	Name	Homepage
3	DEM	dem.jsp
5	LMD	lstart.htm
7	SSM	ssm.asp

```
CREATE VIEW Employee
(ID, Name, HomePage) AS
SELECT S.ID, Name, Location
FROM Stored_Employee S JOIN
   Document D ON DID = D.ID
```

The *Constrained-Group* Pattern: Context, Syntax

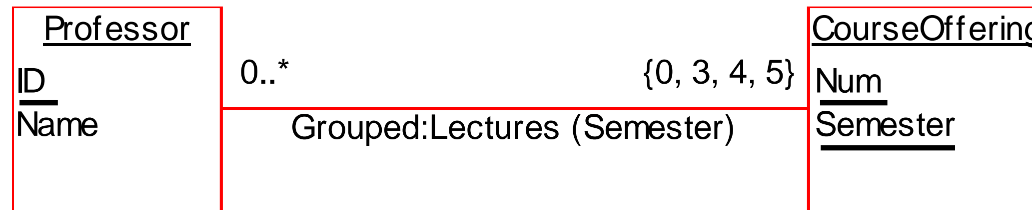
- Context
  - When cardinality constraints must be imposed on a group of attributes
- Syntax
  - Type name is a string of the form

    `Grouped:<type>(<attributes>)`

  - `<attributes>` is a set of attributes to be constrained as a group

* Example due to Thalheim. Fundamentals of Cardinality Constraints. ER 92.

The Constrained-Group Pattern: Example

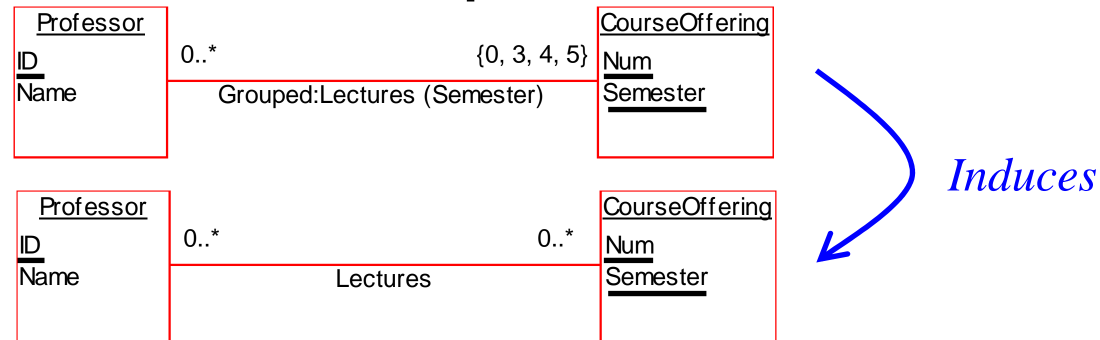


- A professor can teach 0, or 3 to 5 courses a semester*
  - Attribute Semester has values of the form: `S`*`nn`* or `F`*`nn`*, where *`nn`* is a year (Spring or Fall semester)
  - Without grouping, this schema would restrict a professor to 0, or 3 to 5 course offerings *ever*
- The set {0, 3, 4, 5} is an 'Int-cardinality'

The *Constrained-Group* Pattern: Constraints

- The attributes specified must be from one entity (called the *grouped entity*)
- One of the following conditions must be true for the set of attributes specified
  - Contains at least one non-key attribute
  - Contains only some key attributes
- That is, the set of attributes must not be *exactly* the set of key attributes of the grouped entity (reasons for these constraints are described later)

The Constrained-Group Pattern: Semantics



- Groups the specified attributes and key attributes of non-grouped entities; constrains group cardinality
- A constrained-group relationship induces a traditional relationship
  - If `(l, u)` is the group constraint, `(l,*)` is the constraint on the grouped entity. All other cardinality constraints remain the same

The Constrained-Group Pattern: Consequences

- Instance validation
  - Enforce cardinality constraints of the group
  - Enforce cardinality constraints of the induced traditional relationship
- Storage
  - Store the induced traditional relationship
- Conversion
  - Convert the induced traditional relationship type
  - Generate procedure to assist enforcement of group cardinality constraints

The Constrained-Group Pattern: Mechanics

Entity Sets

Professor	
ID	Name
3	DEM
5	LMD

CourseOffering	
Num	Semester
410	F05
512	F05
522	F05
486	F05
410	S05

Relationship Set

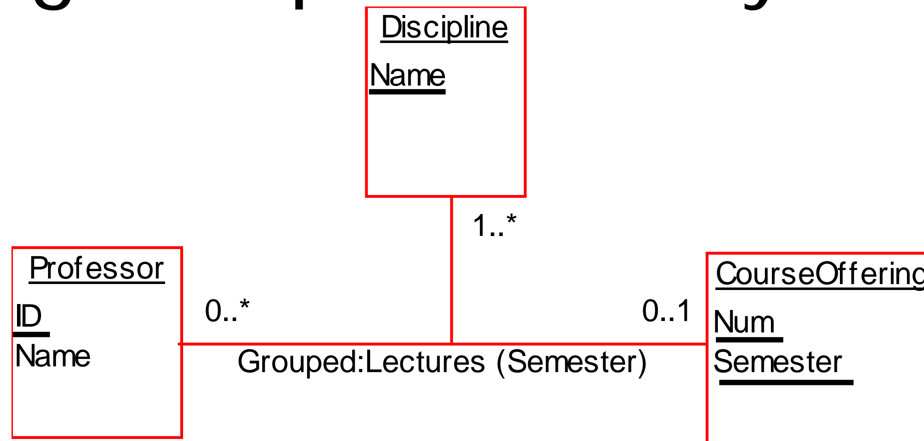
Lectures		
PID	CNum	Semester
3	410	F05
3	512	F05
3	522	F05

```
GROUP BY PID, Semester
```

PID	Semester	Count
3	F05	3

Use count to enforce group cardinality constraints
– Can relate PID 3 with one or two more courses for F05, or delete all three course associations of PID 3 with F05

Constraining Groups in Ternary Relationships



- A professor may teach at most one course in each discipline each *semester*
  - Without grouping, this schema would restrict a professor to one offering in each discipline, ever
- A course offering may be in more than one discipline; a course offering in a discipline may have any number of professors

* Relation Discipline not shown; Lectures models the induced traditional relationship (Ferg's M:M:M)

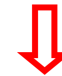
Constraining Groups in Ternary Relationships: Relational Schema*

Professor	
ID	Name
3	DEM
5	LMD

Lectures			
PID	Discipline	CNum	Semester
3	D1	410	F05
3	D2	512	F05
5	D1	522	F05

CourseOffering	
Num	Semester
410	F05
512	F05
522	F05
486	F05
410	S05

```
GROUP BY PID, Discipline, Semester
```

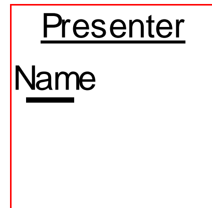


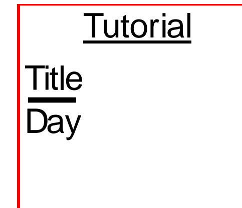
PID	Discipline	Semester	Count
3	D1	F05	1
3	D2	F05	1
5	D1	F05	1

Use count to enforce group cardinality constraints

* This example also demonstrates constraining groups that contain non-key attributes

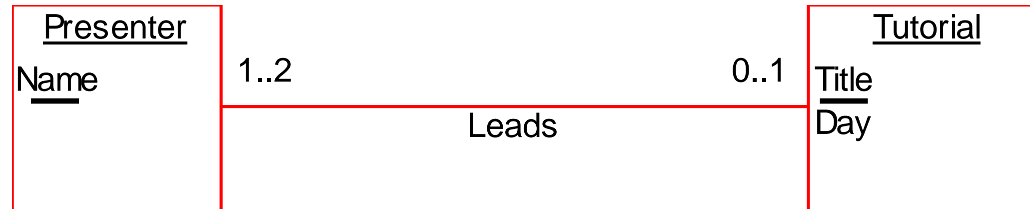
Constraining Groups: Analysis of Alternatives*





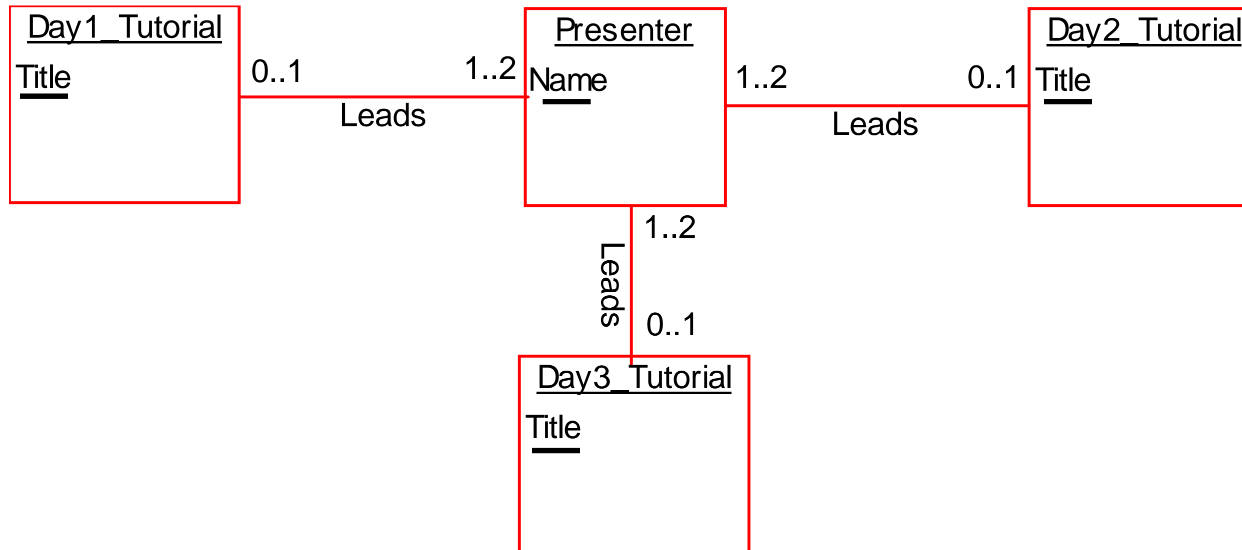
- A 3-day conference offers tutorials
  - A tutorial is 1-day long and is offered once and only once
  - Day is a non-null integer value from {1, 2, 3}
- A tutorial has 1 or 2 leaders; a presenter may lead at most one tutorial per day
  - A presenter might not lead any tutorial

Scheduling Tutorials: Alternative 1



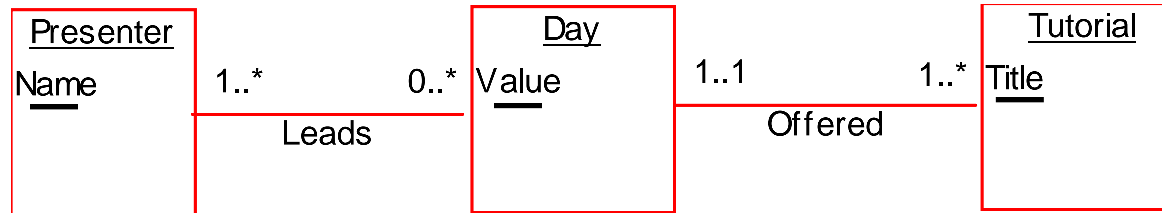
- The entities are modeled "naturally"
  - A tutorial has a title and the day it is offered
- Limits a tutorial to one day, but limits a presenter to one tutorial for the conference
  - Cannot make Title+Day the key; that allows multiple offerings of a tutorial

Scheduling Tutorials: Alternative 2



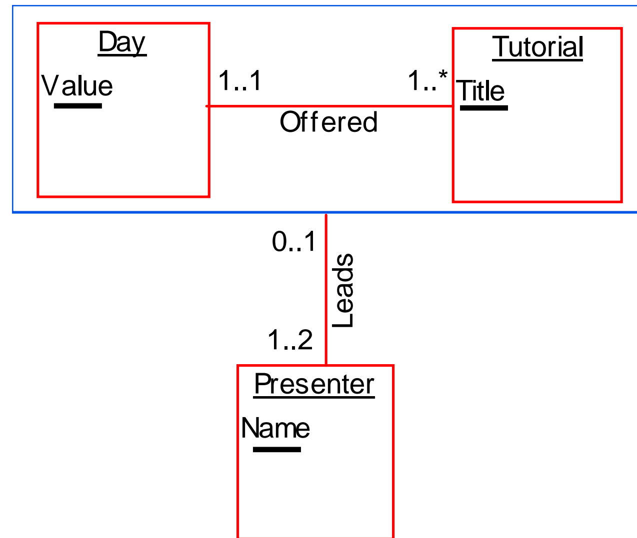
- Models cardinality constraints, but not the uniqueness constraint on tutorial
- Many entity types, yet feasible because of few number of days
- Need union query to list all tutorials

Scheduling Tutorials: Alternative 3



- Limits a tutorial to one day, and a presenter is scheduled at most once a day, but does not constrain number of presenters per tutorial
- Which tutorial does a presenter lead on a given day? (A day has one or more tutorials.)
- The Tutorial entity is broken up, but the scheme works for any number of days

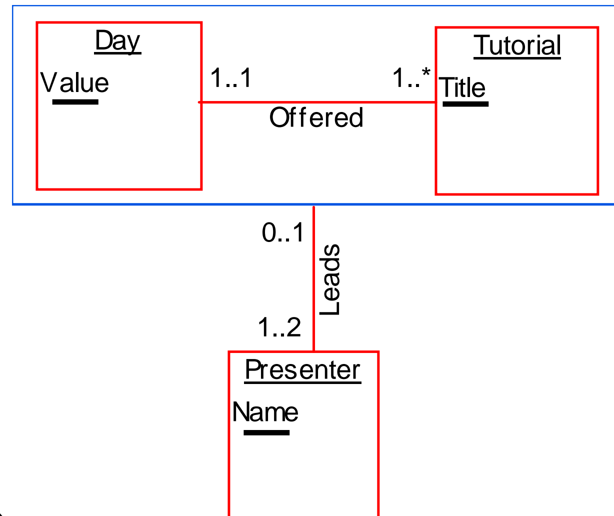
Scheduling Tutorials: Alternative 4



- Correctly models all constraints
- Relational schema generated using traditional conversion procedure does not preserve constraints

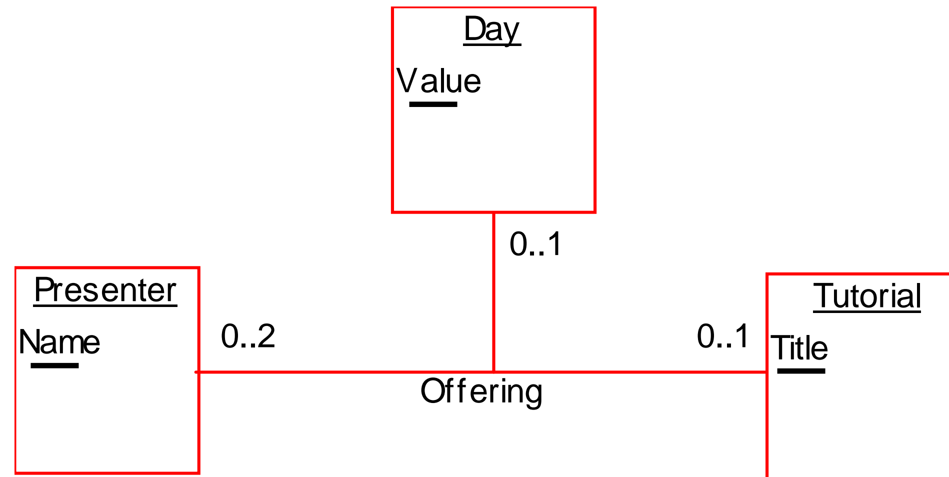
* Tutorial.Offered, and Presenter.Title are foreign keys

Relational Schema for Alternative 4



- Day (Value)
- Tutorial (Title, Offered) (Offered non-NULL)
- Presenter (Name, Title) (Title can be NULL)
- Presenter has at most one tutorial for conference

Scheduling Tutorials: Alternative 5



- Models all constraints correctly
- Tutorial entity broken up
- Hard to comprehend
- Conversion to relational schema is not easy

* Ferg's M:1:1 scenario, two candidate keys for the relationship table; Relation Day not shown

Relational Schema for Alternative 5*

Presenter
Name
DEM
LMD

Tutorial
Title
ER-1
UML-1
XML-1
ER-2
UML-2
XML-2

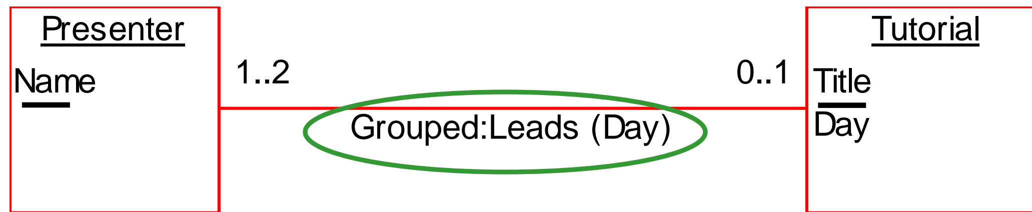
Offering-1		
Presenter	Tutorial	Day
LMD	ER-1	1
DEM	XML-1	2
LMD	UML-2	3

Presenter limited to one tutorial a day; a tutorial not limited to one day, and it might not at all be scheduled

Offering-2		
Presenter	Tutorial	Day
LMD	ER-1	1
DEM	XML-1	2
LMD	UML-2	3

Presenter not limited to one tutorial a day; a tutorial not limited to one day, and it may not at all be scheduled

Scheduling Tutorials: Alternative 6



- Constrains the group Presenter+Day
- Satisfies all key and cardinality constraints
  - Constraint 0..1 applies to presenter and day group
- Relationship is binary
- Tutorial entity is intact
- Traditional conversion to relational schema suffices

* A join is needed because Day is not in relation Leads

Relational Schema for Alternative 6

Presenter
Name
DEM
LMD

Tutorial	
Title	Day
ER-1	1
UML-1	1
XML-1	2
ER-2	2
UML-2	3
XML-2	3

A tutorial has a day, and only one day

Leads	
Name	Title
LMD	ER-1
DEM	XML-1
LMD	UML-2

```
JOIN Leads, Tutorial;
GROUP BY Name, Day *
```

Name	Day	Count
DEM	2	1
LMD	1	1
LMD	3	1

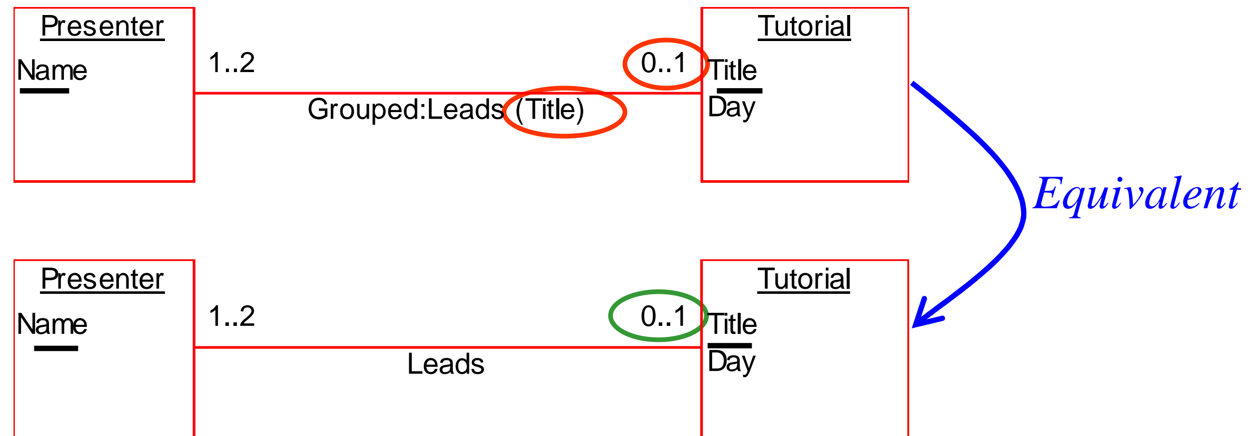
LMD can now only be assigned to XML-1 or ER-2

* Yes/No means possible/not natively in relational model; post-processing can address items with ‘No’

Enforcement: Alternative 5 vs. Alternative 6*

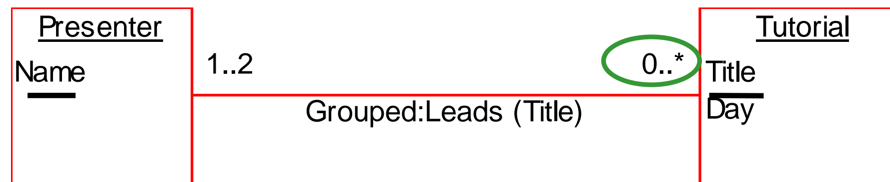
Constraint	Alternative 5 (using Offering-1)	Alternative 6
A tutorial is scheduled	No	Yes
A tutorial is scheduled once	No	Yes
A tutorial has a presenter	No	No
A tutorial has no more than two presenters	Instance precondition	Instance precondition
A presenter does not have to lead on a day	Yes	Yes
A presenter leads no more than one tutorial a day	Yes (but must pick the right schema. *E.g.*, Offering-1 vs. Offering-2)	Instance precondition (but automatic schema and query)

Constraining Keys as Groups: Equivalence



- The first relationship type constrains a presenter to one tutorial
- The second, traditional relationship type conveys the same meaning

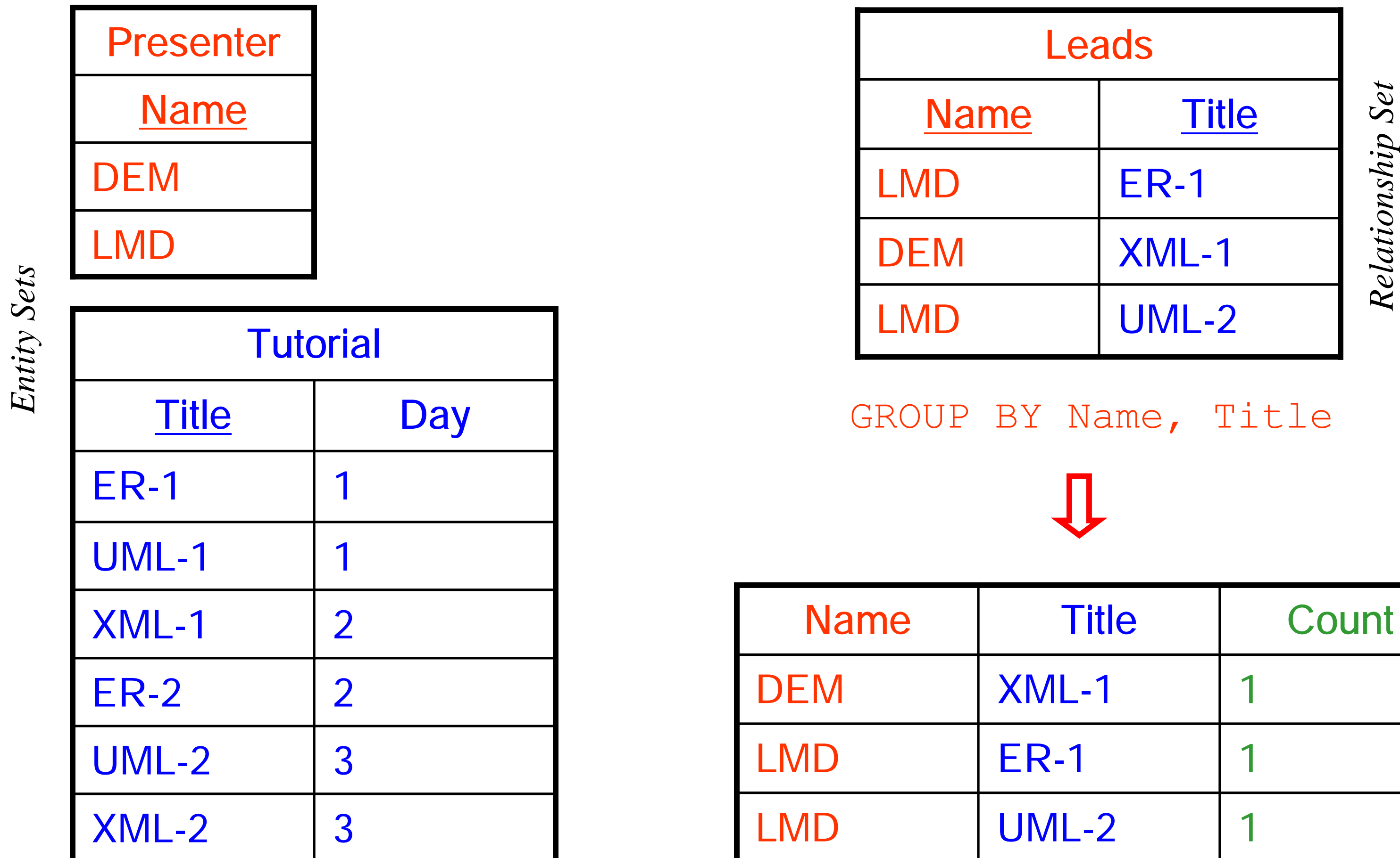
Constraining Keys as Groups: Redundancy



- This relationship asks a presenter to be associated with the *same* tutorial many times
  - Causes redundant relationships
  - Our mechanism to enforce group constraints does not support this scenario
- The *History* pattern captures temporal relationships
  - A presenter is assigned different tutorials over time, but at any one time is assigned *one* tutorial

* If such grouping were allowed

Constraining Keys as Groups: Mechanics*



Entity Sets

Presenter
Name
DEM
LMD

Tutorial	
Title	Day
ER-1	1
UML-1	1
XML-1	2
ER-2	2
UML-2	3
XML-2	3

Relationship Set

Leads	
Name	Title
LMD	ER-1
DEM	XML-1
LMD	UML-2

```
GROUP BY Name, Title
```

⇩

Name	Title	Count
DEM	XML-1	1
LMD	ER-1	1
LMD	UML-2	1

Group count can be at most 1 because of grouping on key attributes. So, cannot support cardinality > 1

Constraining Keys as Groups: Restrictions

- A constrained-group relationship over exactly the key attributes of an entity can be a source of confusion and conflicts
  - Defeats a purpose of conceptual modeling
- We could improve our mechanism to handle grouping of keys exceptionally, but we disallow such grouping in the interest of clarity
  - We recommend the use of other appropriate patterns such as Traditional and History patterns

Constrained Groups: Summary

- Constrained grouping controls the number of times entities might be associated with a set of attributes of a grouped entity
- Cardinality enforcement requires a join with the grouped entity if one of the grouping attributes is not a key attribute
- A constrained-group relationship induces a traditional relationship
- Constrained-grouping over exactly the key attributes of an entity is not allowed

* See: Fowler. Analysis Patterns: Reusable Object Models. 1997.

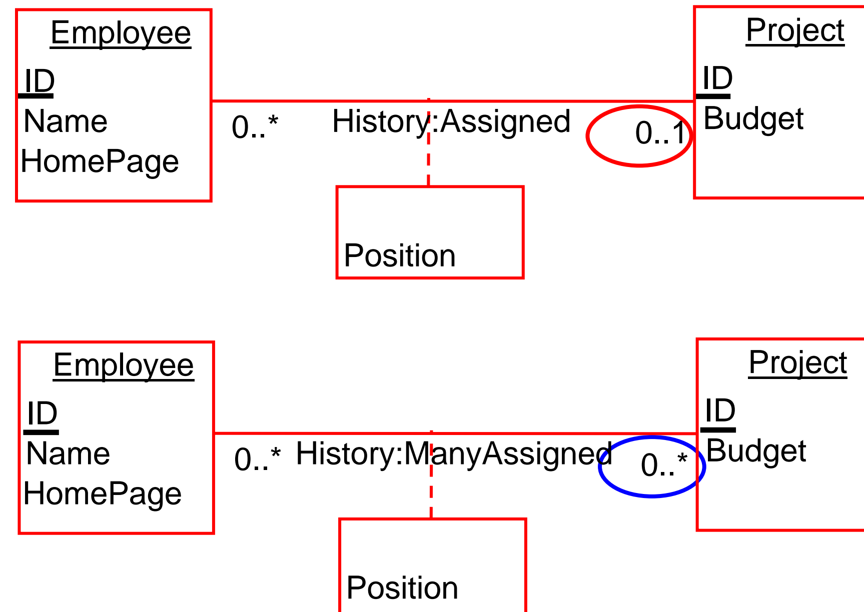
The *History* Pattern: Context, Syntax, Constraints

- Context
  - When a *history* * of relationships needs to be maintained
- Syntax
  - Type name is a string of the form

```
History:<type>
```

- Constraints
  - Same as Traditional pattern

The History Pattern: Examples



- 'Assigned' limits an employee to one project at a time; 'ManyAssigned' allows an employee to be assigned any number of projects
- In both cases we like to maintain a history of changes to relationships

*Some applications might require a timestamp; we use the term timestamp to mean either timestamp or seq.#

The History Pattern: Semantics, Consequences

- Semantics
  - Same as Traditional pattern, except a history of each relationship is maintained
  - A timestamp or a sequence number is used to map the history of a relationship*
- Consequences
  - Storage: Store a copy of each state of a relationship with corresponding timestamp
  - Conversion: Same as Traditional pattern, except *always* create a relationship table and add a timestamp attribute to the key; add an 'Action' attribute if the relationship is M:N

The History Pattern: Updates (M:1 or 1:1)

- Each update (insert, change, delete) creates a new instance, with *current* timestamp
- Insert: Write values (relationship attributes and keys of participating entities)
  - Possible only if the 1-entity has never been related before
- Change: Write new values
- Delete: Write NULL values (except for timestamp)
  - Possible only if participation is optional

* For simplicity, we use sequence numbers instead of timestamps (attribute TS)

The History Pattern: Some Instances (M:1)*

Employee	
ID	Name
3	DEM
5	LMD

Project	
ID	Name
12	FP
19	SW

A traditional M:1 relationship does not need a separate table (would be captured in the relation Employee)

Assigned				
TS	EID	PID	Position	*Action (not recorded)*
1	3	12	MGR	Inserted
2	3	NULL	NULL	Deleted
3	3	19	CST	Inserted
4	3	19	MGR	Changed
5	3	12	MGR	Changed

* Alternatively, can treat as an error

The History Pattern: Updates (M:N)

- Each update (insert, change, delete) creates a new instance, with *current* timestamp (or a new sequence number)
  - 'Action' attribute reflects the operation performed
- Insert: Write values
  - Treat as update if entities are already related*
- Change: Write new values
- Delete: Copy values of the instance being deleted and mark action as 'Delete'

The History Pattern: Some Instances (M:N)

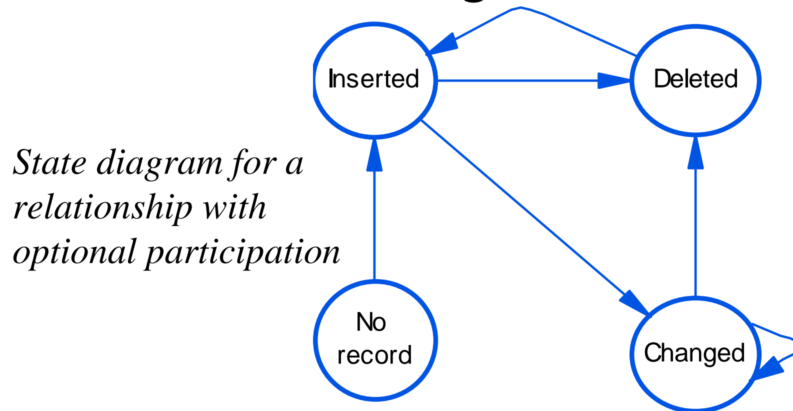
Employee	
ID	Name
3	DEM
5	LMD

Project	
ID	Name
12	FP
19	SW

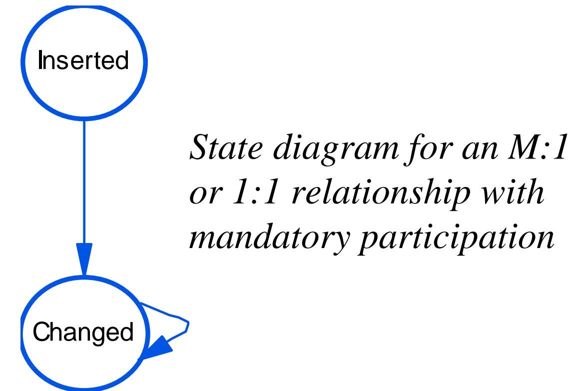
ManyAssigned				
TS	EID	PID	Position	Action
1	3	12	MGR	Inserted
2	3	12	MGR	Deleted
3	3	19	CST	Inserted
4	3	19	MGR	Changed
5	3	12	MGR	Inserted
6	3	12	CST	Changed

* Here we omit the case of an M:N relationship with mandatory participation

The History Pattern: Relationship States



State diagram for a relationship with optional participation



State diagram for an M:1 or 1:1 relationship with mandatory participation

- Generally, a relationship is in one of four states
  - No record: The relationship has no history
  - Inserted: The relationship is current
  - Changed: Some aspect of the relationship is altered
  - Deleted: The relationship is not current
- An M:1 or 1:1 relationship with mandatory participation has only Inserted and Changed states*

* These issues do not arise in the history of M:1 and 1:1 relationships

The History Pattern: Handling Changes (M:N)*

- Changing *key attributes* of participating entities needs special care. *E.g.*,
  `UPDATE ManyAssigned SET PID=19 WHERE PID=12`
- If the new primary key is not current (`EID=3, PID=19` has 'No record' or is 'Deleted')
  - 'Delete' the relationship being changed (`EID=3, PID=12`) and 'Insert' a new relationship (`EID=3, PID=19`)
- If the new primary key is current (`EID=3, PID=19` is already 'Inserted' or 'Changed')
  - The change operation is an error

* We omit the details of retrieving history

The History Pattern: Retrieval*

- Retrieve current relationship or relationships for one entity
- M:1 or 1:1 Relationship
  - Retrieve the instance with the highest timestamp
  - No current instance if key value for the related entity is NULL or if an instance does not exist
- M:N Relationship
  - For each related entity, retrieve the instance with the highest timestamp
  - No current instance with entity if action is ‘Deleted’ or if an instance does not exist (‘No record’)

* Stereotype and parameterized collaboration provide the best mechanisms to incorporate patterns into UML

Exemplar vs. UML

Exemplar Pattern	UML Feature	Remarks about UML feature
Predicated	Constraint, OCL Precondition	Same expressive power
Computed	Derived Association	Semantics of multiplicity unclear. Does a derived association need a base association? No clear example
Derived Attribute	Derived Attribute	Association not needed for all derivations (UML is better)
Constrained Group	Qualified Association	Removes grouping attribute from class (that attribute cannot participate in other associations)
History	–	Can be modeled as a stereotype*

ER Relationship Patterns vs. UML

- Enrich ER
  - ER is lightweight; many in the DB community use ER
- Patterns provide more than just notations
  - A pattern analyzes specific needs and provides comprehensive solutions. It makes explicit details such as constraints and consequences
- Patterns can be incorporated into UML
  - UML provides syntax and semantics for some patterns, but we need to address the details. *E.g.*, UML defines no semantics for stereotypes

The Framework

Relationship Typespaces

- A relationship typespace is a set of (related) relationship types
  - *E.g.*, `Computed`; it contains relationship types whose instances can be computed
- A relationship type belongs to one typespace
- It disambiguates relationship types and provides a handle to describe a set of relationship types
- Typespaces are not necessary
  - The *Global* typespace is assumed if no typespace is specified

The Framework Provides

- Requirements for a pattern specification
- A form to specify patterns
  - A template pattern specification is available
- A syntax for names of relationship types, roles, and cardinality constraints
- A syntax for drawing relationships
- Guidelines for the kinds of constraints that need to be considered when defining relationship patterns
- An example pattern language (Exemplar)

A Relationship Pattern Specification Includes

- C : A description of the context in which the pattern applies
- S : A syntax for relationship type names and role names, constraints and parameters, and for drawing
- T : A set of constraints that apply to the relationships and the related entities
- M : A description of the semantics of the relationships
- N : A description of the consequences of using the pattern
- E: A set of instructive example uses of the pattern

$$P = \{ C, S, T, M, N, E \}$$

Relationship Signatures

- The *text syntax* of the relationships of a pattern is specified using a set of three *signatures* to generate typespace and type names, role names, and cardinality constraint descriptions respectively
- The three signatures are together called the *relationship signature* $S = \{ S_T, S_R, S_C \}$
- Each signature is formally defined using a grammar. The framework defines a *super* relationship-signature

Type Signatures (S_T)

- A type signature is a string of the form
  `<typespace>:<type>(<parameters>)`
- `<typespace>` is a string that identifies a typespace
- `<type>` is a string that identifies a relationship type in a typespace
- `<parameters>` is auxiliary information

*Angle brackets contain non-terminal symbols; double angle brackets contain informal definition

A CFG for the S_T Super Language*

```
<S_T>        :: <typespace><type><params>
<typespace>:: ε | <ncpchar><eorncp><colon>
<eorncp>     :: ε | <ncpchar><eorncp>
<type>       :: ε | <ncpchar><type>
<params>     :: ε | (<fparams>) | {<fparams>} |
                [<fparams>]
<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 >>
```

Some Type Signatures

- `<type>` (Traditional)
  - *E.g.*, `Assign`
- `<type>(<predicate>)`
  - *E.g.*, `Assign (Range>1.25*Distance)`
- `Computed:<type>(<predicate>)`
  - *E.g.*,
    `Computed:Assign (Range>1.25*Distance)`

Role Signatures (S_R)

- A role signature is a string of the form

  `<role>(<parameters>)`

- `<role>` is a string that identifies an entity's role in a relationship
- `<parameters>` is auxiliary information

A CFG for the S_R Super Language

```
<S_R>      :: <role><params>
<role>     :: ε | <npchar><eornp>
<eornp>    :: ε | <npchar><eornp>
<params>   :: ε | (<fparams>) | {<fparams>} |
              [<fparams>]
<fparams>:: <chars> | (<fparams>) |
              {<fparams>}| [<fparams>] |
              <fparams><fparams>
<chars>    :: ε | <npchar><chars>


<npchar> :: << not a bracket char ( ) { } [ ] >>
```

* Shows a possible alternative syntax for this pattern

Some Role Signatures

- `<role>` (Traditional)
  - *E.g.*, `Project`
- `<role>(<predicate>)` (Predicated*)
  - *E.g.*, `Project(Budget>100K)`
- `<role>{<constraint>}` (UML constraint on anonymous role)
  - *E.g.*, `{Ordered}`

Cardinality Signatures (S_C)

- A cardinality signature is a string of the form `<lower>..<upper>` or `{<numset>}`
- `<lower>` and `<upper>` represent minimum and maximum cardinalities
  - Zero is the default lower limit
- `{<numset>}` specifies Int-cardinality
- Can specify either look-across or participation cardinalities, or both
  - Look-across cardinality is the default

* <lcard> and <pcard> specify look-across and participation cardinalities respectively

A CFG for the S_C Super Language*

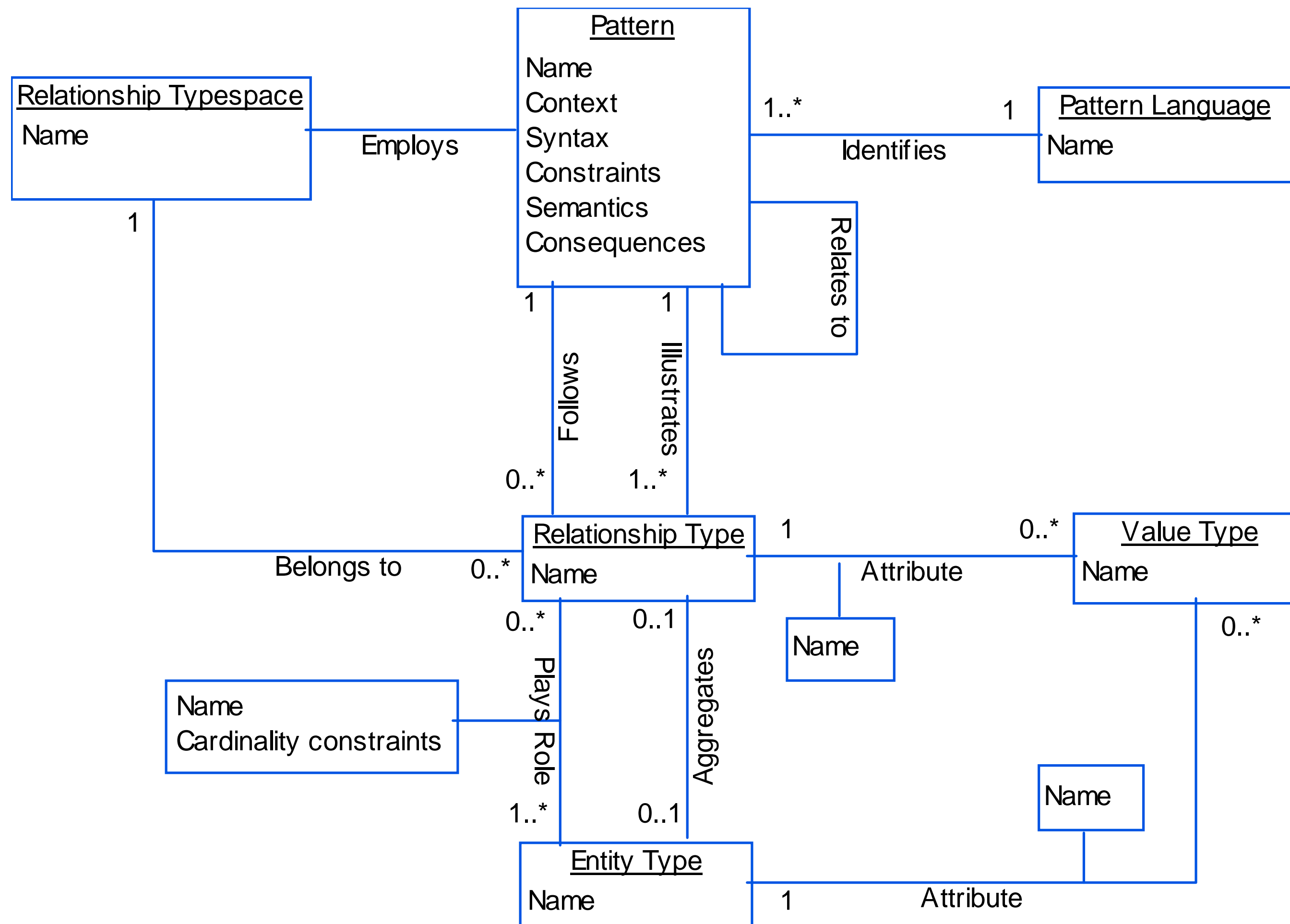
```
<SC>      :: <card> | <lcard> | <pcard> |
             <lcard>,<pcard> | <pcard>,<lcard>
<lcard> :: L#(<necard>)
<pcard> :: P#(<necard>)
<card>  :: ε | <necard>
<necard>:: <lower>..<upper> | <upper> |
           {<numset>}
<lower> :: <letter> | <number>
<upper> :: <letter> | <number> | <star>
<numset>:: <number> | <number>,<numset>

<star>  :: << the character * >>
<letter>:: << a character a-z or A-Z >>
<number>:: << a non-negative integer >>
```

Some Cardinality Signatures

- `<lower>..<upper>`
  - *E.g.*, `0..*`
- `<upper>`
  - *E.g.*, `M` or `*` or empty (means `*`)
- `{<numset>}`
  - *E.g.*, `{0,3,4,5}`
- `<pcard>,<lcard>`
  - *E.g.*, `P#(0..1),L#(*)`

A Conceptual Model for Relationship Patterns



Model Details

- Pattern language identifies at least one pattern
- A pattern or a typespace might never be used
  - ‘Follows’ does not include ‘Illustrates’; separates pattern applications from examples
- A relationship type always follows a pattern, and always belongs to a typespace
  - At least Traditional pattern and global typespace
- An entity type might not contribute to any relationship type; it might have no attributes

Related Work

- Patterns in architecture
  - Christopher Alexander and others
- Software Patterns
  - Gang of Four, Fowler, Pattern Languages of Program Design (series)
- Representing requirements
  - Cysneiros and others (focus on non-functional requirements)
- Rules and constraints
  - ER-R (rules and triggers)
  - UML, OCL

Summary

- Relationship patterns express common conceptual-design situations
- Well-designed relationship signatures can improve expression of relationships
  - They can also assist in customizing actions performed on the conceptual schema
- The framework provides a means to create pattern languages
  - It also defines a pattern language for some patterns observed, including the traditional pattern

Drawbacks of using Relationship Patterns

- Requires "specifications"
  - A pattern is specified once, and used many times
  - Specifications clarify semantics, consequences, …
- The conceptual schema might not be simple
  - But, it is likely more expressive and accurate
  - Build on existing patterns
- A signature allows arbitrary parameters
  - Keep a signature's language simple
  - Reference an external constraint specification