This is a pre-print of the book chapter M. Jeusfeld: “Metamodeling and method engineering with ConceptBase”. In Jeusfeld, M.A., Jarke, M., Mylopoulos, J. (eds): Metamodeling for Method Engineering, pp. 89-168, The MIT Press, 2009; the original book is available from MIT Press http://mitpress.mit.edu/node/192290

Metamodeling and Method Engineering with ConceptBase

Manfred Jeusfeld

Abstract. This chapter provides a practical guide on how to use the meta data repository ConceptBase to design information modeling methods by using meta-modeling. After motivating the abstraction principles behind meta-modeling, the language Telos as realized in ConceptBase is presented. First, a standard factual representation of statements at any IRDS abstraction level is defined. Then, the foundation of Telos as a logical theory is elaborated yielding simple fixpoint semantics. The principles for object naming, instantiation, attribution, and specialization are reflected by roughly 30 logical axioms. After the language axiomatization, user-defined rules, constraints and queries are introduced. The first part is concluded by a description of active rules that allows the specification of reactions of ConceptBase to external events. The second part applies the language features of the first part to a full-fledged information modeling method: The Yourdan method for Modern Structured Analysis. The notations of the Yourdan method are designed along the IRDS framework. Intra-notational and inter-notational constraints are mapped to queries. The development life cycle is encoded as a software process model closely related to the modeling notations. Finally, aspects managing the modeling activities are addressed by metric definitions.

3.1 Introduction

The engineering of a modeling method is per se a software development effort. The result is software implementing a consistent set of modeling tools that are used to represent information about some artifacts. Modeling tools obey to some underlying principles that make a dedicated environment for method engineering useful. The strongest underlying principle is the common artifact focus: the result of applying a method is a set of models which all make statements about the same set of artifacts. Consequently, the models are interrelated. A statement in one model has implications on other statements in the same or other models about the common set of artifacts. The artifact focus of method application regards single models as viewpoints on the artifact. A model is not containing all information about the artifact but only the information that is relevant for the viewpoint. The second principle is modeling lifecycle. Any model has a purpose. The content of the model is the result of some modeling step and pre-requisite for some other modeling, development, or analysis step. The model content must be represented in a way that is useful for the purpose.

This chapter presents a logical approach to method engineering. It uses a simple yet powerful logic to

cover the following aspects of method engineering. First, the set of allowed symbols and the allowed combination of these symbols is represented in the so-called notation level. Second, the semantics of modeling notations is investigated by incorporating a model and data level. Third, a method is defined as combination of several notations interrelated by inter-notational constraints. Fourth, method application is represented by process models that pre- or describe the modeling steps of human experts. Finally, the issue of model quality is discussed. The construction of new methods is facilitated by a multiple perspective approach in which a high level meta-model (the notation definition level) encodes which modeling viewpoints shall be supported and which interrelationships between viewpoints require method support. The chapter uses the ConceptBase meta database system as method engineering tool. It implements the logic underlying our approach and provides the necessary functions for notation definition. Furthermore, it can be used as a prototyping environment for method application. The examples in this chapter are tested with ConceptBase and are also available on the companion CD-ROM.

The chapter is organized as follows. First, the meta-modeling approach is introduced defining the four abstraction levels mentioned above in a propositional logic. Then, the propositional statements are aggregated into frame-like objects using the Telos language. After a small primer on deductive databases, we discuss pre-defined propositions for classification, specialization, and attribution and define their semantics by first-order logic axioms. Then, user-defined deductive rules and integrity constraints are introduced into the language framework. The framework description is completed by subsections on the Telos query language and so-called active rules. The second main part of the chapter is a case study on engineering the Yourdan system modeling method. It starts with engineering the entity-relationship diagramming technique in Telos using the four abstraction levels. The same principle is applied to the second major Yourdan technique, the data flow diagrams. We discuss how interrelationships between notations can be represented as constraints in a query language. Some of the interrelationships between notations can be derived from common patterns of instantiation. After having discussed the definition of two notations, a software process layer is introduced which treats models as products and their creation as development steps. The software process model itself is created from the same abstraction principles as the Yourdan notations.

3.2 Modeling is knowledge representation

ConceptBase has been developed within an academic context but has been applied in both research and industrial projects. Originally, it was designed as a repository system aiding the development of data-intensive software. The modeling notations were covering the whole range from requirements analysis to implementation. Hence, the repository had to be able to manage quite heterogeneous modeling languages and their interrelationships. It was decided to use the features of the Telos (Mylopoulos et al., 1990) knowledge representation language that has its roots in requirements analysis (Greenspan, 1984). As a knowledge representation language, Telos has not a rich set of predefined features. All information in Telos has the basic format:

statement number: subject is-related-to object

A statement relates two things ‘subject’ and ‘object’ to. The statement itself is identified and can occur as subject or object in other statements. The simplicity of this representation is the key to its extensibility. Since the statement is identified (also known as reification), one can express statements about the statement itself. Indeed, the tokens ‘subject’ and ‘object’ are standing for other statements defining them. The statement structure is so universal that is can represent objects, classes, meta classes, attributes, specializations, classifications, queries, rules, and constraints.

To use statements for modeling and meta-modeling, the statements are assigned to IRDS

levels (see chapter sec-12) that express their concreteness or abstractness. The following motivating example illustrates the abstraction levels:

statement 1: Bill earns 10000 Dollars.
(Token level)
statement 2: Employees earn salaries.
(Class level)
statement 3: Entities can have attributes taken from a domain.
(Meta class level)
statement 4: Nodes are connected to nodes.
(Meta meta class level)

The statements become more abstract from one level to the next. More specifically, each statement can be regarded as an example (=instance) of the subsequent. This principle is called *class abstraction*. From the viewpoint of the more abstract statement, it is called *instantiation*. Rather than defining what a class is, we define the instance-to-class statement. This view allows treating objects completely uniformly independent from their abstraction level.

statement 5: Bill is an instance of Employee.
(Token to class)

statement 6: 10000 Dollars is an instance of salary.
(Token to class)

statement 7:The concept Employee is an instance of the concept entity.
(Class to meta class)

statement 9: An entity is an instance of the concept node.
(Meta class to meta meta class)

Meta-modeling as well as modeling is seen as expressing statements about an artifact. The artifact of meta-modeling is a modeling environment: the meta-model expresses what provisions the modeling environment shall have. Hence, meta-modeling is just like any conceptual modeling of information systems. The only specialty is that the artifact of meta-modeling is modeling. In other words: a flexible modeling environment should also be able to model-modeling environments.

The statement structure is so generic that it not only allows expressing relations between ‘simple’ objects like ‘Bill’ and ‘1000’ but also between statements themselves:

statement 10: statement 1 is an instance of statement 2.
statement 11: statement 1 is an instance of an attribute.
statement 12: statement 2 is an instance of an attribute.
statement 13: statement 5 is an instance of instantiation (InstanceOf).
statement 14: Bill is an ordinary object (Individual).

The last 4 statements are referring to predefined statements of the Telos language to express objects, their instantiation, and their attributes. There is one additional predefined statements standing for specialization (IsA). Its use is shown further below.

A Telos database is essentially a semantic network where concepts (nodes) and their interconnections (links) are treated uniformly as objects. An object can be at any abstraction level as motivated above. Even abstraction levels beyond the meta meta class level are allowed. The feature that allows this flexibility is the explicit representation of the instance-to-class relationship. In programming languages and databases, the type of a variable or field (e.g. `INTEGER`) is only visible in the program code or schema definition. At run time, the data is held at a location of memory of suitable size. All references to the type label `INTEGER` are compiled either into memory layout or have been employed for type checking during compile time (or database creation time). In Telos, the instance-class relationships are data themselves. We express them as a binary relationship[1] `(x in c)` with the reserved label ´`in`´. The following facts encode the class abstractions of the motivating example:

```
(Bill in Employee), (10000 in Integer)
(Employee in Entity), (Integer in Domain)
(Entity in Node), (Domain in Node)
```

The example has four abstraction levels. Hence, there are 3 instance-to-class gaps between the objects. The instance-to-class relationship is not sufficient to express all phenomena of knowledge representation. In particular, there are relationships between objects of the same (or different) abstraction levels that are not interpreted as instantiations. An example is the fact the objects Bill and 10000 are connected by a binary relationship ´earns´. Obviously, this is considered to be an example of the fact that Employees have salaries. In Telos, we use a predicate `(x m/l y)` to express such *attribution relationships*. We say that the objects `x` is having an attribute relationship with label `l` and category `m` to the object `y`. The example completes as follows:

```
(Bill salary/earns 10000)
(Employee feature/salary Integer)
(EntityType connectedTo/feature DomainOrObjectType)[2]
(Node attribute/connectedTo Node)
```

The pair of attribute category and attribute label crosses two abstraction levels. For example the category ´salary‘ in the fact `(Bill earns/salary 10000)` is defined at the class level in the fact (Employee feature/salary Integer). The expressive power of this pair becomes apparent when two attributes use the same attribute category, for example:

```
(Employee feature/colleague Employee)
```

The ´colleague´ attribute is a feature of `Employee` just like the ´`salary`´ attribute. This phenomenon is called *multiple instantiation*: the attribute category ´`feature`´ defined at `EntityType` is instantiated multiple times for defining `Employee`. Multiple instantiation is useful at all abstraction level, for example:

```
(Bill colleague/col1 Mary)
(Bill colleague/col2 Jim)
```

Multiple instantiation also occurs for ordinary objects as well. For example, the class `Employee` can

have multiple instances:

```
(Bill in Employee)
(Mary in Employee)
(Jim in Employee)
```

There is a reverse application of this principle is called *multiple classification*: the same object can be instance of multiple classes. For example, the object `Bill` may be known to be instance of `Employee` but also of `Pilot`:

```
(Bill in Employee)
(Bill in Pilot)
```

The third and last structural relationship in Telos is the *specialization* (reverse: *generalization*). While instantiation can be roughly compared to the element-vs-set relationship in set theory, specialization is the counterpart of the subset relationship. A specialization is denoted by a predicate `(c isA d)`. We say that c is defined as *subclass* of d (or: d is *superclass* of c). Specialization can be applied to objects of any abstraction level. For example, one can define a subclass `Manager` of `Employee` and a superclass `ObjectType` of `EntityType`:

```
(Manager isA Employee)
(Manager feature/salary HighInteger)
(HighInteger isA Integer)
(HighInteger in Domain)
(EntityType isA ObjectType)
```

The subclass `Manager` refines the `salary` attribute of `Employee` to a subclass of `Integer`. An instance of `Manager` is then automatically regarded as an instance of Employee as well:

```
(John in Manager)
(John salary/gets 500000)
(500000 in HighInteger)
```

The logical foundation of Telos as presented in section 3.6 makes sure that `(John in Employee)` as well as `(500000 in Integer)` do not need to be stated explicitly but are derivable via built-in rules of Telos. Sometimes, we shall include derived facts in a list of explicit facts. The semantics of the logical framework removes such duplicates automatically. Indeed, the facts as shown above are based on even more basic facts using the P-predicate from which they are derived. The P-predicate is explained in section 3.6.

3.3 Universal references to objects

The factual representation of Telos statements as presented in the previous section always referred to two objects and their relation. For example, the statement `(John in Manager)` is a binary relationship between the objects `John` and `Manager`. The relationship '`in`' is interpreted as 'is an instance of'. While `John` and `Manager` are object names on their own right, it is unclear whether the binary relationship between the two is also an object, i.e. a statement that we can refer to. In fact, Telos allows regarding all relationships between objects as full-fledged objects that are subject to participate in further relationship. For example, the object standing for `(John in Manager)` is an instance of another object standing for the fact that objects can be instance of other objects[3]. The problem now is that we have so far no expression for the objects that are establishing a relationship (or link). This is overcome by the subsequent naming convention.

Objects in Telos are either node objects or link objects. All abstraction principles then also apply to link objects. To do so, one has to define syntax for referring to link objects. This is achieved by the following recursive definition:

R1. The reference of a node object is its label, e.g. `Employee`, `Bill`, etc.
R2. If an object O has the reference N and has an explicit attribute with label a, then the reference of this attribute is `(n!a)`. The parentheses can be omitted when there is no ambiguity in the reading of the expression. Example: `Employee!salary`, `Bill!earns`, `EntityType!feature`.
R3. If there are two objects O1 and O2 with references N1 and N2 and O1 is an explicit instance of O2, then `(N1->N2)` is the reference of the instantiation object. Examples: `(Bill->Employee)`, `(Employee->EntityType)`, `((Bill!earns)->(Employee!salary))`.
R4. If there are two objects O1 and O2 with references N1 and N2 and O1 is an explicit subclass of O2, then `(N1=>N2)` is the reference of the specialization object.

The term 'explicit' requires some explanation. As to be seen later, logical rules can be used to define how to derive attribute, instantiation, and specialization statements. We call a statement like `(Bill in Employee)` explicit if it is explicitly stored in the database. Then, this statement is regarded as an object on its own right that can have a reference, here `(Bill->Employee)`. Derived facts do not have this object identity property. Hence, if `(Bill in Employee)` is a consequence of some derivation rule, then one cannot refer to this fact. It can just occur in other rules, queries, and constraints. In particular one cannot attach further attributes to a derived fact.

With the reference convention, one can express instantiation and specialization relationships between attributes:

```
(Bill!earns in Employee!salary)
(Manager feature/salary HighInteger)
((Manager!salary) isA (Employee!salary))
```

The first fact corresponds to statement 10 in the motivation. The specialization definition in the third fact can be referenced by a rather complicated expression. Note that a reference to an object is

different from its definition. The object referred by

```
((Manager!salary)=>Employee!salary))
```

is indeed a specialization object between two attribute objects. On the other hand, the fact `((Manager!salary) isA (Employee!salary))` represents the statement that there is a specialization relationship between two attributes (regardless of whether this is an explicit fact or derived by some rule).

We use the operator '#' to de-reference object references: If `N` is the object reference of an object `O`, then `#N` is returning `O`.

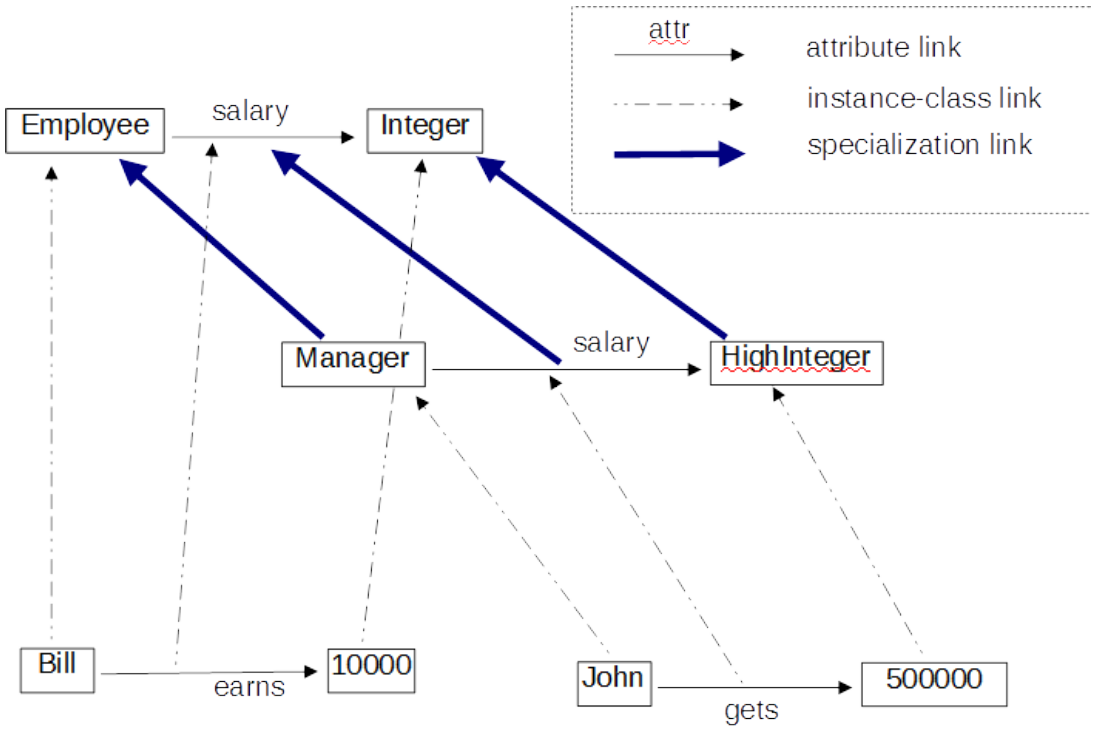

Figure 3.1: Graphical representation of Telos objects

Figure 3.1 shows a part of the objects defined so far. Node objects appear as nodes. Instance-class relationships appear as broken directed links (between the instance and the class). Specialization links are thick links from the subclass to the superclass. Attributes are labeled links from the object to the attribute value (which itself is an object). Note that the attribute `Manager!salary` is a subclass of the attribute `Employee!salary`. Hence, the attribute `John!gets` is also regarded as an instance of `Employee!salary` via deduction.

Our motivating example has as highest abstraction level a meta meta class `Node`. It uses an attribute category `attribute` to define a link `connectedTo`. This category had never been defined before. Indeed, we assume that a built-in object `Proposition` with an attribute '`attribute`' exists. It is defined as follows:

```
(Proposition in Proposition)
(Proposition attribute/attribute Proposition)
```

The first statement is indeed a logical consequence of the built-in axioms of Telos as shown subsequently. It makes however no difference to include the statement as factual data as well. All explicit objects are automatically regarded as instances of `Proposition`, another built-in axiom of Telos. Hence, any Telos object can at is instance of `Proposition` and can instantiate the attribute '`attribute`'.

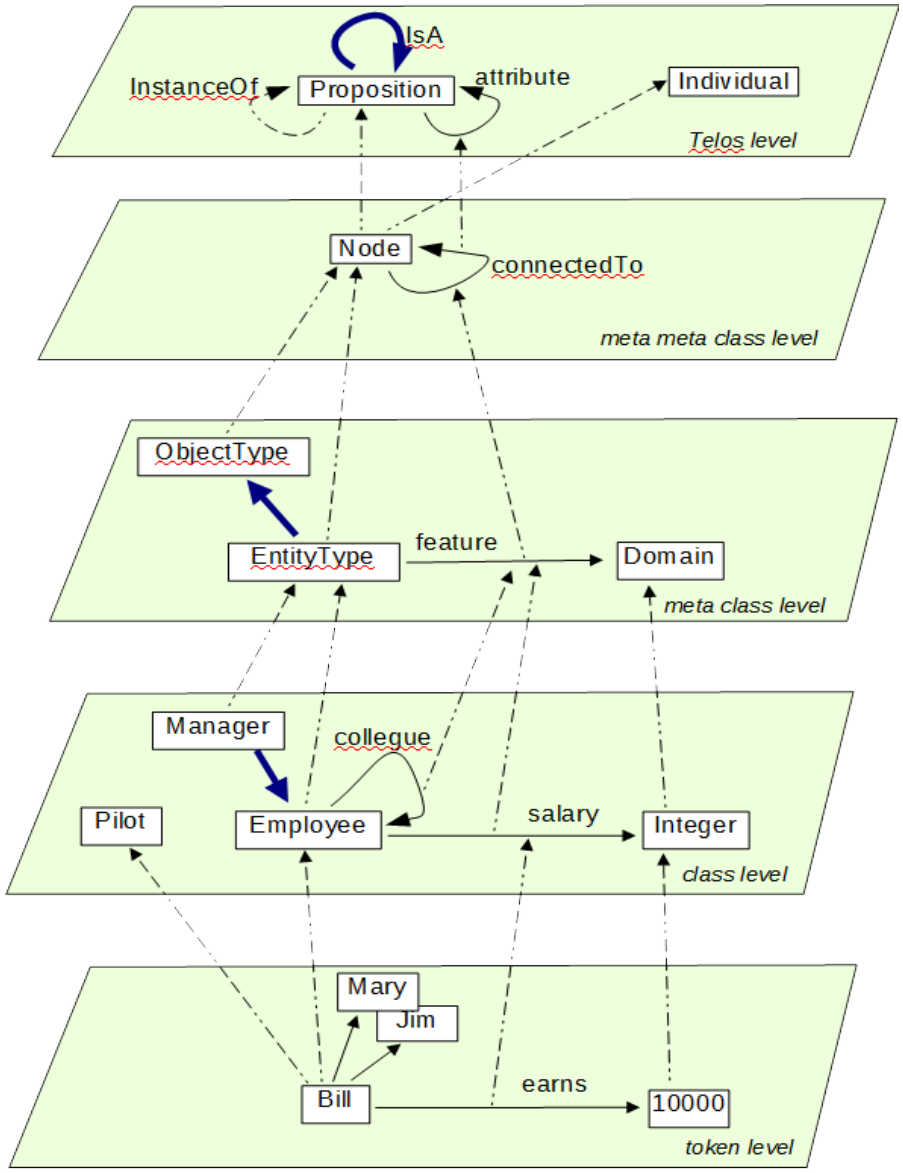

Figure 3.2: Abstraction levels in Telos

Figure 3.2 presents the motivating example in a graphical way. It should be observed that not all definitions of the running example are shown in the figure. For example, the instantiation of `Mary` and `Jim` into `Employee` is omitted. The membership of an object to a certain IRDS level is solely based on its current instantiation. `Bill`, `Mary`, `10000` etc. are regarded as tokens because they happen to have no instances. The object `Employee` is a (simple) class because all its instances are

tokens; the object `EntityClass` is a meta class because all its instances are simple classes and so on. From a logical point of view the levels are not relevant at all. Instead of maintaining "correct" assignment to levels, Telos enforces correct use of instantiation. Nevertheless, the IRDS levels shown in Figure 3.2 are useful for enhancing the understandability of models that are placed ate different IRDS abstraction levels.

There can well be relations other than instantiation between objects placed at different IRDS levels. Consider for example

```
(EntityType attribute/author PeterChen)
```

with the intended meaning "The Employee concept was authored by PeterChen". Here, the object `PeterChen` is naturally placed at the token level whereas the object `EntityType` is placed at the meta class level. It is important to note that the `EntityType` is just an object like any other object. It becomes a *meta class* just by the fact that is has instances that themselves have instances.

Another remark is on the definition order. ConceptBase places no restriction on the order in which objects are defined, i.e. created in the meta data repository. With the *instantiation strategy*, one starts top-down at the level of meta meta classes (called notation definition level in IRDS terminology), then define meta classes (establishing the constructs of modeling notation), and then continue with simple classes (example models), and tokens (example objects conforming to the models). In the *classification strategy*, one starts with token objects, then establishes classes to classify the tokens, then meta classes to classify the classes and so on. Mixed strategies are also possible and the most likely scenario since some classes emerge only when one finds example instance that cannot be classified into the existing set of classes. The only restriction imposed on object definition strategies are the Telos axioms stated in section 1.3.7. In particular, one can only create links between objects that have been defined before.

The top-most level in Figure 3.2 is reserved for five predefined objects of Telos. `Proposition` is the most general object and has any other object as instance including itself. `Individual` has all objects as instances that are displayed as nodes (i.e. not as attributes, instantiation, or attribution). The `InstanceOf` link of `Proposition` has all explicit instantiations as instance, for example

`Bill->Employee` and
`(Employee!salary->EntityType!feature)`.

The `IsA` link has all specializations as instance, for example `Manager=>Employee`. Finally, the `attribute` link has all attributes as instances. The precise definition of the five objects is presented in section 3.8.

3.4 The Telos frame syntax

The definitions in the preceding sections are utilizing the predicate notation that part of the query language of ConceptBase. The advantage of the logical representation is its preciseness. However, readability suffers from the fact that all information is decomposed into small pieces. The so-called frame syntax is therefore introduced. It provides a denser representation of object definitions by grouping all information regarding one object into one textual frame. In this section, we use the

motivating example to introduce it:

```
Bill in Employee,Pilot with
     salary
          earns: 10000
     colleague
          col1: Mary;
          col2: Jim
end

Mary in Employee end
Jim in Employee end

John in Manager with
     salary
          gets: 500000
end

500000 in HighInteger end
```

The attribute category salary precedes the attribute definitions. If more than one attribute fall under the same category, then a semi-colon separates them. A comma separates multiple classes of an object. Note that objects like `Mary` can exist without instantiating all or even any attribute category of their class.

```
Employee in EntityType with
     feature
          salary: Integer;
          colleague: Employee
end
Manager in EntityType isA Employee with
     feature
          salary: HighInteger
end

Pilot in EntityType end
Integer in Domain end
HighInteger in Domain isA Integer end
Employee in Domain end
```

The frame definition of `Employee` shows that the same syntax is applied for class definitions. Super classes are declared within the definition of the subclass. The subclass `Manager` in the example refines the attribute '`salary`' of `Employee`. In such cases the attribute value (`HighInteger`) at the subclass level has to be a subclass of the corresponding attribute value `Integer` of the superclass `Employee`. If a class has more than one super class, then commas separate them.

```
EntityType in Node isA ObjectType with
     connectedTo
          feature: DomainOrObjectType
end
```

```
ObjectType in Node isA DomainOrObjectType end
Domain in Node isA DomainOrObjectType end
DomainOrObjectType in Node end
```

We include a meta class `DomainOrObject` as a superclass of `Domain` and `ObjectType`. By this, an instance of `EntityType` can either have a domain as feature (e.g. the attribute '`salary`') or a reference to an object type (e.g., the '`colleague`' attribute). The example is completed by the definition of the meta meta class:

```
Node in Proposition with
     attribute
          connectedTo: Node
end
```

Figure 3.3 displays the example using the ConceptBase graph browser. The dotted links are instantiations, the thick links are specializations, and the thin links are ordinary attributes.

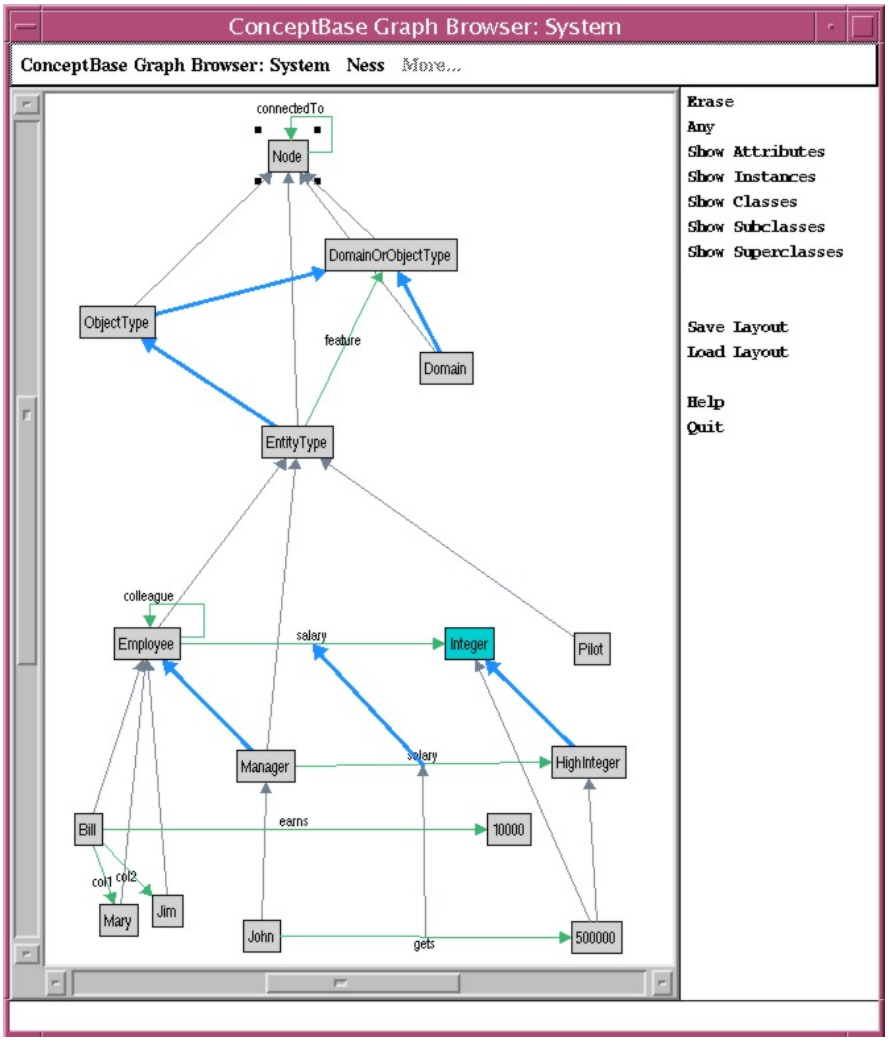

Figure 3.3: Display of the Telos example in the ConceptBase graph browser

3.5 A small primer in logic for databases

The implementation of Telos in ConceptBase is based on a simple logic, which is called Datalog. The advantage of this framework is that the semantics of a Telos model can be precisely defined. It is also the foundation of the expressive query language in ConceptBase and its extensibility at all abstraction level. To understand these features, a few basic concepts have to be understood.

We assume that the reader has some initial knowledge of the relational database model and of predicate logic. In Datalog, a database is defined as a pair `(DB,R)` where `DB` is set of so-called base relations and `R` is a set of so-called deductive rules. A base relation is referred to in logic by a predicate `R(x1,x2,...,xk)` as known in the domain calculus of relational database theory. A deductive rule has the form

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> S(x1,x2,...)
```

Different from standard Datalog notation, the logical quantifier '`forall`' is made explicit here[4]. The predicates R_i are positive literals of the condition; the predicates Q_i are negative literals. The number m may be zero, i.e. a deductive rule without a negative literal is allowed. The predicate `S` is the conclusion literal. There is exactly one conclusion literal. All variables are universally quantified. If a variable occurs in a negative literal of the condition it must also occur in a positive literal of the condition. Recursive rules are those deductive rules where the conclusion predicate also occurs in the condition.

The interpretation of a deductive database is based on Herbrand semantics. In this semantic, a constant like '`10`' is interpreted by itself. A *ground* (=variable free) occurrence of a predicate is also interpreted by itself. A rule with empty condition must be ground. Such a degenerated rule is also called a *fact*. We assume that all facts are in `DB`, i.e. `R` does not contain rules with empty conditions. Datalog only allows constants or variables as arguments of predicates. Then, the interpretation of a deductive database `(DB,R)` is computed as the smallest set of facts (the so-called *fixpoint*) fulfilling the following conditions If `R(c1,...,ck)` is a fact in a base relation of DB, then `R(c1,...,ck)` is in the fixpoint.

1. If there is a combination c1,c2,... of constants and a rule r

```
forall x1,x2,... R1(x11,...) and ... Rn(xn1,...) and
                 not Q1(y11,...) and ... not Qm(ym1,...)
==> S(x1,x2,...)
```

such that the substitution `s=[x1=c1,x2=c2,...]` yields facts `Ri[s]` that are already in the fixpoint and `Qj[s]` that are not in the fixpoint, then `S(c1,c2,...)` is also in the fixpoint.

A *substitution* is an operator that replaces variables in a formula by terms, in our case by constants. The fixpoint computation requires that recursive rules fulfill a so-called *stratification* condition. Stratification is an assignment of levels (numbers) to predicates. A predicate referring to a base relation gets the level 0. A conclusion predicate must be assigned a level that is greater or equal than the level of all positive literal occurrences in the condition and strictly greater than all negative

literal occurrences in the condition. If there is such a level assignment, then the rule set R is called stratified. One can prove that in this case there is a unique smallest fixpoint, the so-called *perfect model*.

An example of a non-stratifiable rule set consisting of just one rule is:

```
forall x P(x) and not Q(x) ==> Q(x)
```

Intuitively, non-stratifiable rule sets are expressing paradoxes[5]. Consider for this the definition of a set M to the set of all sets that do not contain themselves as elements. In logic, this is expressed as equivalence:

```
forall s Set(s) and not Element(s,s)
<==> Element(s,M)
```

We can read the formula as the conjunction of two implications

```
forall s Set(s) and not Element(s,s)
  ==> Element(s,M)

forall s Element(s,M) ==> Set(s) and not Element(s,s)
```

The first logical implication would is not stratifiable because the implied predicate occurs negatively in the condition.

Integrity constraints in deductive databases are special rules of the form

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> inconsistent
```

A deductive database `(DB,R)` is called consistent, if the fact '`inconsistent`' is not in its fixpoint, i.e. the fact '`inconsistent`' cannot be derived. The predicate '`inconsistent`' may not occur in a condition of any rule. We assume that an update to a base relation may not lead to inconsistency. If '`inconsistent`' is derivable after an update, the update is rejected and the database is rolled back to the state before the update.

Integrity constraints are useful to guarantee that certain database states are never reached because they represent errors. In modeling and design environments, we are sometimes less strict with integrity constraints. Rather than forbidding certain database states, we are interested in minimizing the number of violations with the goal of zero *violations* at the end of the modeling process. The rationale behind this viewpoint is that the database is incomplete at the start of the modeling process and the integrity constraint violations are removed when the database containing the models is becoming more and more complete. If we would use traditional integrity constraints, then the incomplete early database states would be rejected. This would be unacceptable in a modeling environment. A way out of this dilemma is to define ordinary deductive rules, which derive violations facts that formally are not regarded as integrity violations:

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> violator(x1,x2,...)
```

The current violations can be computed by querying the `violator` predicate. The solutions indicate to the modeler where the current database has to be extended or modified in order to reach fewer violations.

Example 1: Consider the following rule set and check it for satisfiability. The predicate R(.,.) refers to a base relation.

```
forall x P(x,y) and R(x,z) and not Q(x,z) ==> L(z)
forall x,y L(x) and L(y) ==> Q(x,y)
```

The base relation `R` gets the lowest stratification level:

```
strat(R)=0
```

The first rule induces the following conditions:

```
strat(P) ≤ strat(L)
strat(Q) < strat (L)
```

The second rule adds the condition

```
strat(L)  ≤ strat(Q)
```

Thus, there is no stratification for this rule set.

Example 2: Consider the rule set

```
forall x P(x,y) and R(x,z) and Q(x,z) ==> L(z)
forall x,y L(x) and L(y) ==> Q(x,y)
```

This rule set is stratifiable:

```
strat(R)=strat(P)=strat(L)=strat(Q)=0
```

Example 3: Let `move(.,.)` be a base relation. Consider the rule set consisting of a single rule

```
forall x,y move(x,y) and not win(y) ==> win(x)
```

This rule set is obviously not stratifiable because is would require `strat(win) < strat(win)`. The example is however interesting as is encodes ‘win’ positions of simple games. If the move database is acyclic, i.e. there is no cyclic path in the database matching

```
move(x1,x2),....,move(xk,x1)
```

then no fact `win(x)` is derived from the negation of itself. While the rule is not stratifiable, we still can compute a unique fixpoint. Hence, there are cases where the above static stratification test is stricter than necessary. The companion CD contains the ‘Win’ example in Telos and shows some further interesting applications. More on the theoretical foundations of Datalog and its extensions can be found in (Ceri, Gottlob, Tanca 1990) and (Chen, Warren 1996).

3.6 The logical foundation for Telos

The Telos object definitions are up to now just pieces of text in certain syntax. One could imagine that they are stored in a file and then looked up by a standard word processor. The Telos language only becomes useful if some automation is offered. In ConceptBase, this automation is primarily based on a logic-based facility that allows analyzing large sets of Telos definitions (also called a Telos model). This facility comes in three flavors:

1. A *constraint* expresses a necessary condition that has to be fulfilled by the Telos model. For example, one might define that no employee may earn more than her boss.
2. A *deductive rule* can be used to derive new predicate facts from existing. For example, the boss of an employee may be derived from the head of the department where the employee works.
3. A *query class* is syntactically a Telos class with a constraint definition. However, the constraint is not regarded as a necessary condition for being an instance of the class but as a sufficient condition.

We introduced Telos by the three predicates `(x in c)` for class abstraction, `(c isA d)` for generalization, and `(x m/l y)` for attribution. The clue of Telos is however that all these predicates are based in a single base predicate: the P-predicate or "proposition":

`P(o,x,l,y)`

It is used to represent all explicit Telos objects. The component o is called the object identifier, x is the source, l the label, and d the destination of the object. The P-predicate is used to define the already known predicates by the four subsequent axioms:

```
forall o,x,c P(o,x,in,c) ==> (x in c)

forall o,c,d P(o,c,isa,d) ==> (c isA d)

forall o,x,l,y,p,c,m,d P(o,x,l,y) and P(p,c,m,d) and (o in p)
    ==> (x m/l y)

forall x,m,l,y (x m/l y) ==> (x m y)
```

The formulas should be read as a deductive rule: If the condition before the implication sign is true for some substitution of variable, then the correspondingly substituted predicate behind the implication sign is true. We assume that the above deductive rules are predefined, i.e. they are axioms of the logical theory. We define that a Telos database is a triple `(OB,IC,R)` where

- `OB` $\subseteq$ `{P(o,x,l,y)|o,x,y object identifier,l label}` is the finite extensional database of Telos objects,
- `IC` is a finite set of integrity constraints, and
- `R` is a finite set of deductive rules.

An element of the `OB` database is called a *P-fact* or Telos object. Sometimes, we refer to an occurrence of `P(o,x,l,y)` as a *P-predicate*. Then, we assume that it occurs inside a logical formula where some parameters of the P-predicate can be logical variables.

The axioms are predefined entries in `IC` or `R` respectively. We demand that any object must be instance of some class. The object with name `Proposition` was already mentioned as predefined (in `OB`). Indeed, we demand at least the following five objects to be predefined:

`P(p1,p1,Proposition,p1)`

```
P(p2,p2,Individual,p2)
P(p3,p1,attribute,p1)
P(p4,p1,InstanceOf,p1)
P(p5,p1,IsA,p1)
```

For reasons of readability, we occasionally use expressions like `#Individual` to refer to the identifier of an object like `Individual`. The expression stands for the object identifier of `Individual`, here `p2`. The operator '#' is left out when its clear that we refer to the object with the given name. The purpose of the five predefined objects becomes clear with the following axioms (to be read as deductive rules):

```
forall o,x,l,y P(o,x,l,y) ==> (o in #Proposition)
forall o,l P(o,o,l,o) ==> (o in #Individual)
forall o,x,c P(o,x,in,c) ==> (o in #InstanceOf)
forall o,c,d P(o,c,isa,d) ==> (o in #IsA)

forall o,x,l,y P(o,x,l,y) and (o \== x)
 and (l \== in) and (l \== isa)
 ==> (o in #Proposition!attribute)
```

The five deductive rules ensure that we automatically know each object is an instance of the class `Proposition` and that the membership to the four other classes is based on the structure of the P-fact. We use where possible the reference of an object (e.g. `Proposition`) instead of the object identifier (`p1`) to enhance readability. Object references are precisely defined in section 3.3.

The predicate `(c isA d)` for subclasses is supposed to be reflexive and transitive. Moreover, we demand that any instance of a subclass is also an instance of any superclass of that subclass:

```
forall o (o in Proposition) ==> (o isA o)

forall c,d,e (c isA d) and (d isA e) ==> (c isA e)

forall x,o,c,d (x in c) and P(o,c,isa,d)
 ==>  (x in d)
```

Finally, we demand that instances of classes must use the attribute definitions of the class in a conformant way (attribute typing axiom):

```
forall o,x,l,y,p,c,m,d P(o,x,l,y) and P(p,c,m,d) and (o in p)
   ==> (x in c) and (y in d)
```

The last formula is a predefined constraint of `IC`. Consider as example the attribute `Bill!earns` which is an instance of `Employee!salary`. Then, the constraint forces that `Bill` is an instance of `Employee` and `10000` (the attribute value of `Bill!earns`) is an instance of `Integer` (the attribute value of `Employee!salary`).

There are more built-in constraints on attribute specialization and on preventing mal-formed P-facts. They are left out here since they are mainly of technical nature. The complete list of axioms is presented in section 3.8. All axioms can be transformed into Datalog with stratified negation.

3.7 From frames to objects and vice versa

The frame syntax presented before is the standard way to express textual definitions of Telos objects. A frame indeed clusters class membership, superclasses and attributes into one textual object. In order to understand the precise meaning, one has to define the mapping of such frames to P-predicates. This is necessary, because the semantics of relationships like specialization is defined in terms of P-predicates! As an example, we consider the frame

```
EntityType in Node isA ObjectType with
     connectedTo
          feature: Domain
end
```

In a first step, these frames are re-written into the flat predicate facts for classification, specialization, and attribution. It should be noted that the facts establish binary relationships between named objects. The attribution is a special case since it has two labels as infix symbols: first the label of the attribute itself, second the label of the *attribute category* of the attribute.

```
(EntityType in Node)
(EntityType isA ObjectType)
(EntityType connectedTo/feature Domain)
```

In the second step, consider each predicate fact individually and look for an axiom in R whose conclusion predicate matches the predicate fact. We only consider the following three axioms:

```
D1. forall o,x,c P(o,x,in,y) ==> (x in c)

D2. forall o,c,d P(o,c,isa,d) ==> (c isA d)

D3. forall o,x,l,y,p,c,m,d P(o,x,l,y) and
    P(p,c,m,d) and (o in p) ==> (x m/l y)
```

Before mapping a predicate fact to a P-fact, one has to check whether a fact like `(EntityType in Node)` can already be derived. It could for example be derivable via the class membership axiom of the specialization relationship. If not, we match the fact `(EntityType in Node)` to the conclusion predicate `(x in c)` of rule D1 leading to a variable substitution `[x=EntityType,c=Node]`. In order to make `(EntityType in Node)` deducible via rule D1, the predicate `P(o,#EntityType,in,#Node)` must be true for some filler of variable `o`. If the database OB already contains such an object, then nothing has to be done. If not, then a new object identifier like `o1` has to be generated and the object `P(o101,#EntityType,in,#Node)` can be inserted into OB. In the same way, `(EntityType isA ObjectType)` is mapped to an object `P(o102,#EntityType,isa,#ObjectType)`.

The third fact `(EntityType connectedTo/feature Domain)` is an attribute definition. It is matched against rule D1 leading to a substitution `[x=EntityType,m=connectedTo,y=Domain,l=feature]`. Its mapping is slightly more complex because the attribute-typing axiom demands us to exclude attributes that are not conforming

the attribute definition at the class level. So, for the partially substituted predicate `P(p,c,connectedTo,d)` the predicates `(EntityType in c)` and `(Domain in d)` must be derivable. Consequently, we can limit the search for `P(p,c,connectedTo,d)` accordingly. Theoretically, more that one such object may exist since the same object `x` can be instance of multiple classes. However, the built-in Telos axiom A17 (see section 3.8) prevents such ambiguity. Once, the fillers for the variables in `P(p,c,connectedTo,d)` are looked up, one can insert a new object `P(o103,#EntityType,feature,#Domain)` with `o103` substituting the variable `o`. Given the current database state, the resulting substitution is `[p=o11,c=Node,d=Node].` It remains the obligation to make `(o103 in o11)` true. Since `o11` is a constant, we can use the first rule D1 to do so. Hence, the resulting Telos objects are:

```
P(o101,#EntityType,in,#Node)
P(o102,#EntityType,isa,#ObjectType)
P(o103,#EntityType,feature,#Domain)
P(o104,#EntityType!feature,in,#Node!connectedTo)
```

It should be noted that all object references are resolved on the fly against the object identifiers. For example, the reference `EntityType!feature` is resolved to `o103`. This replacement is for the sake of efficiency and uniformity. The P-fact `o104` shows how attribute categories are treated in Telos. They are mapped to instantiation relationships between an attribute object and another attribute object. After replacing object references by identifiers, the stored P-facts will be

```
P(o101,o12,in,o11)
P(o102,o12,isa,o13)
P(o103,o12,feature,o14)
P(o104,o103,in,o15)
```

where we assume the existence of

```
P(o11,o11,Node,o11)
P(o12,o12,EntityType,o12)
P(o13,o13,ObjectType,o13)
P(o14,o14,Domain,o14)
P(o15,o11,connectedTo,o11)
```

The mapping from Telos frames to objects is directly usable for implementing the 'insert' operation, i.e. adding new objects to the Telos database. A similar method can be employed when one wants to remove a Telos frame from the database. First, the frame is mapped to P-facts, and then the P-facts are removed from the database.

A side remark: Since not all axioms of Telos are discussed here, the complete method to translate a Telos frame objects is a bit more complex. For example, a subclass may refine an attribute that is already defined at the superclass like the `Manager!salary` attribute. In such a case, an instance of `Manager` would instantiate the more special attribute whereas an instance of `Employee` would instantiate the more general attribute. The theoretic framework for mapping method is called *abduction*. Abduction is an approach where a user can specify which derived predicates shall be

inserted (i.e. should be derivable after an update) and which shall be deleted (i.e. no longer derivable after the update). Abduction has to cope with the problem that for the same predicate multiple deductive rules can exist. Hence, the mapping is in principal ambiguous and one has to develop notions of minimality to prioritize the rule selection. In our case, we do only consider the three 'basic' rules for the instantiation, attribution, and specialization predicates.

It should be noted that the mapping method allows to 'insert' Telos frames where certain attributes are already derivable from the database. In such cases, no new objects are inserted. Because of this we call the method TELL and the reverse method UNTELL. User of the ConceptBase system can display the P-facts generated (removed) for a TELL (UNTELL) operation by setting the parameter trace mode to 'veryhigh'. See ConceptBase user manual (Jarke, Jeusfeld, Quix 2003) for instructions.

The reification of all Telos statements is the precondition for using the Telos for method engineering, in particular for defining the *semantics* of certain symbols in modeling notations. The treatments of attributes as ordinary objects allows to formulate rules and constraints for such objects regardless whether they are at data level, model level, notation level, or an even higher level. The subsequent list of Telos axioms effectively defines the semantics of the three Telos relationship types: instantiation, specialization, and attribution. User-defined relationship types and the definition of their semantics are presented afterwards.

3.8 List of Telos predicates and axioms

Telos has a single base relation, the P-facts, on which further predicates are defined via logical axioms. Some of these axioms are deductive rules; others are constraints, which forbid certain combinations of P-facts to occur. The list of Telos predicates is as follows:

`P(o,x,n,y)`
: This predicate ranges directly over the P-facts. We say: there is a P-fact identified by o which links two P-facts identified by x,y, the link is labeled by n.

`From(o,x)`
: There is a P-fact identified by o whose second argument is x (the *source* of object o).

`Label(o,n)`
: There is a P-fact whose third component is n (the *label* of object o).

`To(o,y)`
: There is a P-fact identified by o whose fourth argument is y (the *destination* of object o).

`(x in c)` or `In(x,c)`
: The object x is an instance of class c, or in other words, c is a class of object x.

`(c isA d)` or `Isa(x,c)`
: The class c is a subclass of the class d, or in other words, the class d is a superclass of class c.

`(x m/n y)` or `AL(x,m,n,y)`
: There are two objects x,y that are linked to each other by a P-fact that has label n. That P-fact is an instance of another object, which has the label m. We say that m is the category of the link between x and y.

`(x m y)` or `A(x,m,y)`
: There are two objects x and y that are related to each other by a binary relationship m.

`Aid(x,m,o)`
: There is a P-fact o which has x as its source and has category m.

`(x < y)`, `(x > y)`, `(x = y)`, `(x =< y)`, `(x >= y)`
: Numerical comparison between objects x and y. Both objects must be instances of the built-in classes `Integer` or `Real`.

`(x == y)` or `IDENTICAL(x,y)` or `UNIFIES(x,y)`
: The two objects x and y are the same.

Around 30 rules and constraints are predefined in the variant of Telos presented here (Jeusfeld 1992). To distinguish this axiomatization of Telos from earlier Telos definitions, we sometimes refer to it is *O-Telos*. The O-Telos axioms precisely define what we understand by instantiation, specialization, and

attribution. Furthermore, they define the extension (= logical interpretation) of some of the predicates listed above.

Axiom A1: Identifiers of P-facts are unique, or in other words the first field of P facts is a key for the rest.

```
forall o,x1,n1,y1,x2,n2,y2
       P(o,x1,n1,y1) and P(o,x2,n2,y2) ==>
      (x1=x2) and (n1=n2) and (y1=y2)
```

Axiom A2: The name of an individual object is unique.

```
forall o1,o2,n
P(o1,o1,n,o1) and P(o2,o2,n,o2) ==> (o1=o2)
```

Axiom A3: Names of attributes are unique in conjunction with the source object. Or in other words: no object may have two attributes with the same name. This does not necessarily hold for instantiation and specialization objects (see axiom A4)

```
forall o1,x,n,y1,o2,y2
P(o1,x,n,y1) and P(o2,x,n,y2)
==> (o1=o2) or (n=in) or (n=isa)
```

Axiom A4: The name of instantiation and specialization objects (labels *in*, *isa*) is unique in conjunction with source and destination objects.

```
forall o1,x,n,y,o2 P(o1,x,n,y) and P(o2,x,l,y) and ((n=in) or
(n=isa)) ==> (o1=o2)
```

Axiom A5: Instantiation objects lead to solutions for the `In` predicate.

```
forall o,x,c P(o,x,in,c) ==> In(x,c)
```

Axiom A6: Specialization objects induce a specialization relationship `Isa` between the two referenced objects.

```
forall o,c,d P(o,c,isa,d) ==> Isa(c,d)
```

Axiom A7: If there is an attribute with name `n` between two objects `x` and `y`, and this attribute is an instance of an attribute class with name `m`, then a solution for the `AL` predicate can be derived:

```
forall o,x,n,y,p,c,m,d P(o,x,n,y) and P(p,c,m,d) and In(o,p)
==>AL(x,m,n,y)
```

Axiom A8: The ordinary attribute predicate `A` is based on the `AL` predicate by omitting the attribute label `n`.

```
forall x,m,n,y AL(x,m,n,y) ==> A(x,m,y)
```

Axiom 9: An object x may not neglect an attribute definition in one of its classes. If some attribute predicate can be derived for x, then it will be due to an instantiation of an attribute category defined at class level of x. Note that solutions for the A predicate can only be obtained from the Telos axioms. User-defined rules are not interfering with this axioms since their attribution predicates are formally distinguished from the A predicate.

```
forall x,y,p,c,m,d In(x,c) and A(x,m,y) and
P(p,c,m,d) ==> exists o,n P(o,x,n,y) and In(o,p)
```

Axiom A10: The *isa* relation is reflexive. The object identifier #Proposition (=p1) is declared in axiom A24.

```
forall c In(c,#Proposition) ==> Isa(c,c)
```

Axiom A11: The *isa* relation is transitive.

```
forall c,d,e Isa(c,d) and Isa(d,e) ==> Isa(c,e)
```

Axiom A12: The *isa* relation is anti-symmetric.

```
forall c,d Isa(c,d) and Isa(d,c) ==> (c==d)
```

Axiom A13: Class membership of objects is inherited upwardly to the superclasses. This is the only 'inheritance rule' in Telos. Inheritance of attributes to subclasses is a redundant principle that is subsumed via axioms A13 and A14: any instance of a subclass is also instance of the superclasses (axiom A13) and thus can instantiate the attributes of the superclasses (axiom A14).

```
forall p,x,c,d In(x,d) and P(p,d,isa,c) ==> In(x,c)
```

Axiom A14: Instance attributes are "typed" by attributes defined at class level.

```
forall o,x,l,y,p P(o,x,l,y) and In(o,p)
==> exists c,m,d P(p,c,m,d) and In(x,c) and In(y,d)
```

Axiom A15: Subclasses which define attributes with the same name as attributes of their superclasses must refine these attributes. Hence, the attribute definition of a superclass is never violated by a refinement of that attribute at subclass level. Specifically, the refined attribute is a specialization of the superclass attribute and the attribute value of the refined attribute is a specialization of its counterpart at superclass level.

```
forall c,d,a1,a2,m,e,f Isa(d,c) and P(a1,c,m,e) and P(a2,d,m,f)
==> Isa(f,e) and Isa(a2,a1)
```

Axiom A16: If an attribute is a refinement (subclass) of another attribute then it must also refine the source and destination components. Note that the two attributes do not necessarily have the same label as was the case in axiom A15. However, all refinements cf. axiom A15 are also governed by axiom A16 because of the implied truth of `Isa(a2,a1)` in axiom A15.

```
forall c,d,a1,a2,m1,m2,e,f
```

```
Isa(a2,a1) and P(a1,c,m1,e) and P(a2,d,m2,f) ==> Isa(d,c) and
Isa(f,e)
```

Axiom A17: For any object there is always a unique "smallest" attribute class with a given label `m`. Hence, whenever two different attribute with same label are applicable to be instantiated by object `x`, then there is a third attribute `a3` which is specializing both other attributes `a1`, `a2`.

```
forall x,m,y,c,d,a1,a2,e,f
In(x,c) and In(x,d) and P(a1,c,m,e) and P(a2,d,m,f)
==> exists g,a3,h In(x,g) and P(a3,g,m,h) and
      Isa(g,c) and Isa(g,d)
```

Axioms A18: Any object is an instance of the class `Proposition`. Anu instance of `Proposition` is an object.

```
forall o,x,n,y P(o,x,n,y) <==> In(o,#Proposition)
```

Axioms A19: Any individual object is an instance of the class `Individual`. Any such instance must be an individual object. The operator '`\==`'stands for inequality of object identifiers. Note that an individual object is determined by its structure, i.e. identifier, source and destination are the same.

```
forall o,l  P(o,o,n,o) and not  (n \== in) and
not  (n \== isa)  <==> In(o,#Individual)
```

Axioms A20: All instantiation objects are instance of the `InstanceOf` attribute of `Proposition`. Any such instance must be an instantiation object.

```
forall o,x,c P(o,x,in,c) and (o \== x) and
 (o \== c)  <==> In(o,#Proposition!InstanceOf)
```

Axioms A21: All specialization objects are instance of the `InstanceOf` attribute of `Proposition`. Any such instance must be a specialization object.

```
forall o,c,d P(o,c,isa,d) and (o \== c) and
 (o \== d) <==> In(o,#Proposition!IsA)
```

Axioms A22: All attribute objects are instance of the attribute named `attribute` of `Proposition`. Any such instance must be an attribute object.

```
forall o,x,n,y P(o,x,l,y) and (o \== x) and
 (o  \==  y)  and  (n  \==  in)  and  (n  \==  isa)  <==>
In(o,#Proposition!attribute)
```

Axiom 23: Any object, i.e. any P-fact, is either an individual object, an instantiation relationship, a specialization relationship, or a regular attribute. Note that the axioms A22 to A23 exclude P-facts like `P(o,o,n,p)` and `P(o,p,n,o)`.

```
forall o In(o,#Proposition) ==>
In(o,#Individual) or In(o,#Proposition!InstanceOf)
 or In(o,#Proposition!IsA)
 or In(o,#Proposition!attribute)
```

Axioms A24-A28: There are exactly five built-in classes. Note that these are the Telos built-in classes. ConceptBase has a lot more pre-defined classes to manage itself and to provide data structures for its functionality, in particular for user-defined rules, constraints, and queries. The object identifiers p1 to p5 are arbitrary. Any other set of five different identifiers will also work.

```
P(p1,p1,Proposition,p1)
P(p2,p2,Individual,p2)
P(p3,p1,attribute,p1)
P(p4,p1,InstanceOf,p1)
P(p5,p1,IsA,p1)
```

Axiom A29: Objects must be known before they are referenced. The operator '`*prec`' is some predefined total order on the set of object identifiers.

```
forall o,x,n,y P(o,x,n,y) ==>
(x *prec o) and (y *prec o)
```

Axiom schema A30: Let `P(p,c,m,d)` be an arbitrary object. Then, any binary instantiation predicate `In(o,p)` leads to a solution for its unary version `In.p(o)`.

```
forall o In(o,p) ==>  In.p(o)
```

Axiom schema A31: Let `P(p,c,m,d)` again be an arbitrary object. An attribute between `x` and y that is an instance of object `p` induces a binary attribute predicate `A.p` .

```
forall o,x,l,y P(o,x,l,y) and In(o,p) ==> A.p(x,y)
```

The last two axiom schemas are distinguishing the `In` and `A` predicates as defined by the Telos axioms from those declared in user-defined rules and constraints. User-defined rules are always transformed into versions that use `In.c` and `A.p` predicates instead of `In` and `A` predicates. By this transformation, the definition of `In` and `A` cannot be altered by user-defined rules.

We will further restrict user-defined rules to `In.c` and `A.p` predicates in their conclusion part. This completely shields the axioms from any further definition made by a user. For sake of readability, the user-defined formulas use predicates in their original syntax like

`In(x,c)` or `(x in c)`

`A(x,m,y)` or `(x m y)`

Internally, all such occurrences are replaced by predicates `In.c` or `A.p`, respectively. Thus, internally each class object and each attribute category has its own predicate symbol. Without this transformation, only few rule sets would be stratifiable due to the small number of predicate symbols.

The stratification explained in section 3.5 is based on predicate names, i.e. independently from the arguments of a predicate occurrence in a formula. This type of stratification is also called *static stratification*. *Dynamic stratification* extends the principle to predicate occurrences including their arguments. Like its static counterpart, dynamic stratification guarantees perfect model semantics.

Any statically stratified Datalog theory is also dynamically stratified but not vice versa. For the purpose of this book, it is sufficient to assume static stratification. We like to mention however that ConceptBase supports dynamic stratification. It is tested at the time when a formula is evaluated rather than when it is defined. The accompanying CD-ROM contains some examples of dynamically stratified Telos models as well as examples of Telos models that are not dynamically stratified.

The axioms A1 to A31 can be represented in Datalog either as deductive rules or integrity constraints or a combination of both. For example, the direction left-to-right of axioms A18 to A22 should be represented as a deductive rule, whereas the direction right-to-left should be represented as an integrity rule.

One can argue why we have chosen for the 31 axioms and axiom schemas. The majority of the axioms are about the interpretation of the three abstraction principles classification (instantiation), generalization (specialization), and attribution. These three principles occur so frequently in modeling that a pre-definition makes sense. As an alternative, one can start with an empty list of axioms and regard them all as user-defined. The advantage is an even greater flexibility at the expense of interference between domain-specific rules and domain-independent abstraction principles. For example, a user-defined rule could instantiate an individual object to the attribute object of Proposition. This would however destroy the intended interpretation of attribute and individual objects as links and nodes. Hence, the reason for having pre-defined axioms is to start from a well-understood set of abstraction principles that are protected via axioms A30 and A31 against unwanted interference thru user-defined rules.

Section 3.13 will present a technique to define more abstraction principles via so-called meta-level formulas. In fact, a method engineer can define her own brand of attribution, instantiation, and specialization by creating meta-level formulas. Instead of using the labels `attribute`, `in` and `isA`, she would define new attribute categories like `myAttribute`, `myIn`, and `myIsA` then constrain their interpretation by user-defined rules.

3.9 User-defined constraints and rules in Telos

The previous section introduced Telos as a deductive database with a single base relation (the P-facts) and some deduction rules and integrity constraints. The three predicates `(x in c)`, `(x m/l y)` and `(c isA d)` were defined in terms of the P-facts by predefined deductive rules. Some predefined integrity constraints define which Telos databases are regarded consistent. In this section, the logical language of Telos (as implemented in ConceptBase) is presented. It allows formulating deductive rules and integrity constraints at any abstraction level. ConceptBase provides a predefined object `Class`, which offers the two attributes '`rule`' and '`constraint`'. These attribute categories allow to specify user-defined rules and constraints:

```
Class in Proposition with
     attribute
          rule: MSFOLrule;
          constraint: MSFOLconstraint
end
```

The acronym MSFOL stands for *many-sorted first order logic*: all variables are typed by a class name. The formulas ware well-formed expressions over the predicates defined earlier. As a syntactic abbreviation, quantified variables are assigned to *class ranges*:

```
forall x/C F
```
is an abbreviation for
```
forall x (x in C) ==> F
```

```
exists x/C F
```
is an abbreviation for
```
exists x (x in C) and F
```

In ConceptBase, logical formulas are included between two dollar ('$') signs. The example can now be continued. We assume that a *constraint* on the lower bound of salaries shall be formulated:

```
Employee in Class with
     constraint
          c1: $ forall e/Employee s/Integer
                    (e salary s) ==> (s > 1500)$
end
```

The reader should be aware that Telos frames are incremental. The above frame has to be understood as additional information about the object `Employee`. More precisely, the object is made an instance of `Class` and one constraint with label ´c1´ is added[6].

Deductive rules are employed to derive new attribute relationship `(x m y)` or class memberships `(x in c)` from the database. As an example, we model `Department` as a class that can have subordinate departments. A deductive rule is then used to define when a department has another department as its part:

```
Department in EntityType,Class with
     feature
          directSubordinate: Department;
          subordinate: Department
     rule
          r1: $ forall d1,d2/Department
                    (d1 directSubordinate d2)
               ==> (d1 subordinate d2) $;
          r2: $ forall d1,d2/Department
                    (exists d/Department
               (d1 directSubordinate d) and
               (d subordinate d2))
               ==> (d1 subordinate d2) $
end
```

The two deductive rules effectively realize the transitive closure of the ´directSubordinate‘ relation. The derived predicate ´subordinate´ can be used in other logical formulas, e.g.:

```
Department with
     constraint
          c1: $ forall d1,d2/Department
                    (d1 subordinate d2) ==>
                    not (d2 subordinate d1) $
end
```

[source: Logic-1.sml]

Note that attribute labels are local to objects. Thus, the attribute `Employee!c1` is distinguished from the attribute `Department!c1`. The logical expressions in constraints and deductive rules can be arbitrarily nested using the logical operators '`and`', '`or`', '`not`', '`forall`', '`exists`', and '`==>`'. The predicates must be correctly typed, i.e. an attribute expression `(x m y)` or `(x m/l y)` is only allowed if the class of `x` has an attribute with label `m`. For example, the constraint `Department!c1` is valid because the variable `d1` is assigned to class `Department` which has an attribute '`subordinate`'.

3.10 Query classes

The last and most flexible incarnation of logical expressions in ConceptBase are the so-called *query classes*. A query class resembles an ordinary class with a constraint definition. The constraint is however interpreted as a sufficient condition for class membership: all instances that match the query class definition and fulfill the constraint are regarded as instances of the query class. As an example, we first consider a class definition of `Department,` which has a '`head`' attribute and a constraint that each class must have a manager:

```
DepartmentWithBoss in EntityType,Class
                  isA Department with
     feature
          head: Manager
     constraint
          c2: $ forall d/DepartmentWithBoss
                exists m/Manager (d head m) $
end
```

An attempt to create such a class with an instance without filling the '`head`' attribute like '`dept1`' will fail. The second instance '`dept2`' is fulfilling the constraint.

```
dept1 in DepartmentWithBoss
end

dept2 in DepartmentWithBoss with
     head
          manager: John
end
```

The idea of a query class is to let objects like '`dept1`' and '`dept2`' be instance of the superclass (`Department`) and to compute the class membership to the query class:

```
Department with
     feature
          head: Manager
end

DepartmentWithBossQ in QueryClass isA Department with
     constraint
          c2: $ exists m/Manager (~this head m) $
end
```

[source: Logic-2.sml]

The special variable '`~this`' denotes any instance object of the query class that fulfills the constraint. It can be regarded as an implicit universal quantification[7]:

```
DepartmentWithBossQ isA Department with
     constraint
          c1: $ forall this/Department
                  exists m/Manager (this head m) $
end
```

In many cases, it is useful to compute those objects that violate a constraint like `DepartmentWithBoss!c2`. A query class with a negated constraint definition serves this purpose. In our example, we are interested in those departments that do not have a head:

```
DepartmentWithoutBossQ in QueryClass isA Department with
     constraint
       c2: $ not exists m/Manager (~this head m) $
end
```

Department '`dept1`' is an instance of the query class `DepartmentWithoutBossQ` while '`dept2`' is an instance of `DepartmentWithBossQ`. The classification of objects into query classes is a deductive computation. The class membership rule of `DepartmentWithoutBossQ` would be:

```
forall this/Department
     (not exists m/Manager (this head m))
==> (this in DepartmentWithoutBossQ)
```

ConceptBase generates internally such a rule in its implementation of query classes. Since the classification predicate '`(x in c)`' is used as the conclusion, query classes can be referred to like an ordinary class in other query classes or even inside itself. The following example shows the reference to a query class as a superclass and as a constant within the constraint.

```
TopSalary in QueryClass isA HighInteger with
     constraint
          c3: $ exists e/Employee (e salary ~this)
               and (~this > 1000000) $
end

TopDepartmentQ in QueryClass
isA DepartmentWithBossQ with
     constraint
          c2: $ exists m/Manager s/TopSalary
               (~this head m) and (m salary s) $
end
```

Query classes can be considered as classes that offer a method 'ask': whenever the method is called, those objects are returned that fulfill the constraint of the query class. Like other Telos classes, a query class can have more than one superclass:

```
Academic in EntityType end

AcademicEmployee in QueryClass
isA Employee,Academic end
```

The interpretation of this frame is that any instance of `AcademicEmployee` must be both an instance of `Employee` and `Academic`, or logically:

```
forall x (x in Employee) and (x in Academic) ==> (x in
AcademicEmployee)
```

The rule is *deducing* the instances of the query class. Such a deductive rule deriving class membership is generated for each query class. The logical representation allows query classes to appear anywhere where classes can appear. For example, the class `Academic` can itself be a query class. If a query class has some constraint, then its logical representation is added to the conjunction of the precondition in the above rule:

```
RichAcademicEmployee in QueryClass
   isA AcademicEmployee with
     constraint
       c: $ exists s/TopSalary (~this salary s) $
end
```

[source: Logic-2.sml]

The class membership rule is in this case:

```
forall x (x in AcademicEmployee) and (exists s/TopSalary (x salary
s))
==> (x in RichAcademicEmployee)
```

ConceptBase compiles the class membership rule from the query class. If `Q` is a query class, then all objects `x` for which `(x in Q)` is true are called *answer objects* of `Q`, or simply instances of `Q`. Besides the simple class membership, a query class can also specify *answer attributes*. These are either attributes that the answer object has in the database (so-called retrieved attributes) are they are attached to the answer object by a condition formulated in the query class constraint:

3.11 Attributes and parameters in queries

So far, query classes are just capable to express the instantiation of an object in their answer set. In practical application, one needs to retrieve properties of those answer objects as well. Query classes provide two attribute categories for this. *Retrieved attributes* are attributes that an object has independent from the query class definition. *Computed attributes* are properties that are attached to the answer object via the query class definition. The subsequent definition shows a typical use of retrieved attributes:

```
RichAcademicEmployee1 in QueryClass
  isA AcademicEmployee with
     retrieved_attribute
          salary: Integer
     constraint
       c: $ exists s/TopSalary (~this salary s) $
end
```

ConceptBase generates a *query rule* for the retrieved attribute:

```
forall x,s  (x in RichAcademicEmployee1) and (s in Integer) and (x
salary s) and ((s in TopSalary) and (x salary s))
==> Q(RichAcademicEmployee1,x,salary,s)
```

The class membership rule is derived from this rule:

```
forall x,s  Q(RichAcademicEmployee1,x,salary,s)
==> (x in RichAcademicEmployee1)
```

A side effect of this method is, that an object `x` that has no salary, i.e. no filler for the retrieved attribute, is not in the answer set of the query class. So retrieved attributes are *necessary*. They are not necessarily single-valued however: an employee may for example have more than two salaries. Both would be derived by the complex query rule. The destination of the retrieved attribute may be a specialization of the attribute defined at the direct or indirect super class of `RichAcademicEmployee`. In our case, the salary attribute is defined for the class `Employee` and has the destination type `Integer`. Since we defined a subclass `TopSalary` of `Integer`, we can formulate a new version of the query:

```
RichAcademicEmployee2 in QueryClass
  isA AcademicEmployee with
     retrieved_attribute
          salary: TopSalary
end
```

[source: Logic-3.sml]

This equivalent query definition no longer requires the constraint of the previous version. By co-incidence, the class `TopSalary` is also a query class. Since the class membership rule for this query yields instantiations

`(s in TopSalary)`, this is a correct and feasible way. The query rule is in this case:

```
forall x,s  (x in RichAcademicEmployee2) and (s in TopSalary) and (x
salary s)
 ==> Q(RichAcademicEmployee2,x,salary,s)
```

The reader can verify that it is logically equivalent to the previous version. In general, a query class can have any number of retrieved attributes. The use of retrieved attributes can be compared to a

projection in classical relational algebra: only the required attributes are returned in the answer. The difference to relational algebra is that retrieved attributes can be defined at any superclass of the query class. They may be derived by a deductive rule:

```
DepartmentWithBossQ1 in QueryClass
  isA Department with
     retrieved_attribute
          head: Manager;
          subordinate: Department
     constraint
          c2: $ exists m/Manager
                  (~this head m) and
                  not (m head ~subordinate) $
end
```

The query returns department with their heads and subordinate departments where the head is not head of the subordinate department.

The second type of answer attributes is the *computed attribute*. A computed attribute is derived within the query constraint. As an example, consider the '`head`' attribute of `Department`. Instead of attaching the '`head`' in `DepartmentWithBossQ1` to departments, one can formulate a query that returns instances of `Manager` together with the department they are head of:

```
BossWithDeptQ in QueryClass isA Manager with
     computed_attribute
          dept: Department
     constraint
          c2: $ (~dept head ~this) and
                 not (exists upper/Department
                   (upper subordinate ~dept)) $
end
```

The second part of the constraint expresses that only those managers shall be returned which head a department that is not subordinate of an upper level department. Only those managers shall be returned which are heads of at least one department. This is due to the fact that the expression '`~dept`' stands for an existentially quantified variable in the constraint:

```
exists ~dept/Department (~dept head ~this) and
            not (exists upper/Department
                  (upper subordinate ~dept))
```

The construct for computed attributes is per se redundant. Deductive rules have the expressive power to deduce them as well. However, deductive rules are not allowed in a query definition but only for ordinary classes. We show the equivalent deductive rule for the sake of clarification:

```
Manager in Class with
     attribute
          dept: Department
     rule
          deptrule: $ forall m/Manager d/Department
```

```
                    (d head m) ==> (m dept d) $
end
```

Note that the dept attribute is not using the attribute category ‘`feature`’ but rather the predefined category '`attribute`' that is available to all Telos objects. Moreover, the object `Manager` is classified into two classes: `EntityType` (see above) and `Class`. The latter provides the attribute category ‘`rule`’ which is instantiated by deptrule, more precisely `(Manager!deptrule in Class!rule)`. Assuming that '`deptrule`' is defined, the query class is expressed as follows:

```
BossWithDeptQ1 in QueryClass isA Manager with
     retrieved_attribute
          dept: Department
     constraint
          c2: $ (~this dept ~dept) and
                 not (exists upper/Department
                      (upper subordinate ~dept)) $
end
```

The solution with the computed attribute is preferable here since it does not require extending a class by an attribute plus a deductive rule.

The last remaining option for query classes is parameterization. A so-called *generic query class* can contain parameter definitions. Logically, a parameter definition introduces an existentially quantified variable in the query constraint. When calling a query, a user can supply values for parameters (parameter instantiation) or restrict a parameter to some subclass of its original range (parameter specialization). As an example, consider the definition of the class `EntityType`. It mentions an attribute '`feature`' with `Domain` as value. Assume we are interested in getting the list of all instances of `EntityType` that have some instance of Domain as feature. The filler for `whatDomain` shall be provided not at query definition time but at query call time. The generic query class is then:

```
EntityTypeByDomainQ in GenericQueryClass
  isA EntityType with
     parameter
          whatDomain: Domain
     constraint
          c2: $ (~this feature ~whatDomain) $
end
```

The constraint expresses that only those instances of `EntityType` are returned that have a feature matching the parameter '`whatDomain`'. The parameter is a shortcut for a constraint formula that has an existentially quantified variable for it:

```
exists ~whatDomain/Domain
   (~this feature ~whatDomain)
```

The query rule for a generic query class is built like for ordinary query classes. The parameter

becomes a variable in the conclusion predicate:

```
forall x,w (x in EntityType) and (w in Domain) and  (x feature w) ==>
Q(EntityByDomainQ,x,whatDomain,w)
```

Side remark on quantification: The query rule has a universal quantification for the parameter variable `w`. This is consistent with the existential quantification of the parameter `~whatDomain` in the constraint since the constraint is part of the condition of the rule. Consider the general example of a deductive rule:

```
forall x,y A(x,y) ==> B(x)
```

This is equivalent to the version where the exists is pushed into the condition:

```
forall x (exists y A(x,y)) ==> B(x)
```

In case of parameter instantiation, a user calls a query expression of the form

```
QC[parameter-subst1,...,parameter-substn]
```

where `QC` is the name of the generic query class. The parameter substitutions are enclosed in square brackets. A *parameter substitution* for instantiation has the form

```
v/p
```

meaning that the query parameter `p` is replaced by the value `v`. In our example, the query call

```
EntityTypeByDomainQ[Integer/whatDomain]
```

returns all instances of `EntityType` which have some feature with domain `Integer`, e.g. `Employee`. A parameter substitution replaces the parameter variable `p` by the constant value `v` supplied in the query call. The substitution is applied to the conclusion predicate of the query predicate and instantiates all matching parameter variables. In the running example, the substitution for parameter variable `w` (`whatDomain`) yields the substituted query rule:

```
forall x (x in EntityType) and (Integer in Domain) and  (x feature
Integer)
==> Q(EntityByDomainQ,x,whatDomain,Integer)
```

A query class may have more than one parameter. A query call may provide parameter substitutions for all or some of the parameters defined for the query. The parameters that are not substituted remain existentially quantified.

The second type of parameter substitution is to *specialize* it. Each parameter has a range associated to it by the query class definition (for example Domain is the range of the parameter whatDomain). When calling the query, the user can provide a stricter range, i.e. a subclass of the original range.

```
NumberDomain in Node isA Domain end
Integer in NumberDomain end
Real in NumberDomain end
```

Here, the NumberDomain is a subclass of Domain and thus a possible stricter range for the parameter whatDomain. The syntax for parameter specialization is

```
p:C
```

where `C` is the name of a subclass of the original range of the parameter `p`. A parameter specialization replaces the range of the parameter's variable in the query rule. For example, the query call

```
EntityTypeByDomainQ[whatDomain:NumberDomain]
```

is evaluated on the substituted query rule

```
forall x,w (x in EntityType) and (w in NumberDomain) and  (x feature
w)
 ==> Q(EntityByDomainQ,x,whatDomain,w)
```

If a generic query class contains more than one parameter then a query call can contain any mixture of parameter instantiations and specializations for all or part of the query's parameters. One parameter may however not be both instantiated and specialized in the same query call. The attribute category `parameter` is defined for generic query classes. It may be combined with the categories `retrieved_attribute` and `computed_attribute`. For example, the query class

```
EntityTypeByDomainQ1 in GenericQueryClass isA EntityType with
     parameter, retrieved_attribute
          feature: Domain
     constraint
          c2: $ (~this feature ~feature) $
end
```

returns as answer attribute the feature(s) of instances of `EntityType`. At the same time, the user can parameterize this attribute. The query call expression

```
     EntityTypeByDomainQ1[Integer/feature]
```

returns all instances of `EntityType` that have at least one feature of domain `Integer` and would return those features in the answer.

In principle, a query call is allowed at all positions in a Telos frame where a Telos class is allowed. ConceptBase does however only allow them inside Telos frames of ordinary classes.

3.12 Views as extended query classes

Views are extending the functionality of generic query classes in two directions. First, they allow complex attributes. Second, they introduce a variant for retrieved attributes that allows answers to have zero fillers or more on specified attributes. The definition if the class `View` introduces the new features as follows.

```
View in Class isA GenericQueryClass with
     attribute
          inherited_attribute : Proposition;
          partof : SubView
end
```

Attributes of the category '`inherited_attribute`' are similar to retrieved attributes of query classes, but they are not necessary for answer objects of the views, i.e. an object must not necessarily instantiate the attribute to be a solution of the view.

The '`partof`' attribute allows the definition of complex nested views, i.e. attribute values are not only simple object names. They can also represent complex objects with some further attributes. The following view retrieves all employees with their departments, and attaches the head attribute to the departments.

```
EmpDept in View isA Employee with
     retrieved_attribute, partof
          dept : Department with
                    retrieved_attribute
                         head : Manager
                end
end
```

The subview at `EmpDept!dept` is an abbreviated view definition where the attribute value (here: `Department`) has the role of the subview's superclass. ConceptBase internally decomposes the nested view definition into several un-nested view definitions:

```
EmpDept in View isA Employee with
  retrieved_attribute, partof
     dept : SV_EmpDept_dept
end

SV_EmpDept_dept in SubView isA Department with
  retrieved_attribute
     head : Manager
end
```

A view definition can have arbitrarily many subviews. Each subview can itself have subviews. The level of nesting in the view definition determines the nesting within the answer objects. In frame syntax, they appear as follows:

```
John in EmpDept with
dept
    JohnsDept : Production with
        head
            ProdHead : Phil
    end
end

Max in EmpDept with
dept
    MaxsDept : Research with
        head
            ResHead : Mary
    end
end
```

If one specifies '`inherited_attribute`' instead of '`retrieved_attribute`', then the corresponding attribute may have zero fillers in the answer. As example consider the following view definition:

```
EmpDept1 in View isA Employee with
     retrieved_attribute, partof
          dept : Department with
```

```
                    inherited_attribute
                         head : Manager
                end
end
```

Then, an employee like `Mary` whose department has no head might also be in the answer set:

```
Mary in EmpDept1 with
     dept
          MaxsDept : Marketing
end
```

To make the definition of views easier, we allow some shortcuts in the view definition for the classes of attributes. For example, if you want all employees who work in the same departments as `John`, you can use the term `John.dept` instead of `Department`. In general, the term `object.attrcat` refers to the set of attribute values of `object` under the attribute category `attrcat`, i.e. all objects `x` such that `(object attrcat x)` holds. This path expression can be extended to any length, e.g. `John.dept.head` refers to all managers of departments in which John is working.

A second shortcut is the explicit enumeration of allowed attribute values. The following view retrieves all employees, who work in the same department as `John`, and earn 10000, 15000 or 20000 EUR.

```
EmpDept2 in View isA Employee with
     retrieved_attribute
     dept : John.dept;
     salary : [10000,15000,20000]
end
```

As mentioned before, subviews use the same syntax as normal view definitions. You can also specify constraints in subviews, which refer to the object of the outer frame(s).

```
EmpDept_likes_head in View isA Employee with
retrieved_attribute,partof
    dept : Department with
        retrieved_attribute, partof
            head : Manager with
                constraint c :
                $ (~this likes ~this::dept::head) $
        end
    end
end
```

The variable 'this' in nested always refers to the object of the main view, in this case an employee. Objects of the nested views can be referred by '`~this::label`' where label is the corresponding attribute name of the subview. In the example, we want to express that the employees must like their bosses. Because the subview for managers is part of the subview for departments, we must use the '::' operator twice: '`~this::dept`' refers to the departments of '`~this`' and '`~this::dept::head`' refers to the heads of the departments of '`~this`'.

3.13 Meta-level formulas

Deductive rules and integrity constraints are formulas over the set of allowed predicates. In our examples, the formulas are attached to classes and make statements about the instances of the classes. For method engineering, one has to design modeling notations where the symbols like nodes and links have a pre-defined semantics: when a model is created using these symbols, then their interpretation has to be consistent with the definition of the modeling notation. In such cases, the formulas range not just over instances of classes but over the instances of instances of classes. As an example, consider the sub class relationship. In Telos, this is encoded by the '`isA`' predicate. Its interpretation is defined by the Telos axioms, namely

```
forall o (o in Proposition) ==> (o isA o)
forall c,d,e (c isA d) and (d isA e) ==> (c isA e)
forall x,o,c,d (x in c) and P(o,c,isa,d) ==> (x in d)
```

The interesting formula is the last one. It quantifies over variables `c`,`d` which themselves are classes that stand in sub class relationship.

Definition: Let f be a Datalog formula. The f is called a *meta-level formula* iff a predicate (x in c) with variable c occurs in f.

As such, meta-level formulas are nothing special. They do however have serious implications on the stratification in a Telos database: about half of all Telos objects are instantiation objects. If we would stratify on the predicate '`in`' for instantiation, then we would not be allowed to use a negative literal '`not (x in c)`' in any rule which directly or indirectly derives the predicate '`(x in c)`'. Because the '`in`' predicate is so frequently used in Telos, this would basically eradicate the use of negation. To overcome this problem, ConceptBase applies stratification to the predicate symbol '`In.c`', i.e. the combination of the predicate name plus the second argument. This method however doesn't work directly for meta-level formulas since they feature '`in`' predicates with a variable as second argument.

The problem can be overcome by *partial evaluation*. If the meta level formula f contains a positive literal `Q(…,c,…)`, which binds the variable c, then we compute the extension of `Q` and rewrite the meta-level formula with all facts from the extension of `Q`. In the case of the '`isA`' rule, we can take the predicate `P(o,c,isa,d)` in the role of the Q-predicate. Assume that we have two facts `P(o1,Emp,isa,Person)` and `P(o2,Manager,isa,Emp)` in the extension of the P-predicate. Then the meta-level formula is replaced by two partially evaluated formulas:

```
forall x (x in Emp) ==> (x in Person)
forall x (x in Manager) ==> (x in Emp)
```

The partially evaluated formulas have now constants as second arguments of the '`in`' predicates yielding better stratifiability opportunities. The partial evaluation algorithm for meta level formulas is more complex than the above example suggests. In general, there can be more than one Q-predicate to be used for partial evaluation. The system has then to select one, which does not have too many facts in its extension. A huge extension would result in a huge number of partially evaluated

formulas. In some cases, the partial evaluation must be applied more than once in order to eliminate all free variables of '`in`' predicates.

Example 1. As an example for meta-level formulas consider cardinality of attributes. Let's assume that we want to specify that certain attributes require a filler ('`necessary`') or have at most one filler ('`single`').

```
Manager in Class with
     attribute
          dept: Department;
          leads: Project
     constraint
          onDept: $ forall m/Manager
                    d1,d2/Department
                     (m dept d1) and (m dept d2)
                     ==> (d1 == d2) $;
          onLead: $ forall m/Manager
                     exists p/Project (m leads p) $
end
```

Obviously, the '`dept`' attribute is single-valued and the '`lead`' attribute is necessary. However, the constraints are formulated at the model level (here with `Manager`). We require a generic formulation of single and necessary attributes that we can re-use for any class. To do so, we extend the class `Proposition` by two more attributes and define two meta-level formulas.

```
Proposition in Class with
  attribute
      single: Proposition;
      necessary: Proposition
  constraint
singleIC: $ forall p/Proposition!single c,d/Class x,m/VAR
            P(p,c,m,d) and (x in c)  ==>
                  (forall y1,y2/VAR
                   (y1 in d) and (y2 in d) and (x m y1)
                   and (x m y2) ==> (y1 == y2)) $;
necessaryIC: $ forall p/Proposition!necessary
                 c,d/Class x,m/VAR
           P(p,c,m,d) and (x in c) ==>
           (exists y/VAR (y in d) and (x m y)) $
end
```

The pseudo class range `VAR` can be used when the corresponding variable gets a range by the partial evaluation or is replaced by a constant. For example, the variables `x` gets a range by the partial evaluation of predicate `P(p,c,m,d)`. The same predicate replaces the variable `m` by constants. The meta formulas for single and necessary allow the re-use of their semantics just by using the appropriate attribute categories:

```
Manager in Class with
     single
          dept: Department
     necessary
          lead: Project
end
```

The partial evaluation for the single attribute works as follows: The database implies the facts `(Manager!dept in Proposition!single)` and `P(Manager!dept,Manager,dept,Department)`. This matches the conjunction '`(p in Proposition!single) and P(p,c,m,d)`'[8] in `singleIC` leading to a substitution

```
     [Manager!dept/p, Manager/c, dept/c, Department/d]
```

Applying this substitution to the `singleIC` formula yields the partially evaluated formula

```
forall x/VAR (x in Manager) ==>
       (forall y1,y2/VAR
       (y1 in Department) and (y2 in Department)
      and (x dept y1)  and (x dept y2)
      ==> (y1 == y2))
```

which can be rewritten to

```
forall x/Manager y1,y2/Department
(x dept y1)  and (x dept y2) ==> (y1 == y2))
```

Figure 3.4 illustrates that meta-level-formulas are defined at the level of meta classes. They define properties of the instances of classes, which are themselves instances of the meta classes. Hence, if we use the meta class level for defining modeling notations, the meta-level formulas can be employed to define part of the semantics of the notational symbols, in our example the semantics of the '`single`' and '`necessary`' symbols. Note that the class `Proposition` serves as meta class. Since any object is instance of `Proposition`, this is a consistent use of classification. In the example, the token object `Mary` is a proper instance of `Manager` because neither the `singleIC` nor the `necessaryIC` is violated. The object `Bill` violates both constraints.

Example 2. The second example is about defining the meaning of *transitivity*, *symmetry* and similar relation properties. These properties are so frequent that they are part of any textbook in algebra. Hence, it would be useful to have pre-defined attribute categories for them rather than formulating transitivity over and over again for specific relations like the `subordinate` attribute of section 3.9. The subsequent definitions solve the problem at the most generic level, the level of `Proposition`.

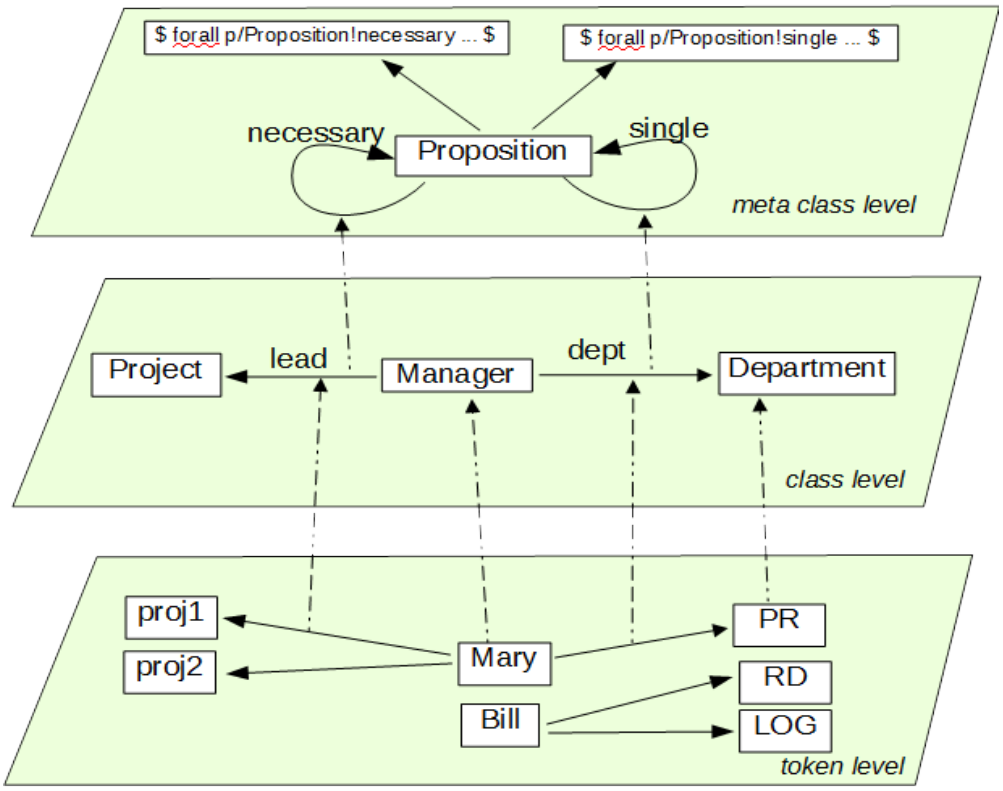

Figure 3.4: Metalevel formulas as properties of metaclasses

```
Proposition with
  attribute
    transitive: Proposition;
    antisymmetric: Proposition
end
```

The two meta-level formulas for transitivity and anti-symmetry are then defined by:

```
RelationSemantics in Class with
  rule
   trans_R: $ forall x,z/VAR
                    (exists
                        AC/Proposition!transitive
                        C/Proposition y/VAR M/VAR
                     P(AC,C,M,C) and (x in C) and
                    (y in C) and (z in C) and
                (x M y) and (y M z)) ==> (x M z) $
  constraint
   antis_IC: $ forall AC/Proposition!antisymmetric
                     C/Proposition x,y/VAR M/VAR
```

```
                    P(AC,C,M,C) and (x in C) and
                    (y in C) and
                    (x M y)  ==> not (y M x) $
end
```

[source:LinkSemantics.sml]

The reader will recognize the pattern in the last lines of the two formulas. The clauses above are for defining the scope of the variables. By defining transitivity and ant-symmetry at the most generic object, `Proposition`, these attribute categories are applicable to any attribute definition by just attaching them as attribute categories. Consider a class `Person` that has an ancestor attribute which should be anti-symmetric and transitive. Given the above-mentioned formulas, the attribute can be fully specified by:

```
Person with
  attribute,antisymmetric,transitive
    hasAncestor: Person
end
```

The properties symmetry and reflexivity can be defined by similar expressions. The full details are in the example models provided on the CD-ROM.

ConceptBase supports meta-level formulas for rules and constraints of proper classes but not for query classes. While meta-level formulas do not extend the formal expressiveness of Telos, they save a lot of coding effort and they seamlessly integrate into the IRDS-based strategy of defining modeling notations. The more of such formulas are expressed at the meta-class level, the deeper is the explicit knowledge about the modeling notations, in particular the meaning of the symbols in the modeling notations. Section 3.16.5 comes back to this issue to define the meaning of cardinality expressions in data modeling notations. Chapter 7 elaborates on an even more complex application of meta-level formulas.

3.14 Active rules

Rules, constraints and queries are the principal means to express computation in the ConceptBase meta database system. Since they are all mapped to Datalog with negation, one can guarantee termination of any computation within this logical framework. Any computation beyond the Datalog scope is supposed to take place in application programs that interact with the meta database system.

In order to extend the computational scope, so-called active rules have been introduced in ConceptBase. An active rule has three parts. The *event section* specifies on which external event the active rule is activated. With ConceptBase, an external event can be an update to the database or a call of a query[9]. The event section binds variables to values. The second part is *the condition section*. It is a logical formula over a superset of the variables bound by the event section. The formula is evaluated over the database state. The last part is the *action section*. It consists of a sequence of procedural calls that either update the database (insert, delete), or invoke further query calls including built-in queries. Built-in queries are queries without a logical constraint. Instead they execute a piece of program code. The class definition of active rules is as follows:

```
ECArule in Class with
  attribute
     ecarule : ECAassertion;
     priority_after : ECArule;
     priority_before : ECArule;
     mode : ECAmode;
     active : Boolean;
     depth : Integer
end
```

The attribute `ecarule` holds the specifications for the event, condition and action parts. The three parts are sometimes referred to by the acronym ECA. An ECA rule is represented as a string that starts with variable *declarations:*

```
$ v1/c1 v2/c2 ...
ON event
IF condition
DO actions-1
ELSE actions-2 $
```

The first line contains the declaration of all variables used in the `ECAassertion`. The specified classes of the variables are only used for compilation of the rule, during the evaluation of the rule it is *not* tested if the variables are instances of the specified classes. The variables are bound to objects by event, condition or action statements.

Possible events are the insertion (Tell) or deletion (Untell) of attributes (predicate `A`), instantiation links (predicate `In`), or specialization links (predicate `Isa`). For example, if the rule should be executed if an object is inserted as instance of `Manager`, then the event statement is:

```
Tell(In(x,Manager))
```

Furthermore, an event may be a query, e.g. if you specify the event

```
Ask(find_instances[Employee/class])
```

the ECA rule is executed before the evaluation of the query `find_instances` with the parameter Employee. It is possible to use a variable as a placeholder for a parameter.

The event detection algorithm takes only extensional events into account. Events that can be deduced by a rule or a query are not detected. However, the algorithm cares about the predefined Telos axioms, e.g. if an object is declared as an instance of a class, the object is also an instance of the super classes of this class. The condition is a predicate evaluated on the database. It can be either a normal literal (`A`, `In` or `Isa`) or a query like `In(x,EmpDept1)`. If the condition contains a free variable the actions of the '`DO`' block are executed for each result for this variable. If the condition contains only constants or bound variables and can be evaluated to TRUE, the '`DO`' action block is executed once. Otherwise the '`ELSE`' block is executed. By default, queries are evaluated on the old state of the object base before the transaction started. If one wants to take into account new information one has to use the `new` operator. For example, if one wants to check the instantiation of an object on the new database state, one has to specify

```
new(In(x,Person)
```

in the condition.

Actions are specified in a comma-separated list. The syntax is similar to that one of events, except that you can also ask queries (`Ask`). All variables in `Tell` and `Untell` actions must be bound. The Tell of an attribute `A(x,ml,y)` is only done, if there is no attribute of category `ml` with value `y` for object `x`. Then, a new attribute with a system-generated label is created. If an attribute `A(x,ml,y)` should be deleted, then all attributes of category `ml` with value `y` for object `x` are deleted. In `Ask` actions, only one free variable is allowed. In that case, the rest of the action block is executed for each result of this variable.

Active rules are linked to transactions on the database. A transaction is a sequence of updates (insertions or deletions) and queries. All events are derived from the entries in the transaction. The attribute `mode` controls when an active rule is evaluated. There are three possible modes:

Immediate: The condition is evaluated immediately after the event has been detected. If it evaluates to TRUE, the action is executed immediately, too.
ImmediateDeferred: The condition is evaluated immediately after the event has been detected. If it evaluates to true, the action is executed at the end of the current transaction.
Deferred: The condition is evaluated at the end of the current transaction. If it evaluates to true, the action is executed immediately after the evaluation of the condition.

The default mode is `Immediate`. The attributes `priority_after` and `priority_before` establish a partial order on the set of active rules. In case that more than one active rule is matching an event, the partial order determines the sequence in which the active rules are evaluated. The attribute `active` has possible values TRUE and FALSE. It allows to deactivate/reactivate an active rule without having to delete or re-insert it to the database.

Finally, the attribute `depth` controls the depth of the call tree of active rules: the action part of an active rule can produce a new event, which triggers the call of another active rule which itself can trigger further active rules. When the specified nesting depth is reached, the execution of nested active rules is aborted. This feature prevents infinite loops. The default value for the depth is 0, i.e. no nested calls are permitted.

Example:
We model a situation where employee candidates are entered into the database. Those objects fulfilling a certain exception condition are put on a ‘watch list’ (first ECA rule). Otherwise they are entered as ordinary employees into the system. As soon as we acknowledge that we have looked at such a candidate, she is removed from the watch list and entered as normal employee. First, the necessary classes have to be defined. The query class `UnderPayed` is defining what is an exception employee candidate.

```
EmployeeCandidate in Class with
  attribute
    salary: Integer
end

Employee isA Employee end
ToBeWatched end
AlreadyWatched end

UnderPayed in QueryClass isA EmployeeCandidate with
  constraint
    c1: $ exists s/Integer (~this salary s) and
             (s < 1000) $
end
```

The first ECA rule is triggered whenever an instance of `EmployeeCandidate` is entered into the database. If so, the exception condition `UnderPayed` is checked for that candidate on the new database state. If the condition is true, the candidate is set on the watch list; otherwise she is entered as normal employee.

```
WatchForUnderpayed in ECArule with
     mode m: Deferred
     ecarule
        er : $ e/Employee
             ON Tell(In(e,EmployeeCandidate))
             IF new(In(e,UnderPayed))
             DO
                Tell(In(e,ToBeWatched))
             ELSE
                Tell(In(e,Employee))
             $
end
```

The second ECA rule is for removing a candidate from the watch list. It is triggered by an instantiation of the class `AlreadyWatched`.

```
Okayed in ECArule with
     mode m: Deferred
     ecarule
        er : $ e/EmployeeCandidate
             ON Tell(In(e,AlreadyWatched))
             IF TRUE
             DO
                Untell(In(e,ToBeWatched)),
                Tell(In(e,Employee))
             $
end
```

The second ECA rule has no ELSE part, i.e. when the condition would be false no action would be executed. Note that the action part contains two actions. The example can be executed by inserting the following frames.

```
mary in EmployeeCandidate with
  salary s: 500
end

mary AlreadyWatched end
```

[source: EcaExample.sml]

3.15 Engineering the Yourdan method

The Modern Structured Analysis (Yourdan, 1989) is an example of a collection of modeling languages that is comprehensive while still rather simple. It has been proposed before the shift to object-orientation. The purpose of the structured analysis method is to specify a system in a more formal way by graphical notations. A *system* here is an object that receives information from the environment and reacts to it by generating output or changing its internal state. The Yourdan method serves here as a test case to demonstrate the meta-modeling approach with Telos.

Developers of the system modeling methods observed that a system could be viewed from multiple perspectives. First, one represents functions (or processes) of the system with their inputs and outputs. This is the *functional perspective*. Second, the *data perspective* specifies about which entities of the external world the system shall make records and what is the structure of the data elements processed by the system. A *control perspective* augments the other two perspectives defining how the system reacts to control event triggered by some object (or person) from the environment.

This section discusses the engineering of the Yourdan method in order to highlight the various aspects of method engineering and meta-modeling. The presentation starts with the data perspective because of its relative simplicity. A very simple notation definition level is assumed which is then applied to the functional perspective. We then discuss the development of inter-notational rules and constraints, which relate models developed in different perspectives. Finally, process models are presented which encode the steps of a modeling method (rather than the results of the steps which are encoded on the modeling notations). The Telos formalizations of the Yourdan method are contained in the companion CD-ROM. They can directly be tested with the ConceptBase system. The detailed structure of the Yourdan method case study is as follows:

- Modeling the ERD-simple notation: This is a basic version of the entity-relationship diagramming notation. It features entity types, relationship types, and attributes.
- Modeling the ERD-advanced notation: This notation extends the previous one by ISA relations and cardinalities.
- Modeling the DFD notation: This notation encodes data flow diagrams
- Modeling inter-notational constraints: This section shows how constraints and queries can be used to manage the connections between models for different perspectives of the same artifact. The generation of theses constraints can be managed via a richer notation definition level.
- Modeling process model notations: Here, the individual steps of a method are themselves encoded in a notation

[source:Yourdan-Complete]

3.15.1 Modeling the ERD-simple notation

Figure 3.2 introduced abstraction levels that can be used to define modeling notations as well as models, and their incarnations. Entity-Relationship diagrams (ERD) are models to specify data sets. So-called *entity types* are the denotations of sets of identifiable objects (entities) that can have descriptive properties (attributes). An attribute has a label. Its values are elements from a domain, for example integer or string. A *relationship type* is denoting a set of relationships. A relationship links several participating entities. The relationship type defines so-called roles under which entities can

participate. For example, a product entity can be linked to a customer entity by a relationship of type '`orders`'.

Relationship types can restrict the cardinalities of the entities that participate in a relationship. For example, in an '`orders`' relationship there must be exactly one customer and at least one product. The Telos model of the ERD notation is first defining the base concepts mentioned above:

Notation definition level

```
Node with
  attribute
    connectedTo: Node
end
```

This simple notation definition level just provides the concepts `Node` and `Node!connectedTo`, i.e. the ability to represent graphs. It is sufficient for the moment. We will later extend it to represent software development steps.

Notation level for a simple ERD notation

```
ObjectType in Node end   {* used later *}

EntityType in Node isA ObjectType with
  connectedTo
    ent_attr: Domain
end

RelationshipType in Node isA ObjectType with
  connectedTo
     role: EntityType
end

Domain in Node end
```

Model level for an example ERD

```
Integer in Domain end
Domain String end
Date in Domain end
String isA Date end

Employee in EntityType with
ent_attr
    e_name: String;
    earns: Integer
end

Project in EntityType with
  ent_attr
```

```
    budget: Integer
end


worksFor in RelationshipType with
  role
   toEmp: Employee;
   toProj: Project
end
```

[source: ERD-Simple]

The first 3 objects define the three domains `Integer`, `String`, and `Date`. The classes `Integer` and `String` are pre-defined in ConceptBase. The class `Date` is defined here to subsume `String`. In other words, any instance of `String` is allowed as an instance for `Date`. This is a technical trick when the details of the `Date` domain are not of further interest in the modeling. The domains are part of the model level because their instances (the values) are at the data level.

The remainder of the model defines two entity types and one relationship type. Note that the relationship type uses the role names '`toEmp`' and '`toProj`'. Since they point to different entity types, these role names are not displayed in an ERD. However, the Telos representation has to assign such names because the role links have to be identifiable. An equivalent definition would be:

```
worksFor in RelationshipType with
  role
   role1: Employee;
   role2: Project
end
```

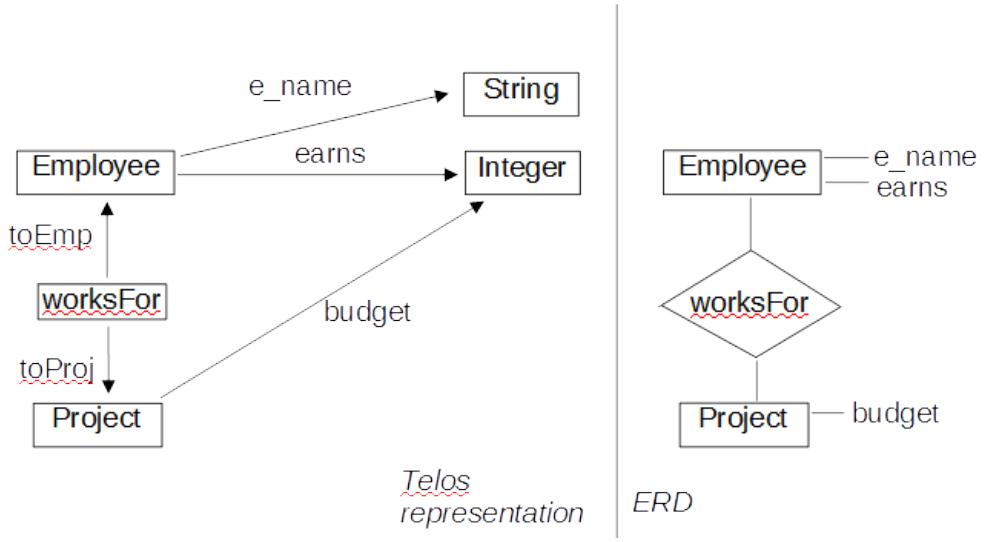

Figure 3.5: ERD versus its Telis representation

Figure 3.5 displays the model in a conventional ER diagram and contrasts it to the Telos representation. The difference is that the Telos representation uniformly assigns role names and attaches domains to attributes. The ER diagram has different graphical symbol for entity types and relationship type. The Telos model contains this information via the classification links `(Employee in EntityType)`, `(Project in EntityType)`, and `(worksFor in RelationshipType)`. The role links in the ER diagram appear undirected but they always link a relationship type with an entity type. Therefore, the role links are implicitly directed. It follows that the ER diagram can always be reconstructed from the Telos representation.

The ERD-simple notation has no user-defined constraints (called balancing rules in the Yourdan textbook) and lacks symbols for ISA relationship types and complex object types (called associative object indicators by Yourdan). The subsequent section introduces them and gives extensive examples how to model constraints on ERD models and their semantics.

3.15.2 Modeling the ERD-advanced notation

Besides the properties introduced in ERD-simple, we might want the following new features:

1. There shall be a special relationship type ISA that has one entity type as 'super type' and several entity types as 'sub types'. For a given entity type, there can be several ISA relationship types which have this entity type as super type. Similarly, an entity type may occur as sub type in multiple ISA relationship types. The intended semantics is that each instance of a sub type is also an instance of the super type.

2. A new object type shall be introduced which is both an entity type and a relationship type, sometimes called a complex object type. Classical example is an order that has role links to several entity types but also has some attributes and may be referred to by other relationship types.

3. Cardinalities for role links have to be introduced. We limit consideration to the cardinalities 1, 2 and many. Cardinalities have to be enforced at the data (base) level. Cardinalities have to be used in a consistent manner. For example, one may not specify that a role link is both 'minimum 2 fillers' and 'maximum 1 filler'.

4. Key attributes identify entities, i.e. there may not be two different entities with the same value for the key attribute(s).

The models for the ERD-advanced notation are on the companion CD-ROM.
[source: ERD-Advanced]

3.15.3 Modeling the DFD and Event list notations

Data flow diagrams consist of so-called processes, data stores, data flows, terminators, control processes, and control flows. The DFD notation has as new element the so-called leveling. Leveling is

a way to decompose a process into a DFD diagram consisting of DFD elements (processes, stores, etc.). The processes in the leveled DFD diagram can themselves be leveled or they are specified by a so-called process specification (pseudo code).

Since we now employ a second notation, it makes sense to introduce a slightly extended notation definition level that allows to keep models apart from each other and to express that certain models belong to the same modeling project.

Notation definition level with Model and Project

```
Node in Individual with
  attribute
    connectedTo: Node
end

Project in Individual with
  attribute
    produces: Model
end

Model in Individual with
  attribute
   contains: Node
end
```

The new meta concept `Model` aggregates some content (here: nodes) and can be compared with a file containing some records. The meta concept `Project` aggregates several models that belong to the same modeling project. Among others, the two new concepts at notation definition level allow to express constraints on models that depend on whether they are belonging to the same modeling project.

Notation level for the DFD notation

```
Node DFD_Node in Node with
  connectedTo
    dataflow: DFD_Node
end

DFD_Node!dataflow with
  attribute
    withType: DataType
end

Process in Node isA DFD_Node with
    connectedTo
```

```
      leveledTo: DFD_Figure
end

Terminator in Node isA DFD_Node with
End



Store in Node isA DFD_Node with
  connectedTo
    withType: ObjectType
end

DataType in Node with
end

DFD_Model in Model with
  connectedTo
     contains: DFD_Node
end

DFD_Figure in Node,Model isA DFD_Model end

nowhere in DFD_Node end


PreliminaryBehavioralModel isA DFD_Model end

DataType in Node with
end {* reference to data dict. *}

ObjectType in Node isA DataType with
end {* reference to ERD notation *}
```

[source: DFD]

The notation level for DFD makes use of the fact that some model components (`DFD_Node`) can be aggregated to model (`DFD_Model`). The leveling related a process to a `DFD_Figure` (a special DFD model). The leveled `DFD_Figure` can contain data flow links whose source or destination is not part of the `DFD_Figure`. Such data flows are called dangling links in the Yourdan method. The object '`nowhere`' serves as artificial source/destination for those data flows. The concept 'preliminary behavioral model' refers to a DFD model that is created from the so-called event list (see below).

The DFD notation also contains the notion of a so-called control process. A control process has no dataflows but so-called control flows. Incoming control flows are labeled by control events denoting that some condition in the environment or in the system has become true. Outgoing control

flows have no label and always connect a control process with an ordinary process: the control process can activate the process under certain conditions. The additional definitions for control processes are:

```
DFD_Node in Node with
  connectedTo
    incomingCF: ControlProcess
end


DFD_Node!incomingCF with
  attribute
     withEvent: EventType
end

ControlProcess in Node isA DFD_Node with
   connectedTo
     outgoingCF: Process
end
```

So-called event types decorate the incoming control flows. Some of these event types are originating from the environment. In such cases they are grouped in the so-called event list. External event types can be associated to a terminator who originates the event. We distinguish three sub-classes of external event types: flow event types are defined as data inputs from a terminator to the system, control event types are stating that a certain condition has become true in the environment, and temporal event are true when a certain point of time has been reached. The precise definition can be looked up in the textbooks about the Yourdan method. It is worth noting here that models group some details together, here: an event list is a just a list of external event types.

[source: ERD-Simple]

Notation level for the event list notation

```
EventType in Node with
   attribute
     eventtext: String
end

ExternalEventType in Node isA EventType with
   connectedTo
     agent: Terminator;
     answeredBy: Process  {* links events to processes *}
end

EventList in Model with
  contains
    containsEvent: ExternalEventType
end
```

```
FlowEventType in Node isA ExternalEventType end
ControlEventType in Node isA ExternalEventType end
TemporalEventType in Node isA ExternalEventType end
```

Figure 3.6 visualizes the DFD notation and its relation to other Yourdan notations. The *cross-notational* links called '`withType`' and '`withEvent`' happen to start from concepts of the DFD notation. It's also possible to have them start from the concepts of the other notations and let them end in the DFD concepts (as is the case for the link '`respondedBy`' of `ExternalEventType`). The method engineer has to decide to which notation the cross-notational links belong. Here, we assume that they belong to the DFD notation. The reader may have noticed that a cross-notational link is not distinguished from other notational links, e.g. the '`dataflow`' link of the DFD notation. The decision to put certain node and link types into a certain notation is made to manage the complexity of a modeling notation within a method. One could also define a single 'Yourdan' notation with all node and link types, i.e. the union of DFD, ERD, and all other Yourdan notations. Such a union notation is however too complex for human modelers to handle. Experience has shown that modeling is more successful when a modeler concentrates on specific aspects of the modeling task.

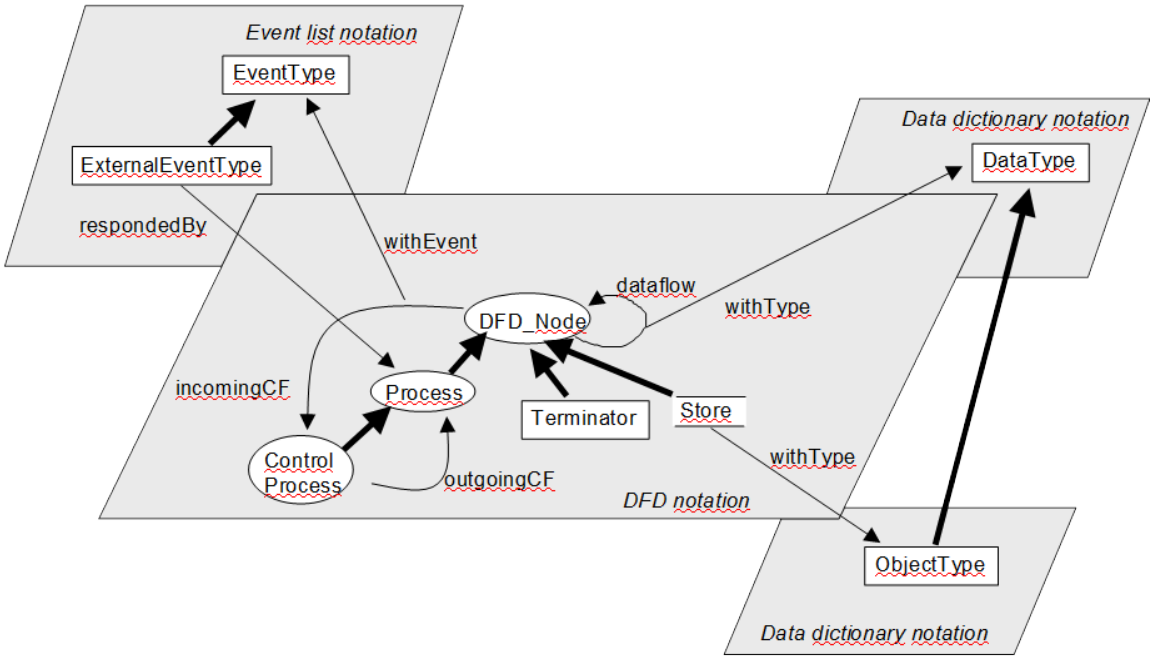

Figure 3.6: The DFD notation cross-related to other Yourdan notations

We observe that the DFD notation is inter-related to other notations in various ways. First, data stores have so-called object types as data structure. The class `ObjectType` is a reference to the ERD notation where it serves as the common superclass of `EntityType` and `RelationshipType`. The data types attached to data flows are can also be regarded as cross-notational links: data types are defined in so-called data dictionaries[10]. Second, the event types attached

to incoming control flows of DFD control processes can be external event types enumerated in the event list notation.

Model level for an example DFD and event list

```
MyDFD in PreliminaryBehavioralModel with
  containsDFDNode
    n1: CUSTOMER;
    n2: CREDITCARDCOMPANY;
    n3: UpdateAccounts;
    n4: Accounts
end

CUSTOMER in Terminator with
   dataflow
      d1: UpdateAccounts
end

CREDITCARDCOMPANY in Terminator with
  dataflow
    d2: UpdateAccounts
end

UpdateAccounts in Process with
  dataflow
    d3: Accounts
end

Accounts in Store end

CUSTOMER!d1 with
  withType
    datatype: payment
end

CREDITCARDCOMPANY!d2 with
  withType
    datatype: maximumcredit
end

UpdateAccounts!d3 with
  withType
    datatype: verifiedpayment
end

payment in DataType end
maximumcredit in DataType end
```

```
verifiedpayment in DataType end

MyEventList in EventList with
   containsEvent
     ev1: E1;
     ev2: E2
end

E1 in FlowEventType with
  agent a1: CUSTOMER
  eventtext t: "A customer makes a payment"
end

E2 in TemporalEventType with
  eventtext t: "At 18:00 the invoices of current orders are sent out"
end
```

3.15.4 Modeling intra-notational constraints

An *intra-notational constraint* is a constraint ranging over concepts of a single notation, e.g., the ERD notation. In this section, we only consider constraints, which restrict the set of allowed diagrams[11]. Some of these constraints are already given by the use of Telos. For example, the ERD notation defines

```
RelationshipType in Node isA ObjectType with
  connectedTo
     role: EntityType
end
```

Then, axiom A14 demands that any instance of relationship type that uses the '`role`' link must end in an instance of entity type. Furthermore, any instance of relationship type is also an instance of object type by means of axiom A13. Beyond such pre-defined rules of well-formedness, a notation definition should also include internal constraints.

With ConceptBase, there are two principal strategies to encode notational constraints. First, they can be represented as ordinary class constraints. The second option is to encode them as query classes, which return the 'violators' of the constraint. To show the principle, let us consider the following constraint for the ERD notation:

C1: Any entity type must have at least one describing attribute.

If we want to realize the condition as a class constraint, we have to instantiate the concept `EntityType` to the built-in object `Class`, which defines the attribute category `constraint`.

```
EntityType in Node,Class isA ObjectType with
  connectedTo
```

```
    ent_attr: Domain
  constraint
    c1: $ forall e/EntityType
           exists a/EntityType!ent_attr From(a,e) $
end
```

The above class constraint is a realization of our example but it has one important drawback: no database violating the constraint shall be accepted by the ConceptBase system. In practical modeling applications, we do however want that violations are tolerated *during* the modeling process, i.e. when we constructing the diagrams. During a modeling process, the models are typically incomplete. Only at certain milestones, the incompleteness is expected to be resolved and violations are regarded as not acceptable. For the above constraint, the strict enforcement would prevent us to define any entity type without providing at least one entity attribute. One solution to the problem is to tell the constraint to the system at the milestone. If the constraint is fulfilled, it is accepted by the system. Otherwise, an error message is generated.

There is a second alternative, which is in most cases preferable: represent the constraint as a query class, which returns the violators. This method works for all constraints that have the format the format

`forall X A(X) ==> B(X)`

The violators of the constraints are all `X` that fulfill the condition

`A(X) and not B(X)`

In the above example, we can apply the pattern as follows for `X=e`:

`A(e) = In(e,EntityType)` where the `In` predicate is implicit by the typed quantification `e/EntityType`

`B(e) = exists a/EntityType!ent_attr From(a,e)`

Given that transformation, we can mechanically create the query class, which computes the violators of the constraint:

```
EntityTypeWithoutAttribute in QueryClass
                           isA EntityType with
  constraint
    c1: $ not exists a/EntityType!ent_attr
              From(a,~this) $
end
```

The answer variable '`~this`' assumes the role of the variable '`e`' in the first solution, as elaborated in the section on query classes. The query class representation doesn't have the problem of the first solution. At any point of time, a modeler can ask for the violators of the original constraint. It is a matter of the process model of the method to define when the query class has to return an empty answer.

Being more complex than the ERD notation in terms of the number of concepts, the DFD notation also features more intra-notational constraints. Let us consider the following excerpt:

C2: Any process must have at least one ingoing and one outgoing dataflow.
C3: Dataflows between terminators are forbidden
C4: Dataflows must have data types attached to it except for dataflows starting from or ending in data stores.

The following query classes encode these constraints. Constraint C2 is split in two query classes for the sake of readability and usability.

```
ProcessWithoutInput in QueryClass isA Process with
  constraint
    c21: $ not exists d/DFD_Node!dataflow
               To(d,~this) $
end
```

```
ProcessWithoutOutput in QueryClass isA Process with
  constraint
    c22: $ not exists d/DFD_Node!dataflow
            From(d,~this) $
end
TerminatorWithforbiddenCommunication in QueryClass isA Terminator
with
  constraint
    c3: $ exists t/Terminator d/DFD_Node!dataflow
              From(d,~this) and To(d,t$
end

DataflowWitoutLabel in QueryClass isA
 DFD_Node!dataflow with
  constraint
     c4: $ not (exists s/Store From(~this,s) or
              To(~this,s)) or
         not (exists dt/DataType (d withType dt)) $
end
```

The complexity of a notation can be measured in the number and size of intra-notational constraints. The more such constraints are defined, the more is known about syntactically correct models in that notation. As a consequence, there are also more opportunities to violate the constraints, which makes the notation more difficult to handle. The method engineer should take such considerations into account when defining a notation.

3.15.5 Modeling inter-notational constraints

Inter-notational constraints are constraints that are referring to concepts from more than one notation. Since Telos provides a uniform framework for all notations (as well all models, data, and process executions), such constraints are looking like ordinary intra-notational constraints. In fact, the decision to assign concepts to more than one notation is arbitrary and for the sake of readability of the models and understandability of the notations themselves. A simple example for an inter-notational constraint is the following:

C5: Each data store must have an object type associated to it and vice versa.

Two query classes realize this constraint:

```
ObjectTypeWithoutStore in QueryClass isA ObjectType with
  constraint
    c21: $ not exists s/Store (s withType ~this) $
end

StoreWithoutType in QueryClass isA Store with
  constraint
    c21: $ not exists o/ObjectType
              (~this withType o) $
end
```

The constraint uses the cross-notational link ‘`withType`’ that has been defined as part of the

DFD notation. The concept `ObjectType` is part of the ERD notation whereas `Store` is a concept of the DFD notation. Besides that, there is nothing special with the intra-notational constraint.

We observe that the concepts `ObjectType` and `Store` are closely related. They have to occur in pairs. So apparently, they are two aspects of the same abstract thing. This observation leads indeed to a principle of method engineering that can be represented in the notation definition level and then exploited for the detection of inter-notational constraints. In the example, both `ObjectType` and `Store` are incarnations of an abstract Data concept. Object types are referring to the interrelationships between data concepts (as handled by the ERD notation). Stores are about the location of data in the network of communication processes (as handled by the DFD notation). The common abstract object is the manifestation of the artifact focus mentioned earlier: the models and notations are interrelated because they make statements about the same artifact, e.g., an information system.

Hence, it is logical to enrich the notation definition level by such abstract concepts and then investigate the links between their incarnations in different notations. The following richer notation definition level may be suitable for system analysis methods like Yourdan.

Notation definition level with abstract concepts

```
Node in Individual with
  attribute
    connectedTo: Node
end

Project in Individual with
  attribute
    produces: Model
end

Model in Individual with
  attribute
   contains: Node
end

DataConcept in Individual isA Node end
ActivityConcept in Individual isA Node end
ControlConcept in Individual isA Nod end
```

The three abstract concepts here are data concept, activity concepts, and control concept. Whenever two different notations instantiate the same abstract concept, there is an obligation to check the necessity of a cross-notational link and inter-notational constraints. In the case of object type and stores, we have the following scenario displayed in Figure 3.7.

If we would deal with the data concept in a single notation, there would be no need to distinguish the `Store` aspect of data from the `ObjectType` aspect. By separating them, we need to manage the fact that their definitions are synchronized. In Telos, the synchronization is realized by the cross-notational link plus the inter-notational constraints as shown before. The Yourdan method features more examples of the phenomenon. Two further examples are

- Process and process specification are both incarnations of the activity concept.
- Control process and state transition diagrams are incarnations of the control concept.

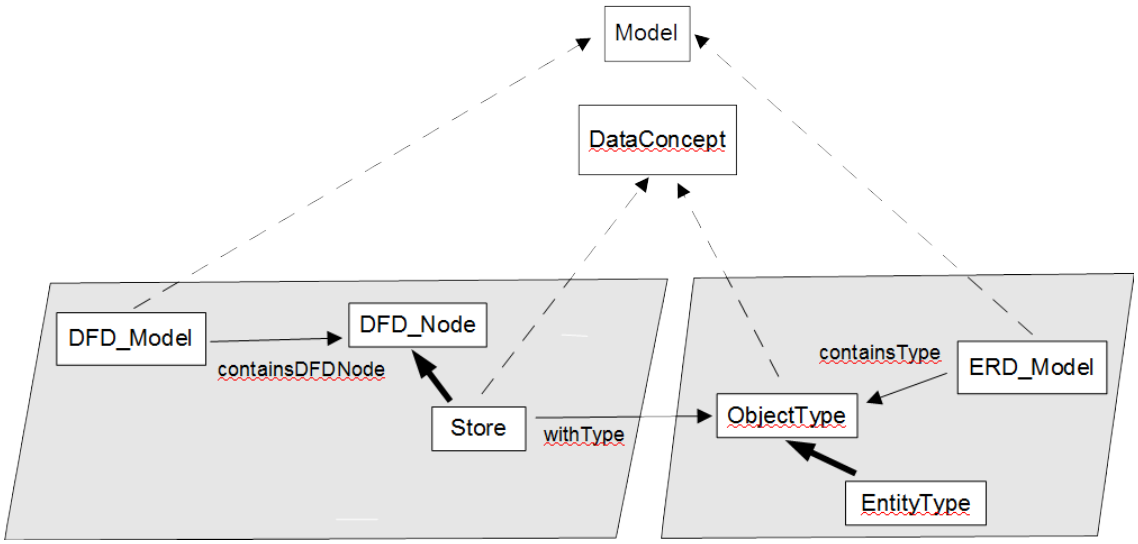

Figure 3.7: Incarnation of `DataConcept` in ERD and DFD

The artifact focus in method engineering has another consequence: only when two models are about the same artifact like an information system, they need to be synchronized. Assume that we would maintain models of different modeling projects in the same repository. How can we keep them separate from each other? The answer lies in the objects `Project` and `Model` of the notation definition level. If the repository stores details of several independent projects, then the notational constraints need to be made 'project-aware'. For example, if a modeling project contains a DFD model which itself contains a store, then there must be an object type linked to this store that is defined in an ERD model of the same modeling project:

```
YourdanProject in Project end

StoreWithoutTypeV2 in QueryClass isA Store with
  constraint
    c21: $ exists p/YourdanProject dfd/DFD_Model
    (p produces dfd) and(dfd containsDFDNode ~this)
     and not (exists erd/ERD_Model o/ObjectType
                (p produces erd) and
                (erd containsType o) and
                 (~this withType o) $
end
```

The object `YourdanProject` is at the notation level. It subsumes all modeling projects using Yourdan notations. Such project-aware intra-notational constraints can be written in various combinations and versions. For example, assume that the method engineer wants to check whether a

given Yourdan project violates constraint C2. In this case, a generic query class is a facility for formalization of the constraint:

```
DFDwithUntypedStore in GenericQueryClass isA DFD_Model with
  parameter,computed_attribute
     project: YourdanProject
  computed_attribute
     store: Store
  constraint
    c21: $ (~project produces ~this) and
    (~this containsDFDNode ~store)
    and not (exists erd/ERD_Model o/ObjectType
    (~project produces erd) and
    (erd containsType o) and (~store withType o) $
end
```

The query will return all DFD model together with their project and the violating store. The parameter '`project`' can be left free. Then, the query will scan all known Yourdan projects in the repository. The attribute category '`computed_attribute`' makes sure that the involved project and data store are returned together with the name of the violating DFD model.

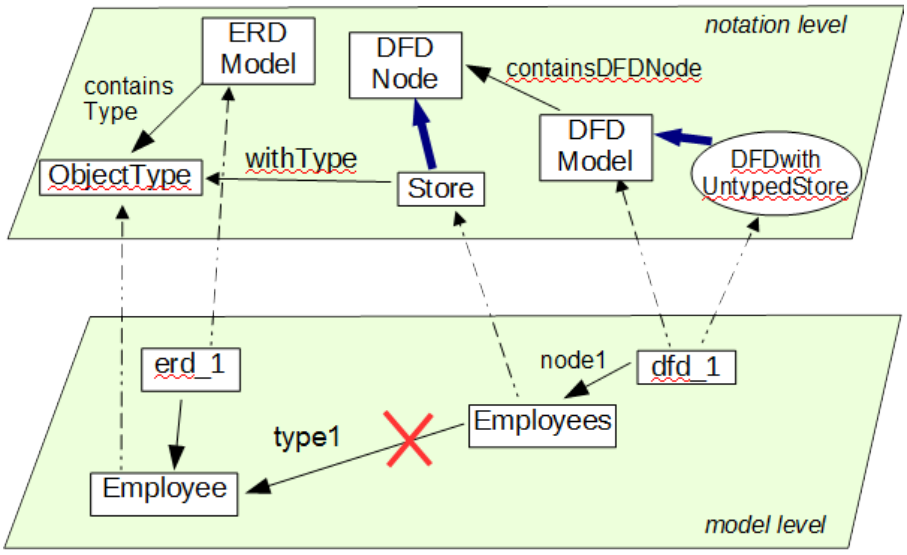

Figure 3.8: Internotational constraints at notation level

3.15.6 Multi-level statements to express semantics

The queries discussed above constrain the set of acceptable models of an information systems development method. A violation of some intra- or inter-notational constraint is detected just by analyzing the models. We can regard the set of all notational constraints as the *syntax* of the modeling notations. A characteristic of the query classes for notational constraints is that their variables range over objects at the IRDS model level

Figure 3.8 displays the query `DFDwithUntypedStore` of the last section. The query is an

example of an inter-notational constraint. It is defined at the notation level: it is a subclass of `DFD_Model` and refers to objects defined at the notation level. Its variables like `~this` and `store` are ranging over objects at the model level. The query returns those DFD models that violate a certain inter-notational constraint, namely that all data stores of a DFD must be linked to an object type in an ERD model belonging to the same system development project. All intra- and inter-notational constraints share the property of being *defined at the notation level and having variables ranging over objects at the model level*. However, there are more ways to express knowledge about modeling notations, namely to express *semantics* of certain notational symbols.

In predicate logic, the semantics of a logical theory, i.e. a collection of logical formula, is based on sets of objects. In particular, mathematical relations with matching arity interpret predicates. In the restricted framework of Datalog, predicates are interpreted by sets of facts where variables in the predicates are replaced by constants. Consider the example predicate `In(x,Employee)`. A possible interpretation *ext* computed by the fixpoint algorithm for Datalog could be:

ext`(In(x,Employee)) =`
`{In(bill,Employee), In(mary,Employee)}`

Remembering that Employee is the name of a class, we can say: the interpretation of the class `Employee` is the set `{bill,mary}`. Analogously, attribute predicates can be interpreted by relational facts:

ext`(A(e,salary,s)) =`
`{A(bill,salary,1000),A(mary,salary,2000)}`

We can easily extend this type of semantics to the other IRDS abstraction levels, e.g. the semantics of the predicate `In(e,EntityType)` is computed by the Datalog fixpoint mechanism and yields sets like

ext`(In(e,EntityType)) =`
`{In(Employee,EntityType),`
`In(Project,EntityType}`

There are however properties of notations that do not fall directly into this simple kind of semantics. For example, the cardinality of role links in the ERD notation is checked at the data level while the semantics of cardinality tags like "1:n" ought to be defined at the notation level. A solution to this problem is the use of meta-level formulas. They are formulated at a meta class level can have variables ranging over objects that are instances of instances of the objects occurring in the formula. Figure 3.9 shows the situation with a relationship type `worksFor` whose role link to `Employee` has a "1:n" cardinality. The example in the data level violates the cardinality. Hence, the semantics is dependent on the database content while we consider cardinalities to be part of the notation. The following meta-level constraint defines the semantics:

```
Cardinality in Class with
  constraint
    c1: $ forall R1,R2/RelationshipType!role
                 RT/RelationshipType ET1,ET2/EntityType
                 x/VAR
                 From(R1,RT) and To(R1,ET1) and
                 From(R2,RT) and To(R2,ET2) and
                 not (R1 == R2) and
                 (R1 card "1..n") and (x in ET2)
            ==>
              ( exists  y/VAR r1,r2/VAR rt/VAR
                 (r1 in R1) and (r2 in R2) and
                 (rt in RT) and (y in ET1) and
                 From(r1,rt) and To(r1,y) and
                 From(r2,rt) and To(r2,x)
              )
        $
end
```

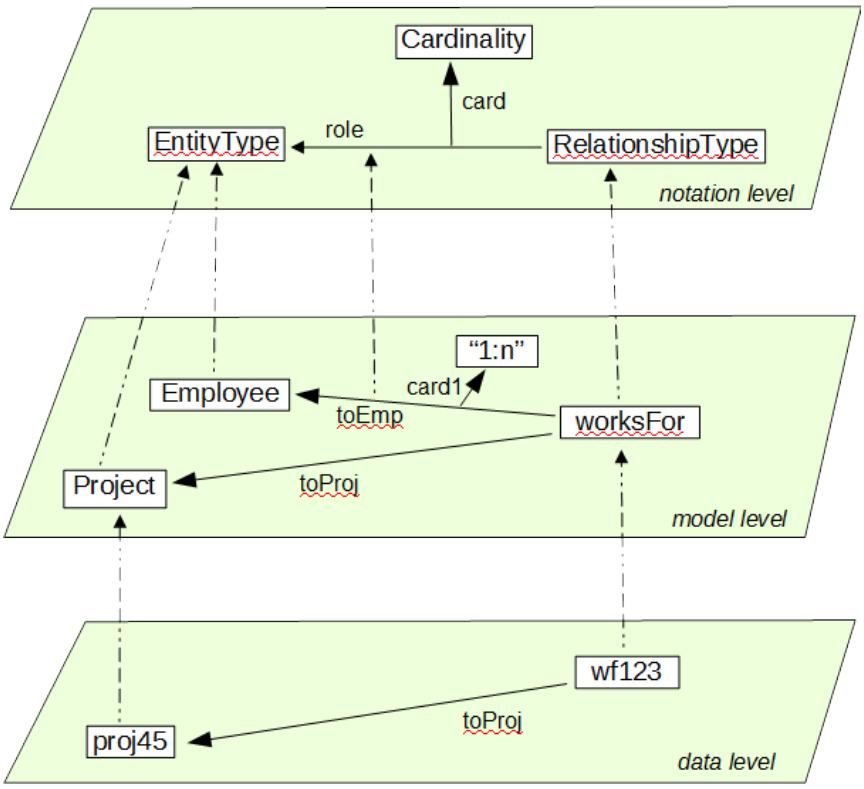

Figure 3.9: The semantics of the notational concept ‘cardinality’

The constraint refers to the notation level concepts `EntityType`, `RelationshipType`, `card` and `"1:n"`. All other objects are ranged over by variables. The variables `R1`, `R2`, `RT`, `ET1`, and `ET2` are ranging over objects at the model level. Possible values for `ET1` and `ET2` are `Employee` and `Project`, respectively. A possible value for `RT` is `worksFor`. The variables `x`, `y`, `r1`, `r2` and `rt` are ranging over objects at the data level. For example, a possible value for `x` is `proj45`. So, whenever an

object like `proj45` (`x`) is defined to be an instance of `Project` (`ET2`), then there must be at least one instance `y` of class `Employee` (`ET1`) that stands in the `worksFor` (`RT`) relationship to it. The cardinalities "1:1", "0:1" etc. can be expressed in a similar way. ConceptBase will partially evaluate the meta-level formulas to compile them into a set of simple formulas ranging over just one IRDS level as explained in the subsection on meta-level formulas.

Another example on the use of meta-level formulas is the definition of the disjoint specialization in UML class diagrams (see section 1.3 and figure 1.5). UML has a variant of class specialization that demands any two subclasses of a superclass to be disjoint, i.e. no object x may be an instance of two such classes at the same time. While the specialization in UML is defined at the notation level, the object x is part of the data level. The Telos realization of the semantics of the disjoint specialization is contained on the accompanying CD-ROM.

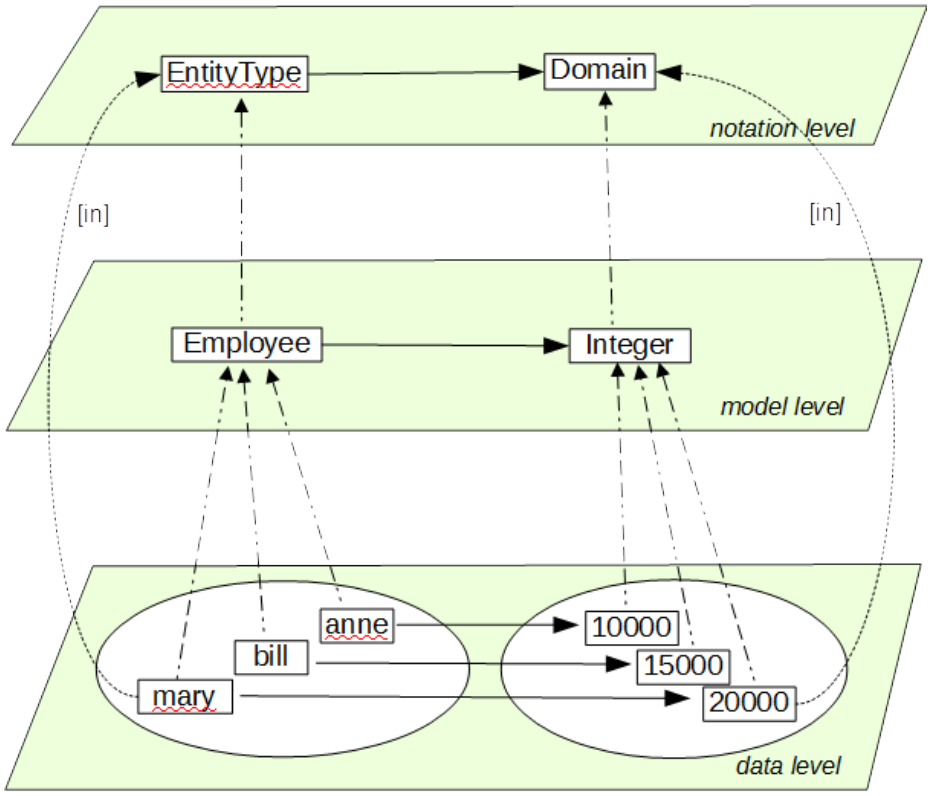

Figure 3.10: Understanding the data level from the notation level

Besides specifying the semantics of notational link symbols, meta-level formulas create opportunities to clarify the semantics of node symbols as well. Consider again the class `EntityType` defined at the notation level. Its Herbrand interpretation *ext*(`EntityType`) is the set of all its instances. The instances of the instances of EntityType are called *entities* and they are located at the data level. Let us use the predicate `(x [in] mc)` to express that the object `x` is an instance of some instance of `mc`. Then, the concept of an entity is formalized as:

```
forall x/Individual c/EntityType
    (x in c) ==> (x [in] EntityType)
```

Analogously, the set of all *values* is the set of all objects that are instance of some instance of the class `Domain`:

```
forall x/Individual c/Domain
    (x in c) ==> (x [in] Domain)
```

Both formulas are structurally derived from the generic formula

```
forall x/Individual c/Individual mc/Individual
    (x in c) and (c in mc) ==> (x [in] mc)
```

By coding the generic formula in ConceptBase, one can query the data level independently from the model level. In the ERD case, one can ask for all *entities* or all *values* that are currently known **regardless to which entity type or domain type they belong**. Even more, one could ask for all entities that are related to some other entity regardless of the underlying ERD model, i.e. database schema. An example is shown in figure 3.10: the currently known *entities* are `mary`, `bill` and `anne`. The *values*, i.e. instances of instances of `Domain`, are `10000`, `15000`, and `20000`. A ConceptBase implementation of the predicate `(x [in] mc)` can be found on the accompanying CD-ROM.

Please note that not all meta-level formulas are eligible for partial evaluation. The ERD example shows that the semantics of certain notational symbols can be expressed by constraints in Telos. Moreover, the concepts at the notation level can be used to directly query the objects at the data level. The semantics of dynamic models like data flow diagrams are more difficult to capture since they express complex transformations. Mapping models of such notations to simulation environments appears to be more suitable to capture their semantics. Alternatively, they can be analyzed by formal methods such as axiomatically defined partial functions.

3.15.7 Modeling process model notations

The notations discussed so far are the languages to record models about some artifact. The models are restricted by constraints, some of them ranging over models recorded in different notations. There is another relevant connection between models: some models are created out of others. In the case of system development, the code of a program module is constructed from some module chart and the specification of the processes that are part of the module. There is some *process model* that tells the modeler, out of which input models a new output model can be created. We regard such a process model to be part of the engineering of a method.

Process models can be *descriptive* (a structured documentation of what steps have been performed during a modeling project) or *prescriptive* (a set of rules that define which development steps are allowed in a given situation). Process models play a major role in any modeling method because they are the basis to refine the modeling method as a whole and to define new services like the traceability of model details.

A specialty of process models is that they apparently blur abstraction levels. Consider the following excerpt from a fictional software development project:

10-Oct-2002, 10:23: Mary has completed interviews with the user group and publishes her interview report REQ14.doc.
12-Oct-2002, 15:56: John has read the report REQ14.doc and produces the event list EL1 out of it.
13-Oct-2002, 9:45: John produces the preliminary behavioral model MyDFD1 out of EL1.

The statements themselves are concrete (i.e. data level in the IRDS framework) and cannot be instantiated. They constitute the *trace* of an actual modeling project. On the other hand, some parts in the trace are referring to concepts that we previously classified into the model level: the report REQ14.doc, the event list EL1, and the preliminary behavioral model MyDFD1 are all models whose content are at class level (i.e. they do have instantiations). The reason for this mixture of levels is that the very activity we are observing is producing models not concrete data. The concrete statements of a process trace follow some patterns. The patterns for the above example could be:

An interviewer completes interviews and publishes interview reports.
An analyst reads interview reports and produces event lists.
An analyst produces preliminary behavioral models out of event lists.

A collection of such statements is called a *process model* (for modeling projects). We can continue this abstraction and conclude that all three statements of the process model are examples of the generic statement:

An expert reads models and creates models.

The last statement is defining how process models shall look like; hence it is part of a *notation for process models*. Note that the term ‘Model’ is occurring in the notation definition level of the modeling products. Hence, the mixture of levels that was observed at the trace continues with the more abstract statements.

As such, the statements about model production have the same nature as the statements that were used to describe various abstractions at the product side (data, models, notations, notation

definition levels). The extra property is that they relate in a new way to objects defined at the product side. Figure 3.11 relates production-oriented process models (right side) to their products (left side). The known IRDS abstraction levels are applicable to process modeling as well but the two sides are *skewed* by one level against each other. The reader should note the similarity with the fact `(Entity Type attribute/author Petersen)` discussed in the first part of this section. There like in this case, objects from different IRDS levels are associated with each other. The origin for the skewing of levels lies in the nature of models. A model can on the one hand be seen as a collection of classes whose instances are possible interpretation of the classes. On the other hand, the objects in a model are input/output of some development steps, i.e. they are data themselves.

The companion CD-ROM contains in folder SPM an example for software process modeling, i.e. a notation plus examples on how to specify the modeling steps of a notation. The example also features an almost complete representation of the notations of the Yourdan method. The reader is advised to take the definitions as an example: they show how the process model integrates with the notations defined earlier. The notational constraints are realized as query classes and can be attached to individual modeling steps: whenever such a step is executed, the specified constraints should be checked.

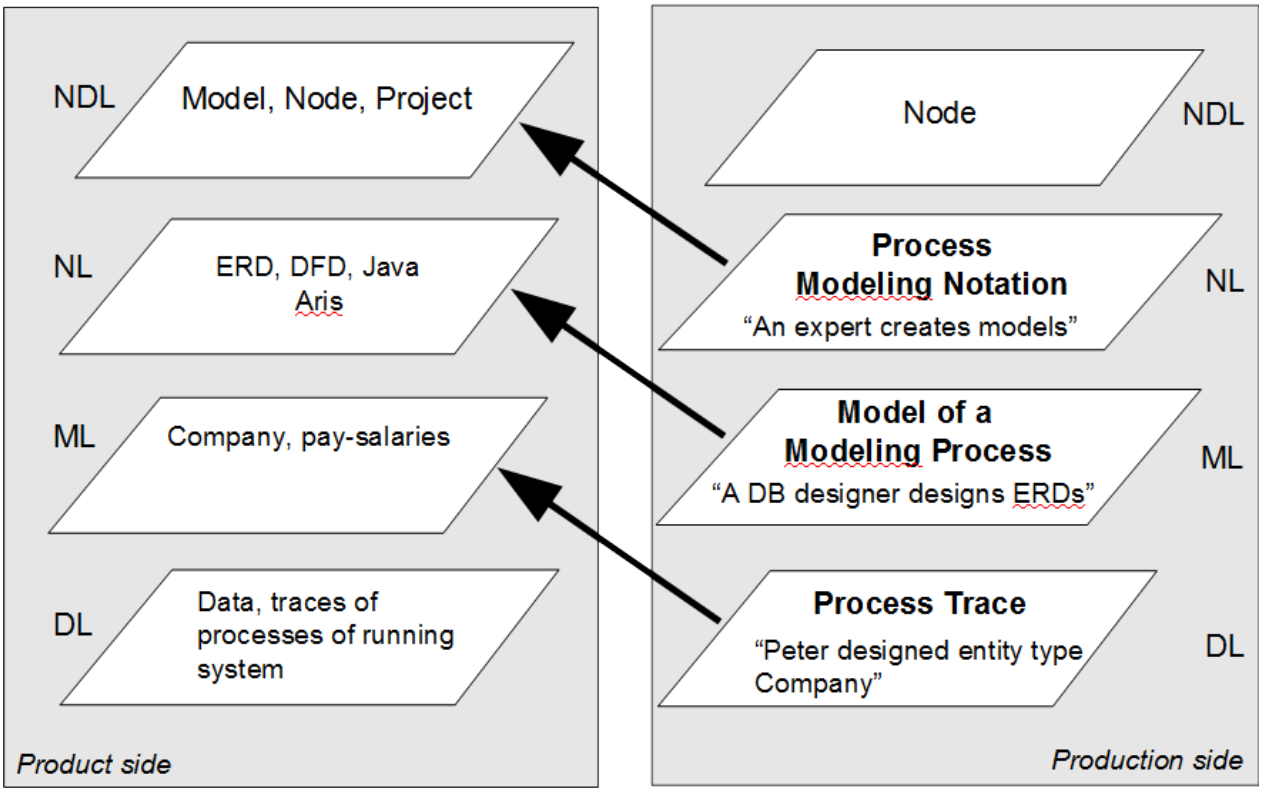

Figure 3.11: Production and product side of modelling.
NDL: notation description level; NL: notation level; ML: model level; DL: data level

Simplified notation level for the process model

```
     ModelingStep in Node with
       connectedTo
         precedes: ModelingStep
       attribute
         inputObject: Node;
         outputObject: Node;
         postcondition: QueryClass
     end

     Methodology with
       attribute
         containsStep: ModelingStep
     end

     Project with
       attribute
         produces: Model;
         uses: Methodology
     end
```

Model level for the process model (excerpt)

```
YourdanProject in Project with
  produces
    eventlist: EventList;
    prelimbehavioralmodel: PreliminaryBehavioralModel;
    processspecification: ProcessSpecification;
    erd: ERD;
    std: STD;
    sourceProgram: SourceProgram;
    testPlan: TestPlan;
    testData: TestData;
    testResult: TestResult
  uses
    methodology: YourdanMethod
end
```

```
YourdanMethod in Methodology with
  containsStep
    extractEventList: ExtractEventList;
    mapToDFD: MapToDFD;
    specifyProcess: SpecifyProcess;
    codeProcess: CodeProcess;
    createTestPlan: CreateTestPlan;
    executeTest: ExecuteTest
end

YourdanStep in ModelingStep with
  attribute
    input: Proposition;
    output: Proposition;
    starttime: Real;
    endtime: Real
end

ExtractEventList in ModelingStep isA YourdanStep with
  inputObject
    input1: UserRequirements
  outputObject
    result1: EventList
end

ProduceLeveledDFD in ModelingStep isA YourdanStep with
  inputObject
     prelimDFD: PreliminaryBehavioralModel
  outputObject
     levDFD: LeveledDFD
  postcondition
     r1: NotLeveledNotSpecifiedProcess;
     r2: BothLeveledAndSpecified;
     r3: DanglingInputNotMatchedAtProcess;
     r4: DanglingOutputNotMatchedAtProcess;
     r5: InputNotMatchedByDangling;
     r6: OutputNotMatchedByDangling;
     r7: IllegallyLeveledProcess
end
```

[source: spm.sml]

Data/trace level for the process model (excerpt)

```
step1 in ExtractEventList with
  result1 el: MyEvent_456
  starttime t1: 3.0
  endtime   t4: 4.0
end

step2 in MapToDFD with
  input1 el: MyEvent_456
  output1 dfd: MyDFD_023
  starttime t1: 5.0
  endtime   t4: 7.0
end
```

3.15.8 Managing modeling processes by metrics

The uniform representation of the development trace, the software process model (here: Yourdan example), and the process model notation together with the product counterparts (example models, modeling notations, notation definition level) allows analyzing complex modeling situation by means of queries. In particular, one can express metrics on any of the objects defined in the repository. As example, consider the metric `CodingProductivity`, which measures the lines of code created per time unit in a coding step. The following query realizes the metric in terms of two other metrics `Duration` and `ProgSize`:

```
ProgSize in GenericQueryClass isA Integer with
  parameter,computed_attribute
    whatProg: SourceProgram
  constraint
    c: $ (~this in
           COUNT_Attribute[~whatProg/objname,
                    SourceProgram!lines/attrcat]) $
end

Duration in GenericQueryClass isA Real with
  parameter,computed_attribute
     step : YourdanStep
  constraint
     c1 : $ exists t1,t2/Real (~step starttime t1)
            and (~step endtime t2)
            and (~this in MINUS[t2/r1,t1/r2]) $
end

CodingProductivity in GenericQueryClass isA Real with
  parameter,computed_attribute
     cstep: CodeProcess
  constraint
     c1: $ exists sp/SourceProgram dur/Real
                  size/Integer
             (~cstep program sp) and
              (dur in Duration[~cstep/step]) and
             (size in ProgSize[sp/whatProg]) and
             (~this in DIV[size/r1,dur/r2]) $
end
```

[source: metric.sml]

The queries exploit built-in queries (Jarke, Jeusfeld, Quix 2003) of ConceptBase for simple arithmetic (PLUS, MINUS, DIV). The combination of the attribute categories parameter and computed_attribute makes sure that the query answer can be interpreted in case the parameter is left undefined at query call time. For example the query `CodingProductivity` (without filler for parameter `cstep`) will return the coding productivities of all coding processes (attached as computed

attribute `cstep`). The metric queries can be integrated into the process models as post-conditions of modeling steps very much like the queries for checking notational constraints. More insights on the use of metrics in software development are in (Fenton, Pfleeger 1999).

Chapter 8 contains more details on how metrics can be incorporated into design processes supported by the ConceptBase repository. The application there is a repository for managing the quality of data warehouse systems.

3.16 Discussion and conclusions

This chapter presented the use of the ConceptBase repository for meta-modeling and the engineering of modeling methods. The approach as presented here is focused on *notation definition* and one may ask the question whether this is sufficient to cover all aspects of a complete method like Yourdan's Modern Structured Analysis. The syntactic features of the Yourdan notations for entity-relationship diagrams, data flow diagrams, process specifications, state transitions diagrams etc. can rather easily be translated into suitable Telos meta classes. *Correctness rules* (called balancing rules by Yourdan) map straightforward into query classes. There are however a few *soft rules* that are naming conventions to make models easier to read for humans, e.g. the name of a process should be a verbal phrase. Such rules are vague by nature of human communication and a violation of such soft rules is acceptable to a certain degree.

Another aspect of method engineering is instructional examples. All textbooks on modeling methods are cluttered by examples that demonstrate a feature. This is in fact easily supported by the repository approach since *example models* are just one IRDS level below the notations and are represented in the same Telos frameqork as the notations themselves. They have an even bigger role with our approach, as the relation of example models to notations is now fully formalized as a Telos instantiation relation. During the design of a new notation, the creation of example models is validating the correctness/usefulness of certain notational constructs. Hence, not only the student of a method is supported but also the method engineer in her search for the right notation definition.

A method description is incomplete when it doesn't contain some kind of step-by-step instructions on how to proceed with a modeling project. *Process models* as presented above can cover this aspect. The integration of correctness rules as post-conditions to the modeling steps goes beyond the original description of the Yourdan method. Originally, the rules are formulated as part of the respective notations. With process models, one can precisely design when to perform certain checks on the models created so far.

Finally, the *quality of a collection of models* can be managed by metric definitions. This feature of ConceptBase covers some of the soft aspects of model development (in teams). In the Yourdan case, there are rules on the desirable complexity of diagrams that easily map into metric definitions. The same technique applies to controlling the quality of the modeling steps, e.g. by measuring their productivity.

So what remains uncovered? Insights on *proper usage* of modeling techniques are hard to represent as part of a notation. The proper usage depends on the knowledge of the application domain and the cognitive skills of the modeling expert. Modeling textbooks cover this by including case studies that are extensively commented and where reasons for certain design decisions are elaborated. One way to support experience knowledge with the repository approach is to keep the results of earlier modeling projects within the repository. Then, novice modelers can search in the archive of old projects for solutions to certain problems. Of course, the models of old projects would need to be further annotated and indexed. Moreover, design decisions should be made an integral part of the process models and include the rationale behind them.

A frequently cited aspect of modeling methods are their *pragmatics*, i.e. what resources are needed to apply them, in particular how much time should be spent on teaching the methods. If individual modeling steps have known requirements for resources, then they can be easily included. Essentially, one has to create a plan for project management where resources, time, and cost play some important role. That plan can be encoded with a Telos-based notation for *workflow management*. However, such a notation doesn't really solve the problem. The problem is the allocation of spare

resources, e.g. installing a help desk for questions on a modeling method.

Finally, the decision to uses Datalog as a foundation for the semantics deserves a critical discussion. The perfect model semantics of Datalog is excluding any formal ambiguity in the interpretation (via stratification) and is always finite. The latter restriction forbids a complete coverage of the semantics of dynamic models such as state transition diagrams or program code. Still, the syntactic features of such notations and their inter-relationships with other notations can well be represented. A semantics specification powerful enough for dynamic modeling notations would stand in conflict to the effeciency of the model management.

Perfect model semantics essentially allows to answer queries, i.e. find out which objects of a given finite extension fulfill the membership condition of the query. It does not support reasoning about elements of a model, e.g. whether some class definition is subsumed by another as in section 1.3 on formal modeling languages. One can argue that such reasoning services can be attached to the model repository as external tools.

References

S. Ceri, G. Gottlob, L. Tanca (1990) Logic Programming and Databases, Springer-Verlag, Heidelberg, Germany.

W. Chen, D.S. Warren (1996) Computation of Stable Models and its Integration with Logical Query Processing, TKDE (5):742-757.

N.E. Fenton, S.L. Pfleeger (1999) Software Metrics – A Rigouros and Practical Approach. Second Edition, Revised Printing, PWS Publ., Boston, MA.

S. Greenspan (1984) Requirements Modeling: the Use of Knowledge Representation Techniques for Requirements Specifications. Ph.D. dissertation, University of Toronto, Ontario, Canada.

M. Jarke, M. Jeusfeld, C. Quix (eds., 2003) ConceptBase V6.1 User Manual, Online http://www-i5.informatik.rwth-aachen.de/CBdoc/userManual/.

M. Jeusfeld (1992) Änderungskontrolle in deduktiven Objektbanken. Infix-Verlag, St. Augustin.

J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis (1990) Telos -- a Language for Representing Knowledge about Information Systems. In ACM Trans. Information Systems 8(4):325-362.

N.W. Paton (1999) Active Rules in Database Systems. Springer-Verlag, New York.

E. Yourdan (1989) Modern Structured Analysis. Prentice Hall, Englewood Cliffs, NJ.

Footnotes

1 Telos statements can be expressed as binary predicate facts in infix form. Basically, the statement number is omitted because it is system-generated and carries no meaning as such. At a later stage, it is shown how to map the predicate facts into a reified format, called the P-facts. The reified form is crucial for defining the semantics of a set of Telos statements, and for providing facilities for engineering the semantics of modeling notations.

2 The meta class `DomainOrObjectType` is a superclass of both `ObjectType` and `Domain`. Its proper definition is in the subsequent subsection on Telos frame syntax.

3 The latter object is a pre-defined Telos object with name InstanceOf whose definition can be looked up in section 3.8. There are in total just 5 pre-defined Telos objects.

4 The advantage of explicit quantifiers is that one can formulate also logical expressions with nested quantifications. It can be shown that any such formula can be transformed to a set of Datalog-style deductive rules that have the same semantics.

5 There are however non-stratifiable rule sets that are not at all paradoxical. They even have a unique model. We refer the reader to articles describing stable model semantics for more information.

6 The constraint text itself is an instance of MSFOLconstraint. We omit the Telos frame, which defines this instantiation. ConceptBase will include this instantiation automatically. The same holds for rule texts, which are automatically classified into the class MSFOLrule.

7 The query class with the explicit variable 'this' is shown for illustration only. It does not represent a syntactically correct query class. Note that the variable 'this' ranges over all instances of 'Department', i.e. the superclass of the query class.

8 The predicate `(p in Propositions!single)` is translated from the variable `range p/Proposition!single`. See also syntax definition for formulas in Telos.

9 Full-fledged active database systems provide more event types and a complex event composition language to express sequences of events `(e1 after e2)`, logical connectives `(e1 or e2)`, and temporal events. More on active databases can be found in (Paton, 1999).

10 We do not discuss the data dictionary notation in great details here.

11 One can regard such rules as part of the syntax definition for a notation. The syntax circumscribes the set of allowed statements (=models) in a given notation.