POSTPRINT -- Original paper appeared as: M.A. Jeusfeld: SemCheck – Checking Constraints for Multi-Perspective Modeling Languages. In D. Karagiannis, H.C. Mayr, J. Mylopoulos (eds): Domain-Specific Conceptual Modeling – Concepts, Methods and Tools. Springer Int. Pub., DOI 10.1007/978-3-319-39417-6_2, pp. 31-53, 2016.

SemCheck: Checking Constraints for Multi-Perspective Modelling Languages

Manfred A. Jeusfeld
University of Skövde, IIT, Sweden
manfred.jeusfeld@his.se

Abstract. Enterprises are complex and dynamic organizations that can hardly be understood from a single viewpoint. Enterprise modelling tackles this problem by providing multiple, specialized modelling languages, each designed for representing information about the enterprise from a given viewpoint. The OmiLab initiative promotes the use of meta-modelling to design such domain-specific languages and to provide them by an open repository to the community. In this chapter, we discuss how this meta-modelling approach can be combined with the design of integrity constraints that span multiple modelling languages. We propose the services of the ConceptBase system as a constraint checker for modelling languages created by the ADOxx platform.

Keywords: modelling language, consistency, Telos, ConceptBase

Introduction

Enterprise modelling environments[1] provide viewpoints for modeling goals, processes, resources, enterprise data, events, and more. Each viewpoint may be supported by several modelling languages, e.g. to support alternative representations or to cover different abstraction levels. The resulting enterprise models need to be synchronized since they all make statements about the very same reality, the enterprise. The goal of enterprise modelling is to provide a complete and correct representation of the enterprise, up to the level of detail that is of interest to the modelers. The completeness is rather time-consuming to check since it requires comparing the concepts in the models with observations of the (real) enterprise. The correctness has two aspects:

1. The statements derived from the enterprise models are consistent with the reality. For example, if a process model demands that an activity A always precedes an activity B, then this should hold for all executions of this process in the reality (*external model validity*).
2. The statements in the enterprise models are consistent with each other, or simpler: the statements in the enterprise models do not contradict each other (*internal model validity*).

We focus on the internal model validity in the OmiLab [1] context. OmiLab offers a repository, where multiple enterprise modelling languages can be stored and re-used. The ADOxx platform [2,3] used in OmiLab supports both the design of customized modelling languages and their subsequent use. The challenge is to co-design the consistency rules for the new modelling languages. In particular, the constructs of several interrelated modelling languages are subject to consistency checks. For example, if a process model contains a data flow link that retrieves a certain data element from a data store, then the data model of that data store must also define this data element.

The rest of the paper is organized as follows. The next chapter discusses why constraints crossing multiple modeling perspectives occur in enterprise models. We argue for two types of such constraints: one is linking models at the same abstraction level, e.g. the business level. We call them horizontal constraints. The second type links models at different abstraction levels. We call them vertical constraints. After this discussion, we introduce the constraint checking capabilities of the ConceptBase [4,5] system. ConceptBase allows representing

[1] This work was supported in part by the Erasmus+ project Open Model Initiative (OMI).

both multiple modeling languages and their models in a uniform data structure. Finally, we propose the architecture to integrate the ADOxx platform with ConceptBase. The interaction between the two is described by a simple model exchange interface. The ADOxx platform can incrementally submit language and model update to the ConceptBase constraint checker and receives constraint checking results back. New constraints can be added at any time and old ones can be updated or removed at any time. The examples discussed in this chapter are available online at http://conceptbase.cc/nemo2015.

Constraints between multiple models

Models describe some real or imagined reality, so enterprise models describe an existing or not yet existing enterprise. A model consists of model elements, which represent some physical or immaterial artefact of the reality. Consider an enterprise that maintains a database DB. This immaterial artefact occurs in multiple models. It may occur in a process model as input or output of some process task. It may also occur in a conceptual data model like an ER diagram defining the schema of the database. And it may occur as logical database design defining the precise datatypes of the database.

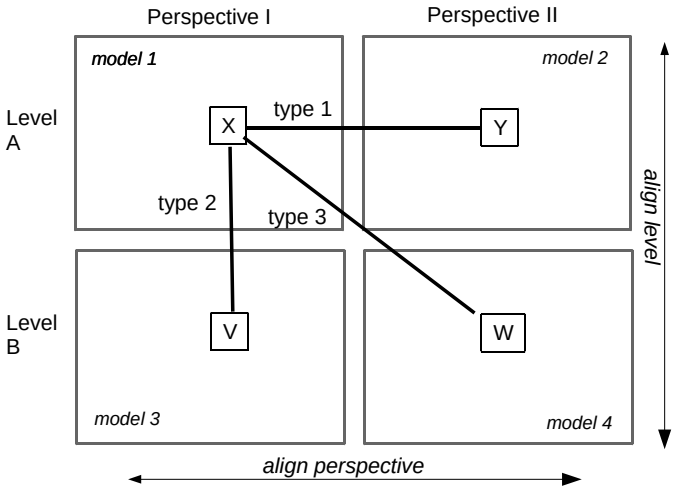

Fig 1: Links between levels and perspectives

Enterprise models cover multiple perspectives (data, process, goal, …) and abstractions levels (business, applications, technology, …). Links between perspectives relate model elements that are represented in different models but still need to be synchronized. For example, a database model element in a BPMN process model is related to the data model that defines the classes stored in that database (link type 1 in figure 1). Link type 2 relates model elements that make statement about the same artefact but uses different levels of detail to do so. This type of link is an "**implementation** link". For example, the relational database schema describes the same database as a UML class diagram but at different level of detail and usually committing to a specific way of implementing. Finally, there may be links of type 3 that change both the perspective and the abstraction level.

Subsequently, we first discuss the constraint language as implemented by ConceptBase. Then, we discuss the types of constraints crossing multiple enterprise models using examples from 4EM [6] and ArchiMate [7].

Constraint checking with ConceptBase

ConceptBase is a deductive database systems specifically designed to manage models and modeling languages. Constructs describing modeling languages are represented in the very same data structure that is used to represent models and even data. The underlying data model of ConceptBase is Telos [8] and the common data structure is the P-fact P(o,x,n,y) ("the statement o establishes a relationship with label n between the statements x and y").

The P-fact data structure is used to store models at any abstraction level. The statements identifiers are generated by the system and carry no semantics from the modeled reality. Thus, we shall use the textual frame syntax of ConceptBase for example. The first example shows how to define a small ER language, use it for a model on employees and projects, and define some data object. The first two frames define the meta-classes "RelationshipType" and "EntityType". They form a meta-model in the ADOxx terminology and would be part of an M2 model in OMG's classification. The subsequent three frames represent part of a conceptual model (OMG level M1) defined in terms of the meta-model. The last frame on "bill" is at the OMG M0 level.

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

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

bill in Employee end
```

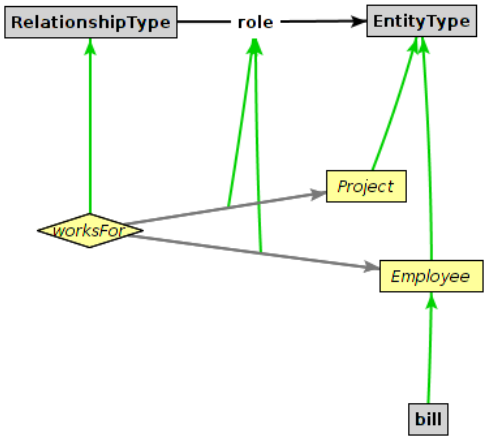

Fig . 2: Example ConceptBase model spanning three abstraction levels

Figure 2 displays the example model as a graph. The green links are instantiations. Note that instantiation applies not only two node objects like "bill", "Employee", "EntityType" but also to link objects like the role link of "RelationshipType". The uniform representation of objects, classes, and meta-classes allows to specify rules and constraints at any of the abstraction levels. There are no explicit abstraction levels in ConceptBase but rather instantiation relations between objects.

ConceptBase implements a rule and constraint language based on Datalog [9]. Since statements at any abstraction level are represented in the same way, one can also define rules and constraints at any abstraction level. The syntax of the rule and constraint language follows is a first-order predicate logic, where variables are bound to class objects, i.e. they range over the instance of the class objects. The most important predicates are

(x in c): The object x is an instance of the class object c, for example (bill in Employee)
(c isA d): The object c is a specialization of the object d, for example (Manager isA Employee)
(x m/n y): There is a link with label n between x and y and this link has the category m, for example (worksFor role/toEmp Employee)

(x m y): There is a link between x and y and this link has the category m, for example (worksFor role Employee); this predicate is derived from the previous one

Links are treated as objects. The expression worksFor!toEmp references the toEmp link of Employee. A complete list of predicates is available from the ConceptBase user manual [10]. To continue the example, we define two constraints, one at modeling language level and the other at model level:

```
forall R/RelationshipType exists E/EntityType
    (R role E)

forall e/Employee exists p/Project w/worksFor
    (w toEmp e) and (w toProj p)
```

The two formulas realize multiplicity constraints, however, the constraint language is not restricted to them. Note that the two constraints are syntactically rather similar. They operate at different abstraction levels but ConceptBase does not treat them differently. Abstractions levels are only a user interpretation of the models in ConceptBase.

ConceptBase also supports deductive rules. They are characterized by a single predicate in the conclusion and all variables in the conclusion predicate are forall-quantified, for example

```
forall w/worksFor e/Employee p/Project b/Integer
   (w toEmp e) and (w toProj p) and
   (p budget b) and (b > 0)
==> (e workIn p)
```

Queries in ConceptBase amalgamate the concept of a class and the concept of a constraint. They are defined as subclass of another class and a membership constraint specifies the condition, which instances of the superclass are instances of the query class. The variable ‘this’ stands for an instance of the superclass ‘Project’.

```
BigProject in QueryClass isA Project with
  constraint
   c1: $ forall b/Integer (this budget b) ==> (b > 1000) $
end
```

A class constraint may never be violated by a database, hence any attempt to add objects violating a constraints leads to a rejection of the update. In modeling, this behavior is generally note desired since one starts with incomplete models that may violate certain constraints. Query classes are not constraining the database but returning an answer based on the query class definition. This behavior allows re-formulating the original constraints into a negated form that returns all violators. Consider for example the constraint

```
forall this/Employee exists p/Project w/worksFor
    (w toEmp this) and (w toProj p)
```

For the query class re-formulation, we decide to return those employees who *violate* the constraint:

```
EmployeeWithNoProject in QueryClass isA Employee with
  constraint
   c1: $ not exists p/Project w/worksFor
          (w toEmp this) and (w toProj p) $
end
```

The instances of `EmployeeWithNoProject` are precisely those employees that violate the original constraint.

Attributes of objects are represented in the same way as relationships. Values like integers or strings are objects as well:

```
Employee with                bill in Employee with
 attribute                       age
   age: Integer;                   billsage: 27
   colleague: Employee           colleague
end                                 col1: mary;
                                    col2: anne
                             end
```

The above frame syntax is closely linked to the base predicates of ConceptBase. The frame for “bill” is equivalents to the predicate facts (bill in Employee), (bill age/billsage 27), (bill colleague/col1 mary), (bill colleague anne). The frame for “Employee” corresponds to the facts (Employee attribute/age Integer), (Employee attribute/colleague Employee). A number of built-in rules and constraints make sure that instantiation and specialization are done in the proper way. For example, the object “27” must be an instance of “Integer”, and “mary” and “anne” must be instances of ‘Employee’.

ConceptBase also supports active rules that can update the database if certain events (query calls, insertions, and deletions) occur. Active rules are more expressive than deductive rules. In particular, they could loop forever if not carefully programmed. Deductive rules, constraints, and queries shall always terminate. Another addition are (recursive) functions including arithmetic. Function calls can create new objects on the fly, e.g. 100+1 creates the new integer object 101. Like with active rules, functions are beyond the expressiveness of classical deductive rules. We refer to the ConceptBase user manual [10] for more details on active rules and functions. A particular case for using them is the definition of the execution semantics of process models, see end of this chapter.

Case 1: Linking STD and DFD

The first case of linking two modeling perspectives is taken from the structured analysis method [11]. It features data flow diagrams (DFD), entity relationship diagrams (ERD), state transition diagrams (STD), and others. The DFD language includes the construct of a control process. A control process is a process that receives events from other processes or the environment and reacts to them by triggering other processes. The inner behavior of a control process in a DFD is specified by an STD. Figure 3 shows the DFD and the STD modeling language as meta models and below an example DFD and its relation to an example STD.

Like with figure 2, the green links are instantiations. The upper level of the figure introduces a cross-notational link (STD attribute/specifies ControlProcess). This link is of type 1 in the classification scheme of figure 1. At the lower level, the STD ‘AccountsSTD’ is linked to the ‘ControlAccounts’ process:

```
(AccountSTD specifies/cp ControlAccounts)
```

The AccountsSTD itself is a model that is decomposed into states (here ‘Active’ and ‘InActive’) and the transitions between the states. Such a decomposition is called model explosion in MetaEdit+. So, a model construct from the DFD side is linked to a model on the STD side. A simple constraint crossing the two perspectives is that each control process must have a STD that specifies it. In the negated query class format, we return those control processes that have no STD:

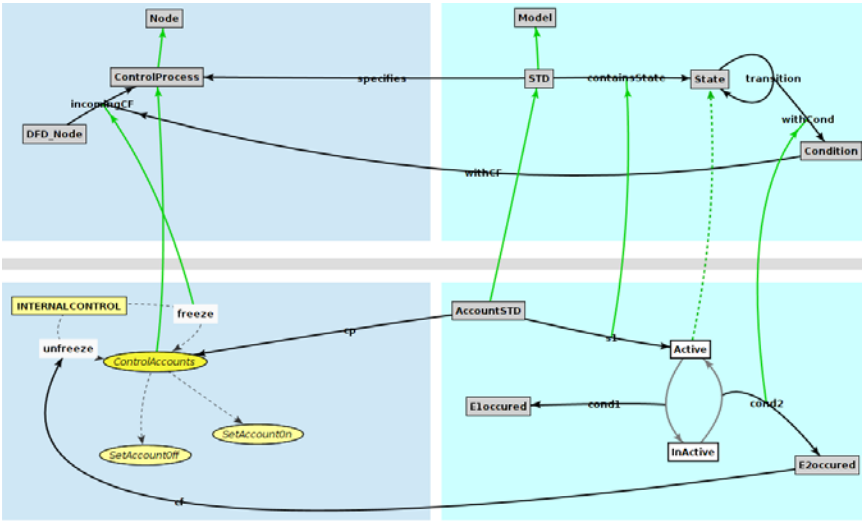

Fig. 3: Linking DFD and STD

```
UnSpecifiedControlProcess in QueryClass
                        isA ControlProcess with
  constraint
    c1 : $ not exists std/STD (std specifies this) $
end
```

A more complex constraint is linking the conditions attached to the STD transitions. They must correspond to incoming control flows on the DFD side. For example, the condition E2occured on the STD side is linked to the ‘unfreeze’ control flow on the DFD side. The following query class returns all those incoming control flows on the DFD side that are not matched with a corresponding condition on the STD side:

```
UnmatchedIncomingControlFlow in QueryClass
                              isA DFD_Node!incomingCF with
 constraint c1 : $ exists cp/ControlProcess
         To(this,cp) and
         not exists std/STD s/State
                    c/Condition t/State!transition
         (std specifies cp) and (std containsState s) and
         From(t,s) and (t withCond c) and (c withCF this) $
end
```

The query class uses two predicates that were not defined yet. The predicate *From(p,x)* returns the source object of a relation, and the predicate *To(p,y)* returns its destination object.

Case 2: Multiple perspectives in 4EM

4EM is an enterprise modeling language that strongly ties the perspectives by a variant of link type 1. The meta-model of 4EM [ref] heavily uses the specialization construct to define interface classes between the modeling perspectives. Hence the link remains in the same modeling perspective, but classes for other perspectives are integrated via specializing the interface class.

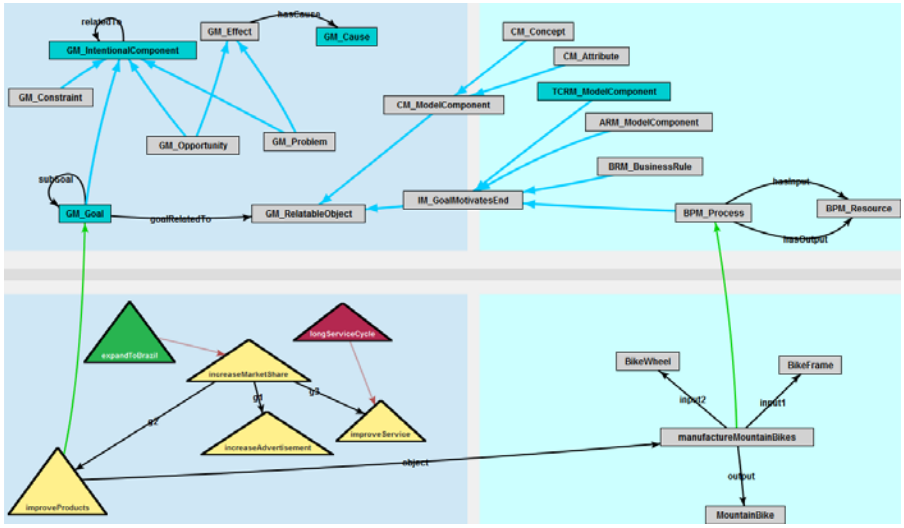

Fig. 4: Linking perspectives in 4EM by interface classes

The interface classes are GM_RelatableObject and IM_GoalModivatesEnd. They belong to the goal modeling perspective. The class GM_Goal is linked to GM_RelatableObject. The upper right side of figure 4 displays part of the 4EM meta-model for the business process perspective. There, the class BPM_Process is defined as subclass of IM_GoalMotivatesEnd. The high number of subclasses of GM_RelatableObject allows to attach goals to virtually any other 4EM object.

The lower part of figure 4 shows an excerpt of a 4EM model, instantiating the meta classes of the upper part. The process manufactureMountainBikes is related to the goal improveProducts. This link crosses the perspective boundaries between the goal model and the process model.

ConceptBase allows realizing analysis services for 4EM models via query classes. For example, we may want to know to which goals a business process related to:

```
ImpactOfGoal in QueryClass isA GM_RelatableObject with
  computed_attribute
    goalElement : GM_IntentionalComponent
  constraint
    hasImpact : $ exists g1,g2/GM_Goal
                (~goalElement relatedTo g1) and
                (g1 subGoal g2) and (g2 goalRelatedTo this) $
end
```

The attribute goalElement is declared as computed attribute. It shall be returned in the answer. The subGoal relation of GB_Goal is defined as transitive and reflexive. These properties are realized by deductive rules, not shown here but easily implemented in ConceptBase. The complete specification is available on the website http://conceptbase.cc/nemo2015.

Case 3: ArchiMate

ArchiMate is a standard meta-model and notation for enterprise architectures. It does not distinguish perspective but rather levels: the business layer, the application layer, and the infrastructure layer. The links between these levels are incarnations of our link types 2 (“implementation. ArchiMate defines two links in its meta-model that are falling in our link type category 2. The ‘realizes’ link is relating concepts that are in the same level but the realizing concept is more concrete than the realized one. The ‘uses’ link relates concepts at different levels.

The main purpose of ArchiMate is to allow traceability between concepts of a complex enterprise architecture. For example, an enterprise architect is interested to know which business services depend on a particular operating system platform like ‘DebianLinux’. ConceptBase allows to implement this traceability by a set of deductive rules for a predicate ‘dependsOn’:

```
forall o/AM_Object d/AM_DataObject
   (d realises o) ==> (o dependsOn d)
forall o/AM_Object r/AM_Representation
   (r realises o) ==> (o dependsOn r)
forall b/AM_Behaviour s/AM_BusService
   (b realises s) ==> (s dependsOn b)
…
forall a/AM_AppFunction i/ AM_InfService
   (a uses i) ) ==> (a dependsOn i)
…
```

The rules then allow following dependencies between model elements spanning multiple levels.

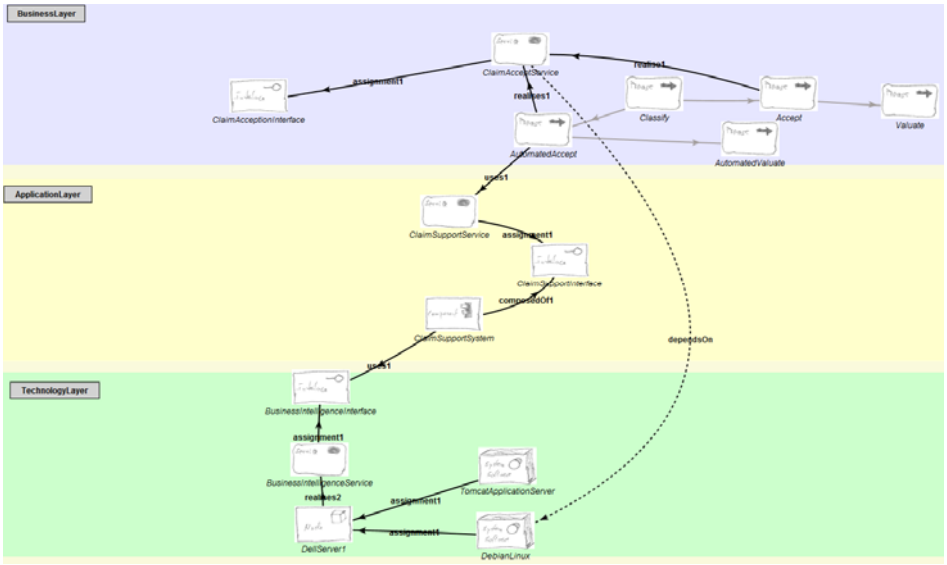

Fig. 5: Tracing dependencies for ArchiMate

There are in total more than twenty such rules for the ‘dependsOn’ relation. A generic query computes them all:

```
DepService in GenericQueryClass isA AnyNode with
  computed_attribute,parameter
    element : AnyNode
  constraint
    c1 : $ (~element dependsOn ~this) $
end
```

The query has a parameter ‘element’ that allows to focus on a specific ArchiMate element, e.g. the ClaimAcceptService from the business layer. The answer to the query lists all ArchiMate object on which this object depends on. The class AnyNode is subsuming any ArchiMate object. Hence, we follow dependencies regardless of the level in which they are defined. The ‘dependsOn’ relation is defined as transitive by the following frame:

```
AnyNode in Class with
  transitive dependsOn : AnyNode
  rule
   generated : $ forall x,y,z/AnyNode ((x dependsOn y)
     and (y dependsOn z)) ==> (x dependsOn z) $
end
```

The rule is generated by ConceptBase from a generic rule defining transitivity of any relation. ConceptBase supports a large library of such generic formulas, e.g. for symmetry, anti-symmetry, reflexivity, and multiplicity constraints.

SemCheck: integrity checking for ADOxx

ADOxx views a modeling method as combination of several modeling techniques, each coming with a modeling language (represented as a meta-model), a modeling procedure the workflow of modeling steps that leads to a desired result), and related mechanisms and algorithms (methods that operate on models). Example algorithms are for example discrete event simulation algorithms that take a process model and a configuration of parameters as input and produce performance data such as the average cycle time. The logical language of ConceptBase provides integrity checking services (called SemCheck) both on the generic level (defined for a given modeling language in the ADOxx development toolkit), and the specific level (only applicable for specific models defined in the ADOxx modelling toolkit).

The dual use in ADOxx is possible since ConceptBase uniformly represents models, meta-models, and meta^2 models with the same predicates for instantiation, specialization, and attribution. The preferred way to realize integrity constraints in ConceptBase is by means of a query class as discussed in the preceding chapters. From the viewpoint of ADOxx, a query class is a method that can be called at any time and returns the 'violators' of the integrity constraint that it implements. Most such query classes are defined for a given modeling language, e.g. entity relationship diagrams. An example is the integrity constraint 'RelationshipTypeLacksRoles' that each relationship type must have at least one role link to an entity type (compare section on constraint checking with ConceptBase):

```
RelationshipTypeLacksRoles in QueryClass
                          isA RelationshipType with
  constraint
     c: $ not exists E/EntityType (this role E) $
end
```

The response to calling this query are all relationship types that match the query class. The execution of the query class call can be linked to a specific step of the modelling procedure defined in ADOxx. Moving from one modelling state to the next then requires that all query classes defined as post-condition of the current stage return an empty answer.

An example of a model-level constraint is that each employee who works for the R&D department must work on at least one project:

```
RDEmployeeWithoutProject in QueryClass isA Employee with
  constraint
     c: $ (this department R&D) and
          not exists w/worksFor p/Project
             (w toEmp this) and (w toProj p) $
end
```

Such an integrity constraint is specific for a given ER model. The mechanism to call it is the same as for the generic constraint "RelationshipTypeLacksRoles". Note that the above constraint requires as sample data level to be evaluated.

Query classes are subclasses of other classes. The query class 'RelationshipTypeLacksRoles' is a subclass of "RelationshipType" (being part of a meta-model), the class 'RDEmployeeWithoutProject' is a subclass of 'Employee', which is part of a conceptual model expressed in terms of a meta-model. All such query classes of the same superclass form the set of constraints that the superclass must eventually fulfill. Asking the query classes returns the violators, i.e. those instances of the super classes that match the condition of the query class. The use of query classes has the advantage that one can ask them when appropriate. In early modeling stages, the conceptual models are incomplete and possibly violate many conditions expressed in the query classes. One can count

the number of instances in the query classes to realize a metric on the degree of inconsistency of a given model, e.g. `COUNT(RelationshipTypeLacksRoles)`. If a class has multiple query classes defined for it, then one can aggregate them into a single query class:

```
FaultyRT in QueryClass isA RelationshipType with
   constraint
       c: $ (this in RelationshipTypeLacksRoles) or
            (this in RelationshipTypeXXX) or …$
end
```

SemCheck is also used for checking the **consistency of a meta-model** (e.g. defining the ER language) against meta^2 models. Consider figure 6 (ER meta-model assimilated in FCML) in the chapter "Fundamental Conceptual Modelling Languages in OMILab" (FCML). The meta^2 model consists of the concepts 'Class' and 'Relationship'. The latter has two role links 'Class' (labelled 'source' and 'target'). The concept 'Class' has a self-referential link 'inheritance' that is used to specify specialization hierarchies. The semantics of the 'inheritance' link can be specified in ConceptBase by the following definitions:

```
CLASS with
  attribute,single,transitive,reflexive,antisymmetric
    inheritance: CLASS
end

InheritanceRule in Class with
  rule r1: $ forall C,D/CLASS x/Proposition
             (C inheritance D) and (x in C) ==> (x in D) $
end
```

The first frame uses a combination of attribute categories 'single', 'transitive', 'reflexive', and 'antisymmetric, which can be imported from a formula repository in ConceptBase. For instance, the definition of 'antisymmetric' is:

```
(x M y) and (y M x)  ==> (x = y)
```

which translates to

```
(x inheritance y) and (y inheritance x)  ==> (x = y)
```

for the 'inheritance link. The other attribute categories are defined in an analogous way. The meta-model instantiated from the meta^2 model uses the 'inheritance link as shown here (see also figure 6 in the FCML chapter).

```
ENTITY in CLASS with
  inheritance
     super: EoR
end

RELATION in CLASS with
  inheritance
      super: EoR
end
```

The 'inheritance rule' of the meta^2 model ensures that any instance of 'Entity' is also an instance of 'EoR', the abstract super-class of 'Entity' and 'Relation'. For example, the concept 'Book' is a direct instance of 'Entity' and via generic inheritance rule also an instance of 'EoR'.

The formal specification of rules and constraints at the meta^2 model assists the method engineer in designing compliant modeling languages. Attempts to create meta-models that violate the constraints result in appropriate

error messages. For example, a circular specialization hierarchy is detected by the transitivity and anti-symmetry rules of the meta^2 model.

Integration architecture for ADOxx and ConceptBase

The three cases discussed above motivate the suitability of ConceptBase as a tool to check constraint and to provide deduction-based analysis services for enterprise modeling frameworks that cover multiple perspectives and levels. ADOxx is such a framework. This chapter discusses how to integrate ADOxx and ConceptBase.

ADOxx offers two toolkits. The development toolkit is used to define a modeling language by means of meta-models. It also assigns a graphical notation of node and link shapes to the elements of the meta-models. Further, the designer of the modeling language can associate semantics to the modeling language by linking them to algorithms. For example, a process modeling language is associated to a mapping to simulation models that utilize specific algorithms. The second toolkit is called the modeling toolkit. The development toolkit generates the modeling toolkit for the given modeling language. Hence, this is the environment that is used by an enterprise modeler.

ConceptBase does formally not distinguish between the constructs of a modeling language and the constructs of a model. They are all represented in the very same data structure. Still it makes sense to distinguish the two types of concepts since ADOxx distinguishes them. To do so, we propose to use the module system of ConceptBase. Modules in ConceptBase are simply sets of objects. Modules can have sub-modules, in which all objects of the super-modules are visible.

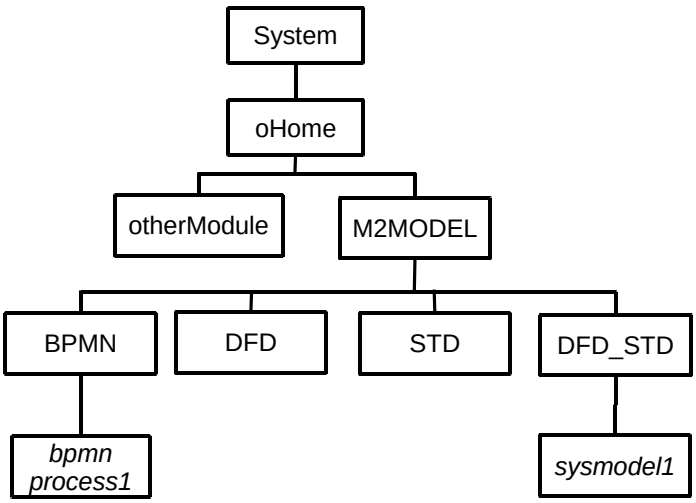

Fig. 6: ConceptBase module structure for ADOxx integration

Figure 6 shows the module structure of ConceptBase adapted to the requirements for ADOxx integration. Each sub-module 'sees' the definitions made in its super-module hierarchy, i.e. the modules on the path from the sub-module to the top module. The top module 'System' includes the pre-defined objects of ConceptBase. Below is the module 'oHome' which hosts the home modules, one of them being 'M2MODEL'. The 'M2MODEL' module contains definitions of meta-classes that make ConceptBase compatible with ADOxx. It also contains a set of generic rules and constraints such as for transitivity and for multiplicity constraints. These constraints can then be re-used for all sub-modules of 'M2MODEL'. The sub-modules like 'BPMN' contain the meta-models of the ADOxx modeling language to be supported by the SemCheck service of ConceptBase. The definitions are passed from the ADOxx Development Toolkit to the suitable sub-module whenever a new modeling language is defined. The translation of the ADOxx meta-model to the ConceptBase frame syntax has to be performed by an adapter. Some sub-modules like 'DFD_STD' combine several modeling languages, here DFD and STD. This is achieved by storing the meta-models of both languages in this sub-module. The model DFD_STD shall also include the query classes to check the semantic integrity and to analyze the models, e.g. on dependencies between model elements.

The sub-module ‘sysmodel1’ is an example for a module storing models in the combined DFD_STD language. It uses the definitions of DFD_STD to represent the models. The ADOxx Modelling Toolkit shall pass the model definitions to the appropriate sub-module via an adapter. It can then retrieve reports on the semantic integrity via the query classes defined in the super-module.

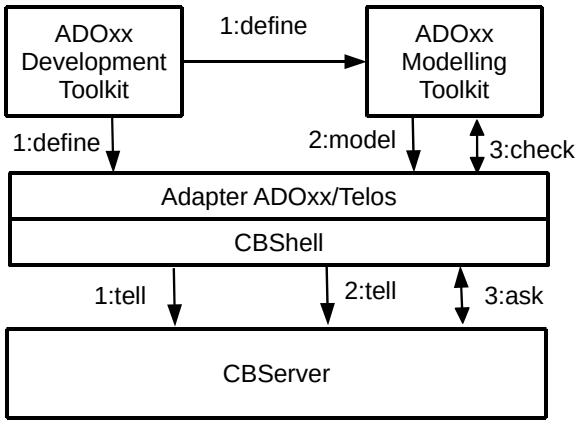

Fig. 7: Integration architecture for ADOxx and ConceptBase

Figure 7 identifies the components that are needed for the ADOxx/ConceptBase integration. The components ADOxx Development Toolkit, ADOxx Modelling Toolkit, CBShell and CBServer are readily existing. The CBShell component is a command interpreter for the ConceptBase server CBServer. CBShell is Java program that accepts commands from a terminal and then calls the CBServer to execute the command. It can be easily adapted to let it be called from another program, here the Adapter ADOxx/Telos. This adapter receives meta-models and models from the ADOxx components and translates them into the ConceptBase frame syntax. This adapter needs to be implemented to get the integration working.

The workflow is starting from the development toolkit. Assume a designer creates a meta-model for DFD_STD. The meta-model is stores in ADOxx’s repository and in parallel the definition is passed to the adapter. The adapter transforms the ADOxx representation into CBShell commands that store the meta-model in the suitable ConceptBase module, here DFD_STD. The next step is that a modeler uses the modelling toolkit to create an example model. This model is stored in the ADOxx repository and then passed it to the CBServer via the adapter and CBShell. The modelling toolkit can then request the consistency checks by calling the query classes implementing them. The answer is a list of ‘violators’, which can then be highlighted in the modelling toolkit. Depending on the modelling phase, the toolkit could request different consistency checks. For example, the query ‘UnmatchedIncomingControlFlow’ of the DFD/STD case could be part of the final consistency checks when both the DFD and the STD are regarded as complete.

The commands of CBShell define the interface between the adapter and the CBServer. The following commands are the most relevant ones for the integration:

startServer *serveroptions*
: start a new CBServer on localhost. The server options allow among others to specify the database to be used and to specify the port number for TCP/IP connections

connect *host port*
: connect to an already running CBServer on host:port

disconnect
: disconnect from the current CBServer

setModule *modulePath*
: set the new module, e.g “setModule oHome/ADOxx/BPMN”

newModule *modulename*

create a new sub-module in the current module

tell *frames*

store the specified frames (given as text string) to the current module

ask *query options*

ask the specified query given by the name of the query class, possibly including parameters in the query call; the options can be used to specify the answer representation

showAnswer

displays the answer to the query called before

The CBServer stores by default all objects persistently. It can however also be configured to only store them in main memory. Below is a trace of CBShell using the above commands for the DFD_STD example.

```
1. connect localhost 4001
2. setModule ADOxx/DFD_STD/sysmodel1
3. tell "INTERNALCONTROL with
     incomingCF
      reset: ControlAccounts
    end"
4. ask UnmatchedIncomingControlFlow
5. showAnswer
   INTERNALCONTROL!reset
```

It is assumed here that the modules ADOxx and DFD_STD have already been defined and that the example model of figure 3 has been stored in ‘sysmodel1’. The tell command incrementally adds a new model element to the existing model, here a ‘reset’ link as incoming control flow of the control process ‘ControlAccounts’. The query ‘UnmatchedIncomingControlFlow’ then exposes this new link as being not matched with the STD specifying the control process.

Note that the ask command is typically called after a meaningful sequence of modeling steps in ADOxx have been executed. The answer ‘INTERNALCONTROL!reset’ identifies the ‘reset’ link of the object ‘INTERNALCONTROL’ as the violator. The query name ‘UnmatchedIncomingControlFlow’ tells ADOxx how to interpret the answer. In this case, ADOxx may present to the modeler that he has link the control flow to some condition in the STD.

Instead of incremental changes, ADOxx can also pass the whole model to ConceptBase. ConceptBase will automatically extract only the new objects and then tell only them to the selected module.

There are a number of improvements that the CBServer could offer to support the integration. The most significant one would be to support merging two existing modules. For example, DFD_STD could be defined as submodule of both DFD and STD and then would see the definitions of both. Currently, one has to duplicate the content of DFD and STD into DFD_STD. A second improvement would be to remove a module, i.e. to delete its content. ConceptBase can handle models with several hundred thousand objects. The query performance for the examples discussed in this chapter are in the range of milliseconds.

Inheriting execution semantics for process models

The ConceptBase module structure discussed in the previous section allows to share meta-models and to separate different modelling environments from each other. In this section, we discuss the uses of so-called active rules and deduction rules to specify the execution semantics for petri nets and then to share this semantics as well.

A classical Petri net consists of places and transitions. Places have a marking being a non-negative integer number. There are flow links between places and transitions. A place is an *input place* for a transition if there is a flow link from the place to the transition, and an output place if there is a flow link from the transition to the place. A transition is enabled if all its input places have a marking greater than zero. In ConceptBase, this can be modeled as follows:

```
GProcessElement with
  attribute flowTo : GProcessElement
end
GPlace isA GProcessElement with
  attribute marks : Integer
end
GTransition isA GProcessElement end


M in Function isA Integer with
  parameter p : GPlace
  constraint c1 : $ (p marks this) $
end
Input in GenericQueryClass isA GPlace with
  parameter t : GTransition
  constraint
    ci : $ (this flowTo t) $
end
ConnectedPlace in GenericQueryClass isA GPlace with
  Parameter trans : GTransition
  constraint
    c : $ (this flowTo trans) or (trans flowTo this) $
end
IM in Function isA Integer with
  parameter p : GPlace; t : GTransition
  constraint
    c1 : $  (t flowTo p) and not (p flowTo t) and (this = 1) or
             (p flowTo t) and not (t flowTo p) and (this = -1) or
            not (p flowTo t) and not (t flowTo p) and (this = 0) $
end
Enabled in QueryClass isA GTransition with
  constraint c : $ forall p/Input[this] (M(p) > 0) $
end
```

The query class ‘Enabled’ returns the currently enabled transitions. The function M returns the marking of a given place and the function IM realizes the incidence matrix between places and transitions. The firing of a transition can be expressed by an active rule:

```
gfire in GenericQueryClass isA YesClass with
  parameter transition : Enabled
end
UpdateConnectedPlaces in ECArule with
  mode m : Deferred
  ecarule
    er : $ t/GTransition p/GPlace m/Integer
           ON Ask gfire[t]
           IFNEW (t in Enabled) and (p in ConnectedPlace[t]) and
           (m = M(p)+IM(p,t))
           DO Retell (p marks m) $
end
```

The active rule is triggered by the command `gfire[t]` for an enabled transition. It will then change he markings of the connected places according to the old state and the incidence matrix IM.

Since the above frames are completely defining the semantics of classical petri nets, we can re-use the definition to define semantics to other process modelling languages such as BPMN, state transition diagrams, event-process chains and others just by mapping their constructs into the petri net constructs for places and transitions.

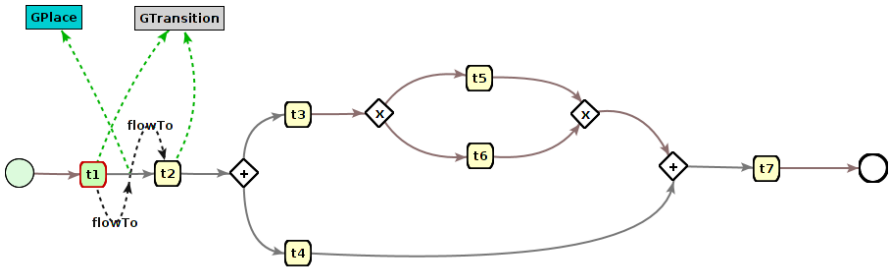

Fig. 8: A BPMN model instantiated to Petri net constructs

Figure 8 shows how the result of the mapping on a sample BPMN model. The figure is an actual screendump of ConceptBase, hence all displayed elements are actually taken from the model definitions stored in ConceptBase. The BPMN tasks are instantiated to GTransition, hence they operate like Petri net transitions. The start and end events are mapped to GPlace. The connection between two transitions like t1 and t2 is interpreted as a place, and there are corresponding derived flow links from t1 to the link and from the link to t2. The following ConceptBase frames are achieving the mapping:

```
TransitionLike isA BPMN_Element,GTransition end
PlaceLike isA BPMN_Element,GPlace end
BPMN_Activity isA TransitionLike end
BPMN_Event isA PlaceLike  end
MapBPMNToGPM in Class with
  rule
    r1 : $ forall a1,a2/TransitionLike link/BPMN_Element!next
           From(link,a1) and To(link,a2) ==> (link in GPlace) $;
    r2 : $ forall a1,a2/TransitionLike link/BPMN_Element!next
           From(link,a1) and To(link,a2) ==> (a1 flowTo link) $;
    r3 : $ forall a1,a2/TransitionLike link/GPlace
              (link in BPMN_Element!next) and
           From(link,a1) and To(link,a2) ==> (link flowTo a2) $
 end
```

This definition allows to directly executing a BPMN process model using the ‘gfire’ command. The complete definition is available via http://conceptbase.cc/nemo2015. Even if the execution semantics is not needed in the integration with ADOxx (since it has more advanced algorithms to specify execution semantics), the query to check enabled tasks on a given state is useful to designers of new process modeling languages.

Conclusions

This chapter motivated that enterprise models consist of multiple modelling perspectives and that these perspectives need be synchronized by semantic constraints. We presented the capabilities of the constraint and query language of the ConceptBase system and showed in three cases that it can represent and evaluate typical constraints.

We also presented an integration architecture where ConceptBase is used as a backend for the ADOxx enterprise modeling platform. Since ConceptBase allows the representation of models at any abstraction level, its service can support both the ADOxx Development Toolkit (meta-modelling) and the ADOxx Modelling Toolkit (modelling).

Finally, we presented an approach to re-use execution and analysis functions for process modelling languages. The constructs are defined for Petri nets and then can be re-used for BPMN and other process modelling languages.

All definitions used in this chapter are also available online. Future work has to be done for actually integrating ADOxx and ConceptBase. One element is the adapter that converts the ADOxx models and meta models into the Telos syntax used by ConceptBase. Since both are based on graphical representations, this should be a rather straightforward step. Secondly, the semantic constraints have to be declared within ADOxx and then passed to

ConceptBase. Finally, the error reports returned by ConceptBase need to be displayed in a suitable way by ADOxx.

There are some services beyond constraint checking that could be outsourced to ConceptBase. One service is the dependency tracking in large enterprise models, see the case study on ADOxx. ConceptBase has a fast Datalog engine to evaluate recursive rules, in particular for following transitive links. A second service are model metrics. The recursive functions in ConceptBase allow the definition of metrics such as for model complexity.

Acknowledgements. The ConceptBase models for ArchiMate were created in 2008 by Sander van Arendonk, Niels Colijn, Dirk Janssen, and Jeffrey Kramer as part of an assignment for the method engineering course at Tilburg University. Their models are available under a Creative Commons CC-BY-NC 3.0 license. Special thanks to Wilfrid Utz and David Götzinger from the University of Vienna for their help in getting the coupling between ConceptBase and ADOxx working.

References

1. D. Karagiannis, W. Grossmann, P. Höfferer (2008): Open Model Initiative – A Feasibility Study. Project Study on behalf of the Austrian Federal Ministry for Transport, Innovation and Technology. Vienna, September 2008.
2. D. Karagiannis, H. Kühn (2002): Metamodelling Platforms. Proceedings 3rd. International Conference EC-Web 2002/Dexa 2002, http://dx.doi.org/10.1007/3-540-45705-4_19.
3. H.-G. Fill, D. Karagiannis (2013): On the Conceptualisation of Modelling Methods Using the ADOxx Meta Modelling Platform. *Enterprise Modelling and Information Systems Architectures* 8(1):4-25.
4. M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer (1995): ConceptBase – A Deductive Object Base for Meta Data Management. J. Intell. Inf. Syst 4(2):167-192, http://dx.doi.org/10.1007/BF00961873.
5. M.A. Jeusfeld (2009): Metamodeling and method engineering with ConceptBase. In M.A. Jeusfeld, M. Jarke, J. Mylopoulos (eds.): Metamodeling for Method Engineering, The MIT Press, Cambridge, MA, USA, ISBN 978-0262101080, 89-168.
6. K. Sandkuhl, J. Stirna, A. Persson, M. Wißotzki (2014): Enterprise Modeling - Tackling Business Challenges with the 4EM Method. The Enterprise Engineering Series, Springer 2014, ISBN 978-3-662-43724-7.
7. M. Lankhorst et al. (2013): Enterprise Architecture at Work. Third Edition. Springer, Berlin, ISBN 978-3-642-29650-5.
8. J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis (1990): Telos - Representing Knowledge About Information Systems. *ACM Trans. Inf. Syst.* 8(4):325-362, http://doi.acm.org/10.1145/102675.102676.
9. S. Ceri, G. Gottlob, L. Tanca (1989): What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Eng.* 1(1): 146-166. https://dx.doi.org/10.1109/69.43410.
10. M.A. Jeusfeld, C. Quix, M. Jarke (2015): ConceptBase.cc User Manual. Version 7.8, Online http://conceptbase.sourceforge.net/userManual78/.
11. E. Yourdan (1989): Modern Structured Analysis. Prentice Hal, Englewood Cliffs, NJ, USA.