1 Introduction

This tutorial extends the first tutorial by examples on metamodeling, i.e. to scenarios where you define objects, classes, and meta classes. Metamodeling is particularly useful in situations where you need to define your own modeling languages (domain-specific languages). You will see that you can use the ConceptBase query language to analyze the models created in your dedicated modeling languages and that it is rather easy to define simple modeling languages. Solutions to the exercises are at the end of this tutorial.

1.1 The Scenario

We start with a simple version of entity-relationship diagrams. First, we will define entity types and relationship types (meta classes). Then, define an example entity-relationship diagram (classes) plus some example data (objects).

In the next part, we add a simple process language to the existing entity-relationship language. We are interested in analyzing process models. In particular, we want to check whether one agent is responsible for two tasks $t_1$ and $t_2$, and there is a task $t$ on the path between $t_1$ and $t_2$ that is assigned to another agent.

1.2 Start ConceptBase

There are several methods to start the ConceptBase server and its user interface CBIva. We decide for the simplest way: start the ConceptBase server from within CBIva. So, switch to the directory to which ConceptBase is installed on your computer and start the ConceptBase.cc user interface CBiva:

```
  cbiva
```

On Windows and Mac OS-X you can also start CBIva by double-clicking on the command file cbiva[.bat] in the directory where you installed ConceptBase.

Figure 1 shows the CBIva window just after starting it. If the indicator on the bottom left corner is green and has the label "Connected", then your CBIva client is auto-connected to a ConceptBase server. If it is red and displays "Disconnected", then the CBIva is not yet connected to a ConceptBase server. Press the "connect" icon just below the "File" menu in such cases to start/connect to a local ConceptBase server.
2 Define a simple Entity-Relationship notation

**Exercise 1:** The task is to define two classes `EntityType` and `RelationshipType`. The class `RelationshipType` shall have an attribute `role` with value `EntityType`.

```
EntityType end
```
```
RelationshipType with
  attribute
    role: EntityType
end
```

Enter the definitions into the *Telos Editor* window and store them to the ConceptBase server with the "Tell" function.

**Exercise 2:** Add to `EntityType` an attribute `attr` with value `Domain`. Also define `Domain` as an object without attributes.

```
EntityType with
  attribute
    attr: Domain
end
```
```
Domain end
```
This provides us with a very simple entity-relationship language. It just allows to define entity types with attributes, and relationship types with role links. Entity attributes are restricted to domains. So we need to specify the allowed domains.

**Exercise 3:** Specify Integer and String as domains, i.e. as instances of the class Domain.

```
Integer in Domain end
String in Domain end
```

The classes Integer and String are predefined in ConceptBase. Any integer number occurring in an object definition will automatically be an instance of Integer. Likewise any double-quoted string will be regarded as an instance of String.

**Exercise 4:** Specify a new domain Date. Include “2009-05-19” and “2001-01-01” as two possible values for dates.

```
Date in Domain end
"2009-05-19" in Date end
"2001-01-01" in Date end
```

The object Date is not predefined in ConceptBase. Hence, we need to take care ourselves about the set of possible values (=instances of Date).

After these exercises, you can visualize the current state with the graph editor. Use RelationshipType as start object. The graph editor is started from CBIva via the menu item "Browse / Graph Editor". Expand the outgoing attributes of RelationshipType (right mouse button) and select “Show all”. Do the same with EntityType. For Domain, show the instances. For Date, show the instances as well.

The graph window shows already three abstraction levels: the objects “2009-05-19” and “2001-01-01” are at the lowest abstraction level (data level). The objects Date, Integer, and String are classes (model level), and the objects RelationshipType, EntityType, and Domain are meta classes (notation level).
3 Define an Entity-Relationship model

Exercise 5: Specify an example ER diagram for an insurance scenario. An insurance policy has a customer, a premium, a start date, and an end date. Customers have names and addresses. A claim has a description and is referring to an insurance policy.

Customer in EntityType with
  attr
    name: String;
    address: String
  end

Policy in EntityType with
  attr
    startdate: Date;
    enddate: Date;
    premium: Integer
  end

holds in RelationshipType with
  role
    customer: Customer;
    policy: Policy
  end
Claim in EntityType with
tag
    description: String
end

class_claim_policy in RelationshipType with
  role
    claim: Claim;
    policy: Policy
end

Figure 3 graphically displays the insurance model. ConceptBase can also assign dedicated graphical symbols to certain objects, e.g. diamond shapes to relationship types. We skip this feature in this tutorial and refer you to the user manual for more details on this.

The green links are instantiations. Hence the insurance model is one abstraction level below the ER language.

Exercise 6: Enter data objects for the following facts. Customer mary signed an insurance policy with start date "2009-05-19" (no end date). The premium is 1000.
mary in Customer end
policy1 in Policy with
 startdate d: "2009-05-19"
  premium p: 1000
end

holds1 in holds with
  customer c: mary
  policy p: policy1
end

Figure 4: Sample data for the insurance model

The display of the data objects in figure 4 completes all three abstraction levels (meta classes, classes, data objects).
5 Define a process modeling notation

Process models can be used to denote workflows, business processes, and algorithms. We are in particular interested in a process modeling notation that allows us to analyze process models for certain patterns. Before we start defining the notation, we define the transitivity construct that shall be useful subsequently for defining the pattern.

Proposition in Class with
attribute
transitive: Proposition
rule
trans_R: $ \forall x,y,z,R/VAR$
\begin{align*}
A/C/Proposition!transitive & C/Proposition \\
P(A,C,R,C) & \text{ and } (x \text{ in } C) \text{ and } (y \text{ in } C) \text{ and } (z \text{ in } C) \text{ and } \\
A_e(x,R,y) & \text{ and } (y \ R \ z) \implies (x \ R \ z)$
end

The predicate $A_e(x,R,y)$ is true if there is an explicit attribute between objects $x$ and $y$ that has the category $R$.

Exercise 7: Define a process notation that allows tasks to be defined. Tasks can have successor tasks. Agents execute tasks. The successor relation shall be transitive.

Task with
attribute,transitive
successor: Task
end

Agent with
attribute
executes: Task
end

The process modeling notation is very simple but it has the ability to represent very complex workflows. Let now distinguish start statements and predicate statements.

Exercise 8: A start statement is a task that has no predecessor (no task has a start statement as successor). A predicate statement is a task that has more than one successor. Define these concepts as query classes.

StartStatement in QueryClass isA Task with
constraint
\begin{align*}
c_1: & \ \exists \ t/Task \ (t \ successor \ this) \\
end

PredicateTask in QueryClass isA Task with
constraint
\begin{align*}
c_1: & \ \exists \ s1,s2/Task \ A_e(this,successor,s1) \ \text{and} \\
& \ A_e(this,successor,s2) \ \text{and} \ (s1 \neq s2)$
end
You can define end statements in a similar way. A more tricky concept is the following.

**Exercise 9 (difficult):** Define the concept of a loop task, i.e. a task that is part of a loop. The name of the query shall be `LoopTask`.

```
LoopTaskOf in GenericQueryClass isA Task with
  parameter
  rep: Task
  constraint
    c: $ (this successor rep) and (rep successor this) and
        (exists s/Task A_e(rep,successor,s) and (s successor rep)) $ $ end
```

```
LoopTask in QueryClass isA LoopTaskOf end
```

The parameter `rep` in the first query stands a representative of a loop. Note that there may be many loops inside a process model and we would like to be able to query, which tasks belong to the same loop. The second query just returns all loop statements regardless of the representative. It is sufficient to leave out a value for parameter `rep` in this case.

There can be several loops inside a process model. Loops can also be nested, i.e. a task can be member of several loops. Note that the regular attribution predicate `(t1 successor t2)` is closed under transitivity!

Now that we have defined loops, let us tackle the pattern "agent with split responsibility".

**Exercise 10 (difficult):** Assume that an agent A is responsible for tasks t1 and t2 in a process model but there is a task t between t1 and t2 that is executed by another agent. This matches situations where an agent does some work, then passes control to another agent, and afterwards resumes control. Define this pattern as a query class named `AgentWithSplitResponsibility` that returns agents with split responsibility.

```
AgentWithSplitResponsibility in QueryClass isA Agent with
  constraint
    c1: $ exists t1,t2,t/Task a/Agent (this executes t1) and
        (this executes t2) and (t1 successor t) and
        (t successor t2) and (a executes t) and (a \= this)$ $ end
```

The condition `(a \= this)` makes sure that the middle task `t` is executed by a different agent.
6 Define an example process model

Recall the insurance scenario. Now we need to represent a workflow in this domain with our newly defined process modeling notation.

**Exercise 11:** Claim handling starts with an insurance agent receiving the claim. Afterwards, the policy is checked. Afterwards, either a payment is proposed or an assessor is assigned. The assessor assesses the damage. On that basis, the insurance agent proposes a payment. After proposing the payment, we either can continue with processing the payment (customers accepts the proposal), or we need to iterate i.e. check again the policy and possibly reposepose a new payment. The workflow is finished after processing the payment.

```plaintext
start in Task with successor n: receiveClaim
end

receiveClaim in Task with successor n: checkPolicy
end

checkPolicy in Task with successor n1: assignAssessor; n2: proposePayment
end

assignAssessor in Task with successor n: assessDamage
end

assessDamage in Task with successor n: proposePayment
end

proposePayment in Task with successor accept: processPayment; reject: checkPolicy
end

processPayment in Task with successor n: finish
end

finish in Task end
```
Assessor in Agent with
  executes
    t1: assessDamage
  end

InsuranceAgent in Agent with
  executes
    t1: receiveClaim;
    t2: proposePayment
  end

**Exercise 12:** Ask the two queries LoopTask and AgentWithSplitResponsibility.

The answer to LoopTask is checkPolicy, assignAssessor, assessDamage, proposePayment. The answer to AgentWithSplitResponsibility is InsuranceAgent, Assessor. Note that the task assessDamage is in a loop with proposePayment. Hence, a sequence assessDamage-proposePayment-checkPolicy-assignAssessor-assessDamage is possible and is the reason to classify both agents into the query class AgentWithSplitResponsibility.

You can also visualize the results of the queries by the graph editor. The example process model together with the classification to the two query classes is shown in figure 5.

![Figure 5: Classifying a process model via query classes](image)

The dotted green links are derived instantiations. So, an object that is in the answer set of a query class is
regarded as a derived instance of that query class. Indeed, query classes are classes where the instances are
derived via the membership condition of the query class.
7 Link the two notations

We have created two simple notations, one for data modeling and the second for process modeling. Now let us combine these two. The most natural way appears to regard object types (entity types and relationship types) as possible inputs and outputs of tasks in a process model.

Exercise 13: Define a new construct ObjectType that generalizes EntityType and RelationshipType.

ObjectType end
EntityType isA ObjectType end
RelationshipType isA ObjectType end

So, this was easy. We now can link the two notations via ObjectType.

Exercise 14: Define object type as possible input/output of tasks in process models.

Task with
    attribute
        input: ObjectType;
        output: ObjectType
    end

Attributes in ConceptBase are by default multi-valued, i.e., they can have zero, one or many values. This is exactly what we want in this case.

We finalize this tutorial by attaching some objects types as input/output of tasks.

Exercise 15: Define some of the object types of exercise 5 as input/output of the process model of exercise 11.

receiveClaim with
    output o1: Claim
end

checkPolicy with
    input i1: claim_policy
end

8 Conclusions

In this tutorial, we defined two simple notations, one for data modeling, another for process modeling. We defined queries to analyze process models for non-trivial patterns, building on a newly defined construct for transitivity. We created example models for both notations. Finally, we linked the two notations to form an integrated method for data and process modeling.

The two notations were both very simple. For example, the ER notation lacks cardinalities of role links. The process modeling notation cannot represent parallel splits. Adding the missing construct would not require too much effort. The interested reader is referred to the CB-Forum (http://conceptbase.sourceforge.net/CB-Forum.html) for extended examples.