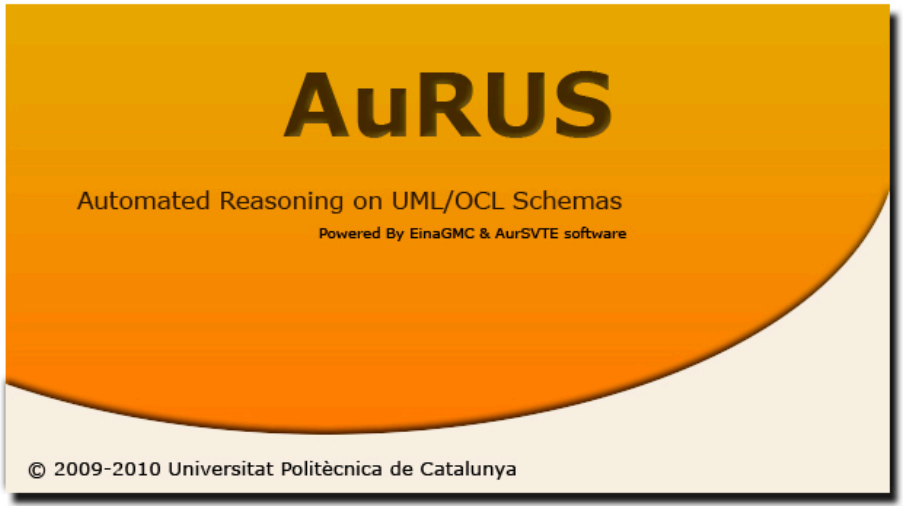


AuRUS USER MANUAL

A glance beneath the reasoning assistant

Barcelona School of Informatics

Revision History

Version	Date	Revision Description
1.0	12/02/2011	User Manual Writing
1.1	05/03/2011	Minor Revision
1.2	07/03/2011	Minor Revision
1.3	03/05/2011	General Formatting

Table of Contents

Introduction

AuRUS is an Automatic reasoning tool for UML/OCL schemas. It can verify the conceptual diagram (that is, ensuring the absence of redundancies or contradictions) by performing some automatic tests like class liveliness tests or OCL Constraints redundancy tests. It also helps to validate the schema (that is, ensuring that its representation corresponds to the real world) by performing some automatic tests that must be interpreted by the specifier like class identifiers detection, inclusion/exclusion paths tests, etc. Furthermore, this tool allows you to define and execute your own state reachability tests.

Purpose

This document has been written to help to get introduced in using this tool. Concretely, this User Manual explains how to load a conceptual schema to the tool, how to make tests and get the results.

Related Documents

This document explains everything you need about using AuRUS. However, if you want to know how it works, you can consult the following paper:

AuRUS: Automated Reasoning on UML/OCL Schemas; *Anna Queralt, GuillemRull, Ernest Teniente, CarlesFarré, Toni Urpí*

Conventions

Throughout this document, we will use some concrete vocabulary which will be written in italic. We will define each concrete word only the first time it appears, but you will be able to find its definition in the Glossary of the end of this manual.

Problem Reporting Instructions

If you find any error in AuRUS, you can report it to the following mail address: teniente<at>essi.upc.edu

Overview

Decidability

It is well known that reasoning on conceptual schemas is not decidable, that is, some tests may not end for some specific diagrams.

However, AuRUS can check some properties on the schema which ensures the termination of any test. This checking is known as the *decidability test*. Notice that, if this checking fails, it is not possible to predict if there is or not any test that would never end.

Completeness

AuRUS was thought to ensure the completeness of any test. That is giving an instance which satisfies the test goal, or answering that it is not possible to find one in case it does not exists.

Nevertheless, it may happen that the unique instances which satisfy the goal are not finite. Then, AuRUS will never finish the instance computation, which is the cause of the never ending tests. Notice that this situation will never occur if the schema is correct or the decidability test has been passed.

Expressiveness

AuRUS deals with the most common graphic UML elements: Classes, n-ary Associations, Association Classes, Hierarchies and Association Cardinalities. However, at the moment it is not possible to define Attribute cardinalities and all the types are interpreted as Real numbers.

One important point about AuRUS is the expressiveness of the allowed OCL Constraints. In the current version the following type of OCL Constraints are admitted (notice that aggregation functions like sum() are not admitted):

Admitted Boolean Operators:
And, or, not, includes(), excludes(), includesAll(), excludesAll(), isEmpty(), notEmpty(), exists(), forall(), isUnique(), oclIsTypeOf(), <, >, <=, >=, <>, =

Admitted Loop Expressions:
Select(), allInstances()

Admitted Collection Operators:
size(), asSet()

Other admitted expressions:
oclAsType()

There are only two restrictions that might be considered when writing the OCL constraints: it is not possible to compare the size of a collection with a variable (only constants) and it is not possible to work with bags (only sets, which means that the asSet() operation should be used if the navigation path has more than one navigation).

Input Creation

The source input normally used in AuRUS is a plain ArgoUML .xmi export file[1]. The current version of ArgoUML (0.32) is based directly on the UML 1.4 specification. Furthermore, it provides an extensive support for OCL (Object Constraint Language) and XMI (XML Model Interchange format). ArgoUML has many features that make it special, but it does not implement all the features that CASE tools provide.
ArgoUML is available for free and can be used in commercial settings. For terms of use, see the license agreement presented when ArgoUML is downloaded.

AuRUS requirements over ArgoUML design

In the current version, there are several limitations that prevent ArgoUML from creating certain types of expressions accepted by AuRUS. Several workarounds have been made in order to solve this issue.
Generally speaking, AuRUS will accept any correct and legal input generated by ArgoUML. However there are some requisites to take into account:

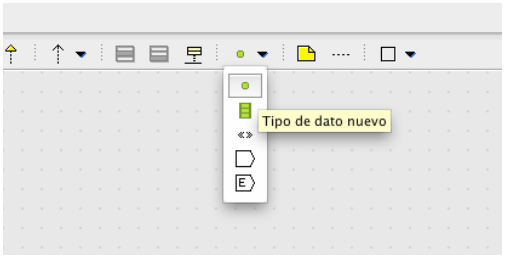
- All attribute types must be custom data types.
- All association roles must be defined in a non-ambiguous way.
- Recursive associations and similar bondings are allowed but must be specified specially.
- OCL constraints must be specified in a note format.
- Generalization restrictions are specified linking a note.

[1]Please, refer tohttp://argouml.tigris.org/for further information about ArgoUML.

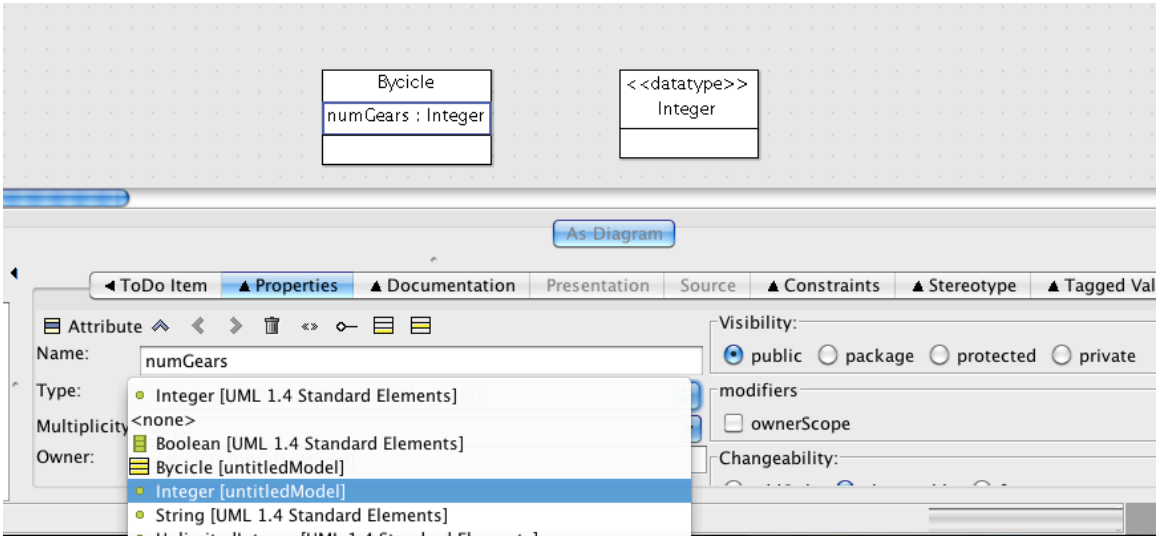
Custom made types for attributes

Attribute types are defined using data types. In order to ensure compatibility with the AuRUS reasoning engine, all attribute types included in the conceptual schema must be custom made.

Firstly, a new data type must be defined.



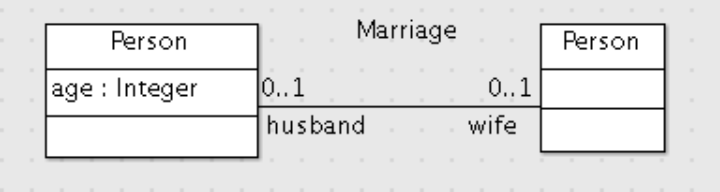
Finally, the custom type is assigned to the selected attribute.



The same data type may be used as many times as needed.

Recursive association specification

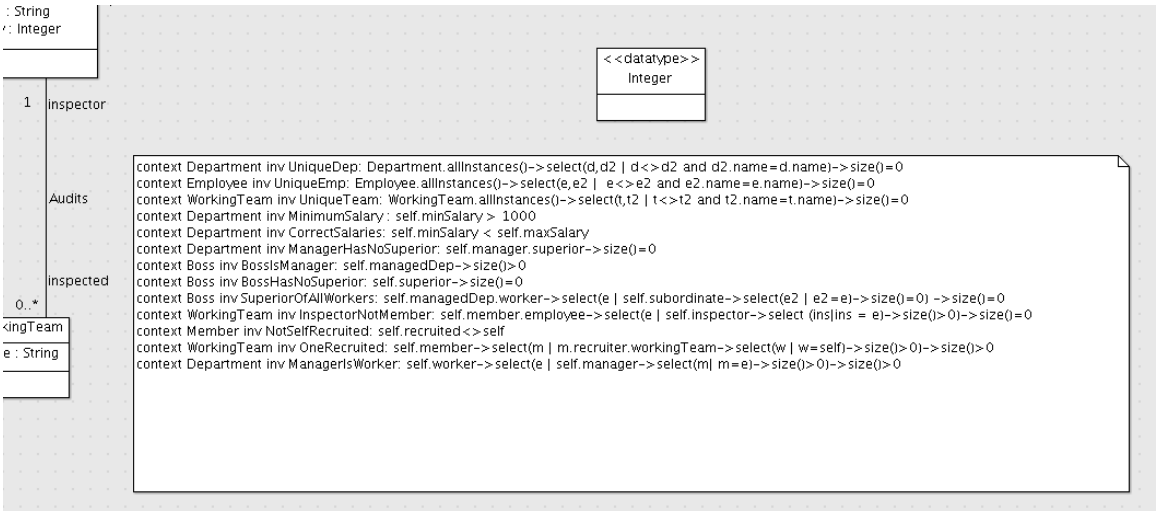
Associations with association ends bonding the same class are not allowed in ArgoUML. However, this restriction is worked around bonding two classes with the same name. Class atributes or other associations should not be duplicated, as they must be only specified once. The following figure shows a well-constructed recursive association.



This ArgoUML design issue also comes up when trying to link other type of classes such as associative classes or generalizations. In any case it shall be solved by the same procedure.

OCL constraint definition

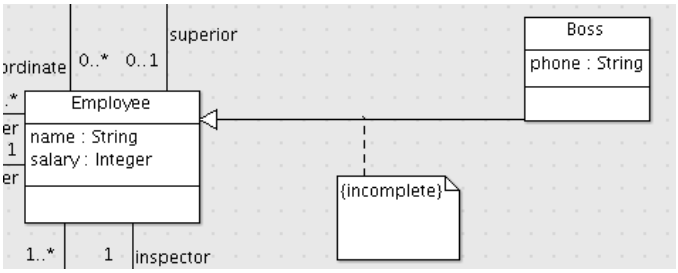
OCL constraints should be specified in a single note, as it is shown in the following figure:



Please note that it is important not to leave a new end of line after the last constraint, as the AuRUS OCL parser will interpret it as a new blank invalid restriction.

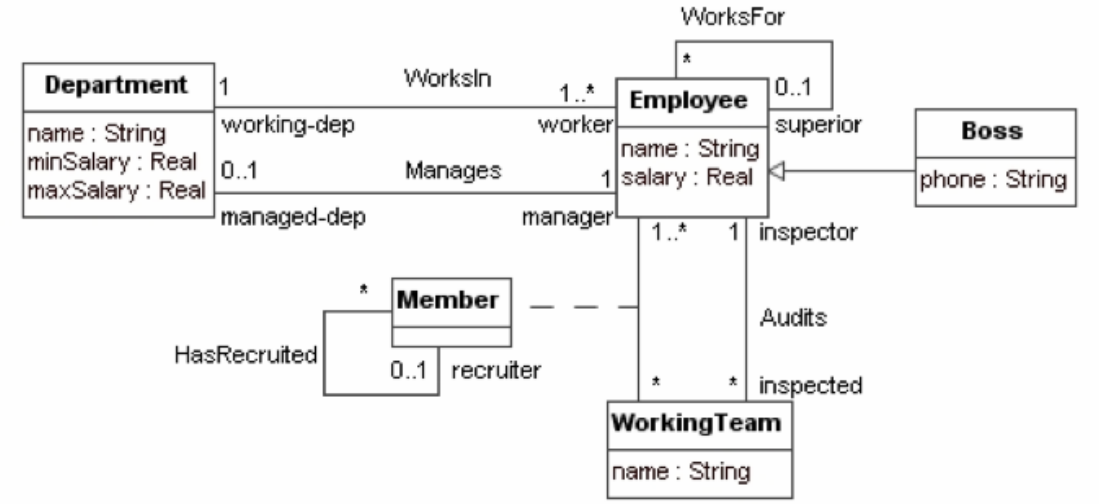
Generalization restrictions

Generalization restrictions should be specified using a linking note and one of the following legal combinations:
{incomplete, overlapping}
{incomplete, disjoint}
{complete, overlapping}
{complete, disjoint}
However, it is also allowed to specify only one of the requirements, either {incomplete}, {complete}, {disjoint} or {overlapping}.



Designing an example

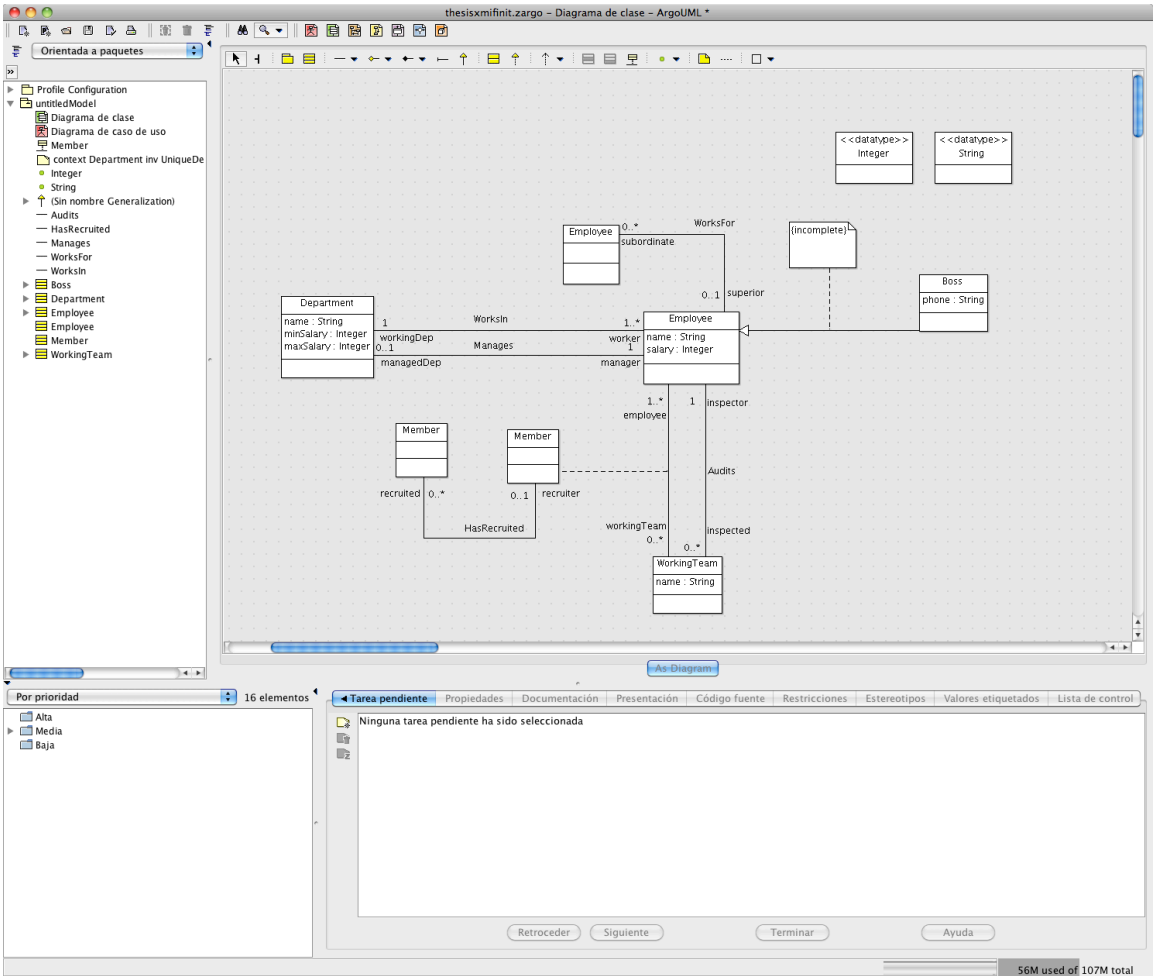
With all the previous limitations in mind, this section will illustrate how to design a sample valid input for AuRUS. The next conceptual schema will be used as an example:



The OCL restrictions associated are the following:

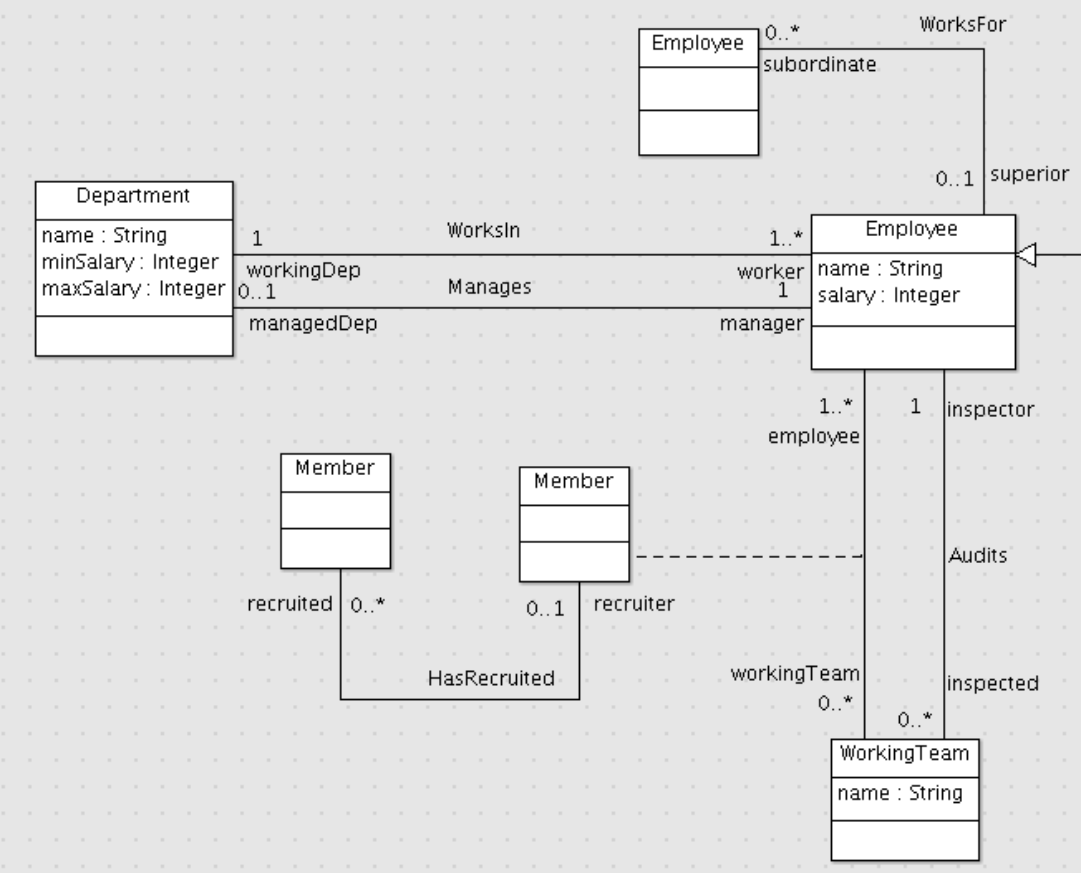
context Department invUniqueDep: Department.allInstances()->select(d,d2 | d<>d2 and d2.name=d.name)->asSet()->size()=0
context Employee invUniqueEmp: Employee.allInstances()->select(e,e2 | e<>e2 and e2.name=e.name)->asSet()->size()=0
contextWorkingTeaminvUniqueTeam: WorkingTeam.allInstances()->select(t,t2 | t<>t2 and t2.name=t.name)->asSet()->size()=0
context Department invMinimumSalary : self.minSalary> 1000
context Department invCorrectSalaries: self.minSalary<self.maxSalary
context Department invManagerHasNoSuperior: self.manager.superior->asSet()->size()=0
context Boss invBossIsManager: self.managedDep->size()>0
context Boss invBossHasNoSuperior: self.superior->size()=0
context Boss invSuperiorOfAllWorkers: self.managedDep.worker->select(e | self.subordinate->select(e2 | e2=e)->size()=0) ->asSet()->size()=0
contextWorkingTeaminvInspectorNotMember: self.member.employee->select(e | self.inspector->select (ins|ins = e)->size()>0)->asSet()->size()=0
context Member invNotSelfRecruited: self.recruited<>self
contextWorkingTeaminvOneRecruited: self.member->select(m | m.recruiter.workingTeam->select(w | w=self)->asSet()->size()>0)->size()>0
context Department invManagerIsWorker: self.worker->select(e | self.manager->select(m| m=e)->size()>0)->size()>0

The first action that should be done should be designing the basis of the schema, including all the role names and cardinalities.

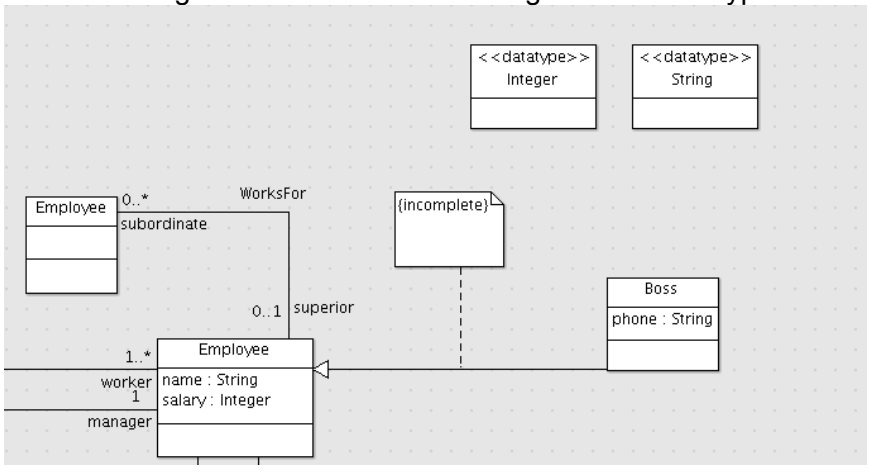


It is important to remember to apply the workarounds shown in previous sections including:

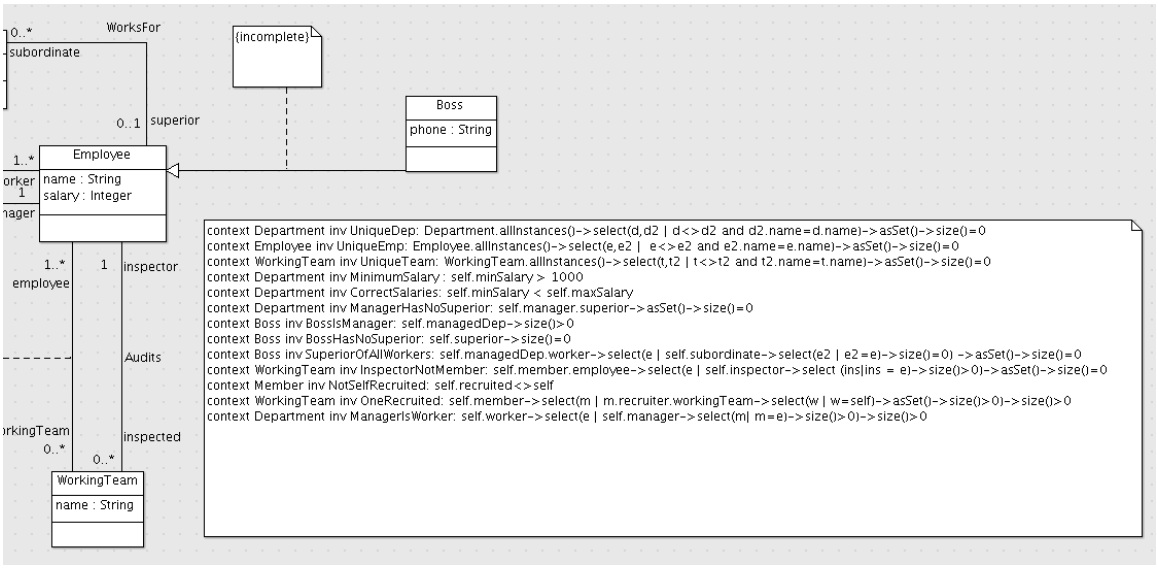
Doubling the classes when designing associations that are not allowed.



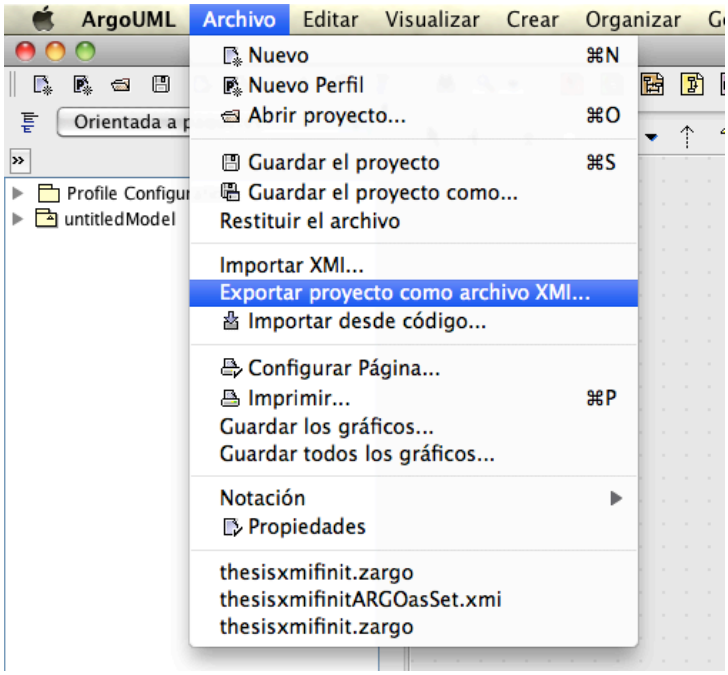
Applying comments for generalizations and creating custom data types.



After designing the core conceptual schema, the OCL constraints must be edited in a note.



Finally, it should be exported as an .xmi extension file.



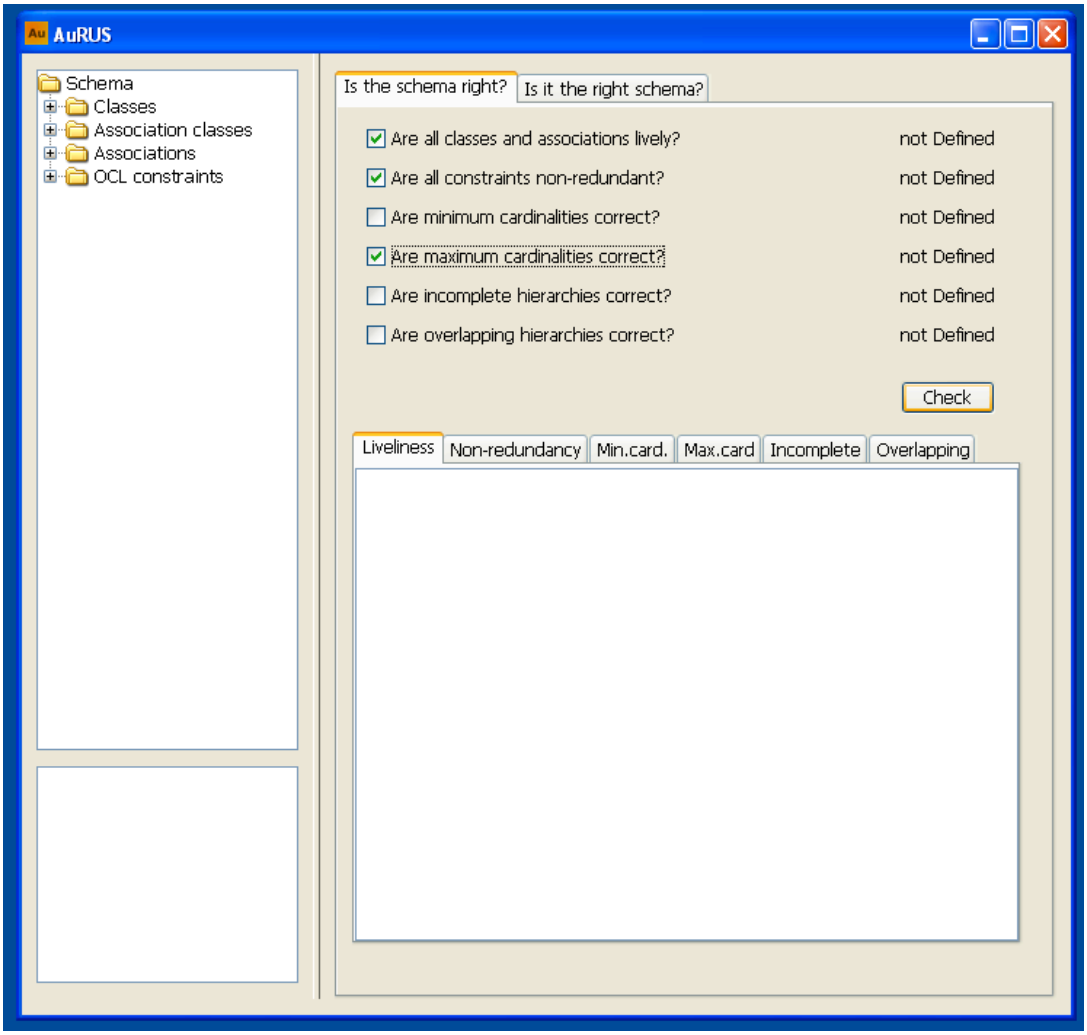
AuRUS Schema Testing

AuRUS verification tests

The AuRUS verification tool includes tests devoted to check the internal correctness of a schema. This assessments can be performed automatically without the designer intervention aiming to answer to the question *is the structural schema right?*. Some of them correspond to well known reasoning tasks (such as schema satisfiability), while others correspond to additional properties that can be automatically drawn from the definition of each conceptual schema.
For all of them, the fact that the test is not satisfiable necessarily indicates that the schema has some kind of flaw. The automatic verification tests offered are:

- Satisfiability of a Schema: A schema is satisfiable if there is a non-empty state of the IB in which all its integrity constraints are satisfied.
- Redundancy of an Integrity Constraint: An integrity constraint is redundant if integrity does not depend on it. This situation takes place if the states of the IB that are not allowed are already prevented by the rest of constraints.
- Maximum cardinality test: This assessment shows whether it is possible to have as many instances as specified by the maximum cardinality *max*associated to the same instances as the other association ends.
- Minimum cardinality test: When there is a cardinality constraint with a lower bound = 0, this check shows whether it is possible that an instance(s) at the other end(s) does not participate in the association.
- Incomplete hierarchy test: If some hierarchy is specified as {incomplete}, it can be proven if it is really such, that is, if an instance may only belong to the superclass. Otherwise, it should be specified as {complete}, or some other constraint should be removed or modified.
- Overlapping hierarchy: If some hierarchy is specified as {overlapping}, it can be checked if it is really such, that is, if an instance may belong to more than one subclass simultaneously. Otherwise, it should be specified as {disjoint}, or some other constraint should be removed or modified.

To perform verification tests check the desired properties under the *Is the schema right?*tab and press the *Check* button:



AuRUS validation tests

The AuRUS validation tool also offers a set of tests to check if the schema correctly specifies the requirements set by the designer, which are not usually formalized. That is why it is desirable to help the designer to analyze the schema so that he can decide whether it represents the intended domain.

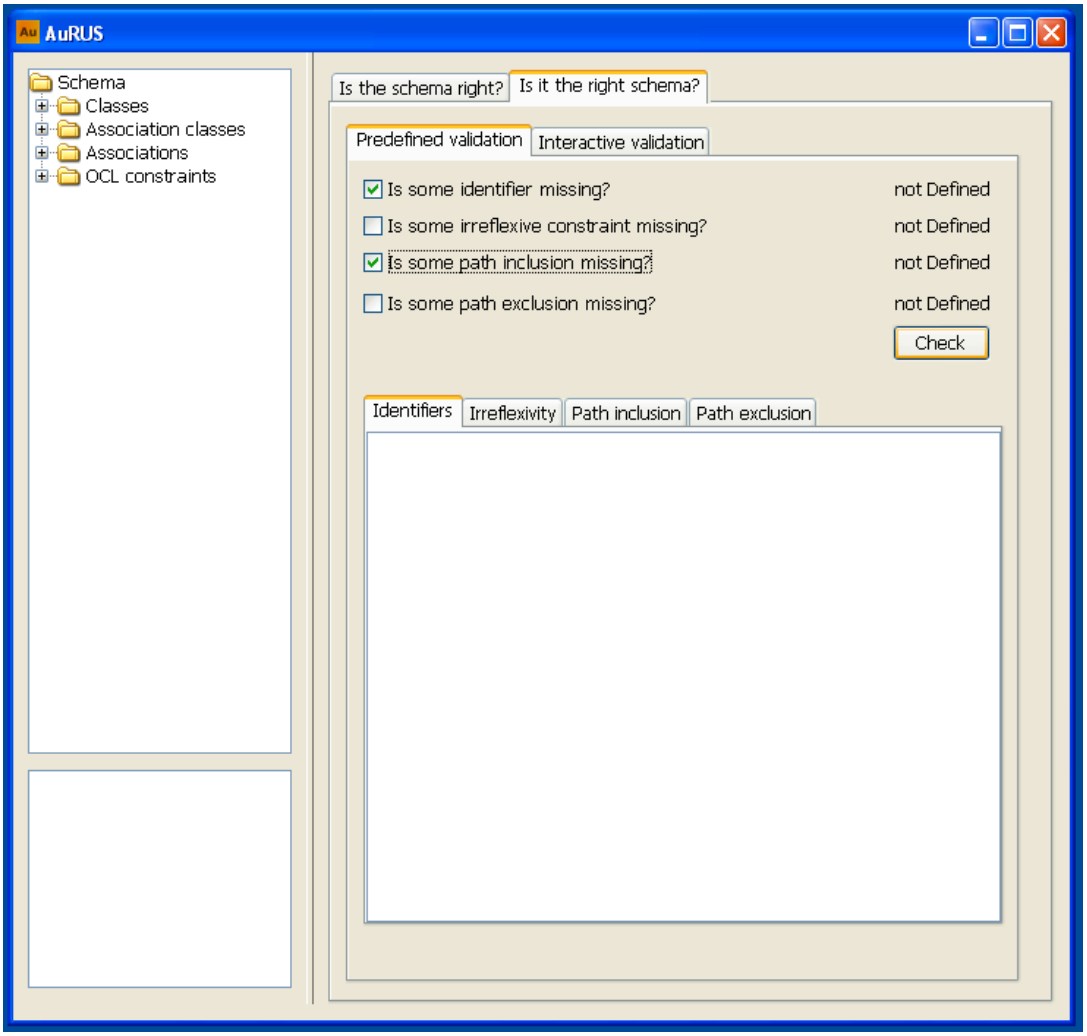
Some standard automatic tests are offered in order to check potentially undesirable common situations:

- Irreflexive constraint test: If a recursive association can relate the same instance, then an irreflexible constraint may be missing. This test can be defined for each recursive association in the schema.
- Identifier constraint test: If a class admits several valid instances with the same value in all its attributes, then an identifier constraint may be needed to state the set of attributes that uniquely identify each instance. This test can be generated

for each class of the schema, as long as it is not an association class or a subclass, since in these cases their identifiers are already known.

- Path inclusion test: If c is an instance of C, the set of instances of D related to c by means of R is a subset of the set of instances of D related to c by means of S, or the other way around. This test helps determine if there are some associations that are subset of another.
- Path exclusion test: If c is an instance of C, the set of instances of D related to c by means of R excludes the set of instances of D related to c by means of S. This test helps determine if there are some associations that imply a mutual exclusion between them.

To perform verification tests check the desired properties under the *Is it the right schema?* tab and press the *Check* button:

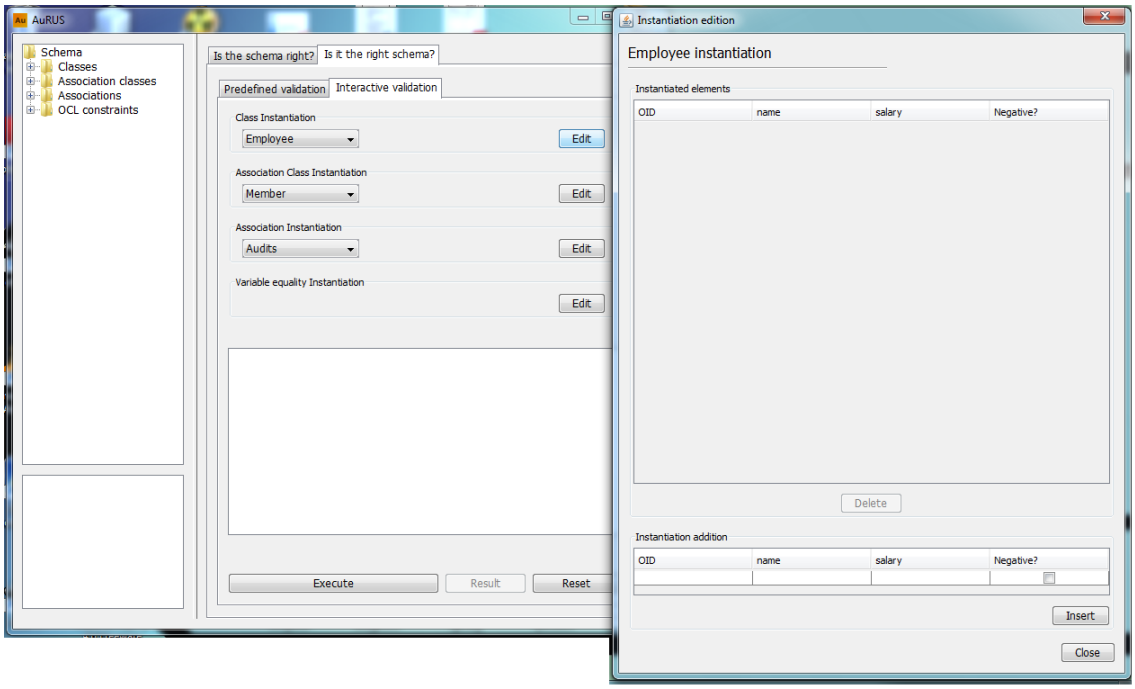


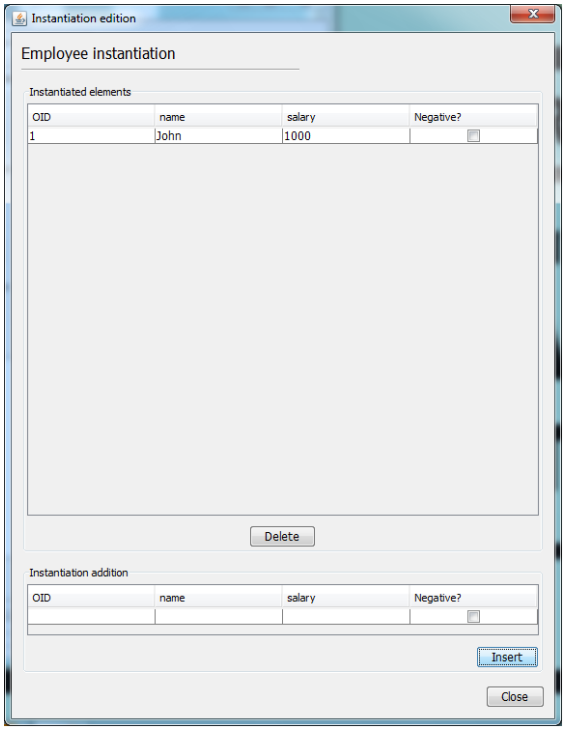
Interactive Validation

In order to completely validate your schema, it is necessary to make some custom test to check if the schema represents exactly your domain.

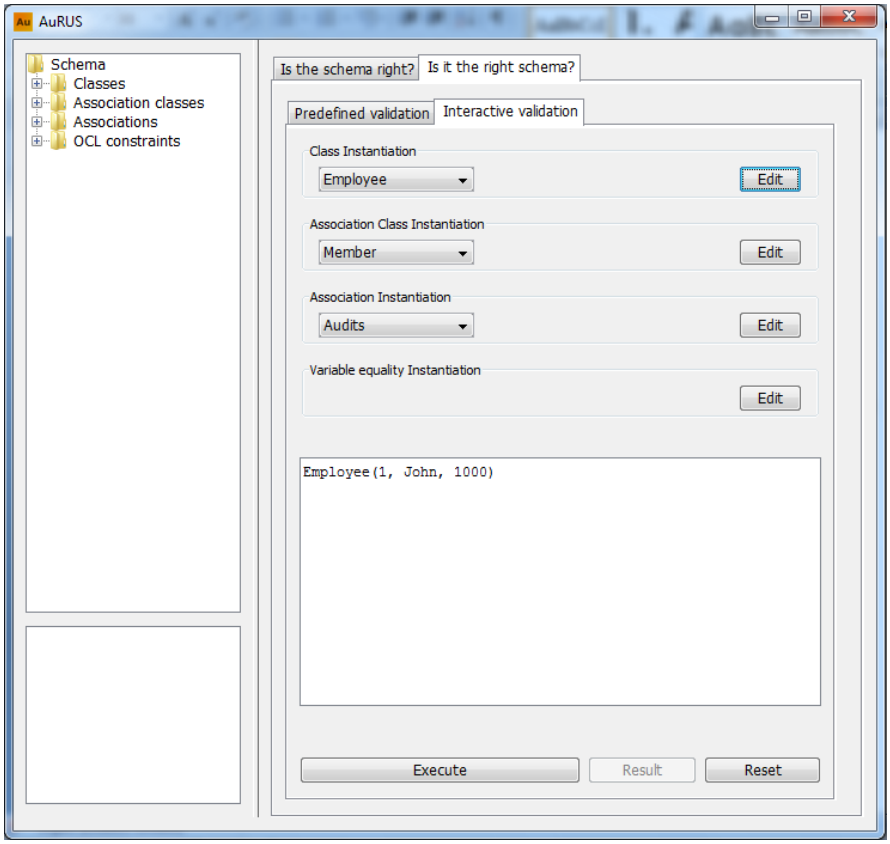
This checking is made by some scheme-reachability tests. That is, the user defines a concrete instance for the schema, and the reasoner tries making that instance consistent by adding new instances to the original ones. If it is not possible to make it consistent, AuRUS will answer the restrictions that cause this situation. However, this response is not ensured in case that the decidability test would have failed.

For example, we can check if there can be an employee called John whose salary is 1000. To perform this test, we have to select the Interactive Validation tab, and look through the Class Instantiation the Employee class. After that, we can push the Edit button and define the desired attributes as commented.





In this panel you can add or delete as rows as you want. All the modifications will be saved in the schema instance after closing it, which is displayed in the main panel:



Constants, variables and not defined

You can define the instances using constants, variables and/or omitting some columns.

Every String will be considered as a variable while real numbers will be interpreted as constants. So, in the last example the name “John” will be interpreted as a variable.

Variable values will be substituted by the reasoner for constants to find a schema instance which is consistent. Again, if there is no such substitution, AuRUS will answer the restrictions involved in that fact.

Notice that you can use the same variable in other instances, even in instances of other types. For example, you can test if there can be two employees with the same salary “a” which is the maxSalary of a defined Department.

Nevertheless, if you do not care of the value of a column, you can leave the definition in blank if it does not represent an OID value, this definitions would be displayed as “null” in the main panel.

Negative instances and Variable Equality

AuRUS permits you to define some not desirable instances checking the column called “Nevative?”. In this case, the reasoner will never use them to make consistent the defined schema instance.

However, it is not allowed to add a negative instance which uses variables that do not appear in positive instances.

With negative instances, you can test some scheme-instances like:

```
WorksIn(Emp, Dep)
Department(Dep, DepName, 500, max)
Not(Employee(Emp, null, 100))
```

This test can be used to proof that it is not possible to have an employee in a department with a salary lower than the departments’ specification.

However, this example is not exactly true, since it can only test that property for the specific values 500 and 100.

The correct test would be

```
WorksIn(Emp, Dep)
Department(Dep, DepName, min, max)
Not(Employee(Emp, null, sal))
Sal < min
```

Where sal < min can be defined in using the variable equality instantiation option, which allows you to define <,>,<=,>=,=,<> between variables and constants.