User-guide to modeling language for AMS601.3

Table of contents

- Scope
- Entities
  - RObject
    - properties
    - operations
  - RRelation
    - properties
    - operations
  - RModel
    - properties
    - operations
- Operations
  - Abstraction
  - Flattening
  - Comparison
- Miscellaneous
  - RPickle
  - RException
- Examples
  - Load model
  - Change model
  - Abstract model
  - Flatten model
  - Compare models
  - Save model

Date of change	Author	Change
12/12/2013	Valentin Sviridov	Initial version of documentation
14/12/2013	Valentin Sviridov	Finished initial version of documentation and added examples

Authors: Khushbu Bhalla, Carlos Kekwa, Valentin Sviridov

Version: 0.2

Scope

This documents gives a brief introduction into the modeling language tool developed for the course AMS601.3. This introduction consists in defining the main concepts used to model and the main functions that make the value of the tool

The tool is an application written in Python3 language that implements specifications defined in the document “AMS 601.3 language” dated from 25/11/2013 (http://goo.gl/hBD6oM).

Entities

There are three concepts that are used for modeling:
RObject - represents an object
RRelation - represents a relation
RModel - represents a set of objects and relations that produce a model

---

The properties and operations listed below may be used from user-code to change model.

RObject

properties

`extends`
string that holds name of “prototype” RObject defined in the same model or in the library

`objects`
dictionary that holds references to RObject objects

`relations`
dictionary that holds references to RRelation objects

`properties`
dictionary that holds string pairs that represent properties

operations

`parse(dict)`
static function that takes Python dictionary object that holds RObject representation (usually result of JSON deserialization) and returns actual RObject

---

RRelation

properties

`extends`

string that holds name of “prototype” RRelation defined in the same model or in the library

`from_ids`
list that holds names (strings) of objects defined in the same model or in the library that are a part of from-part of the relation

`to_ids`
list that holds names (strings) of objects defined in the same model or in the library that are a part of in-part of the relation

`directional`
boolean value that determines if the relation is directional or not

`properties`
dictionary that holds string pairs that represent properties

operations

`parse(dict)`
static function that takes Python dictionary object that holds RRelation representation (usually result of JSON deserialization) and returns actual RRelation

---

RModel

RModel is an extention of the RObject that has some additional properties and operations

properties

`library`
string that holds filepath to the library of the model

operations

`parse(dict)`
static function that takes Python dictionary object that holds RModel representation (usually result of JSON deserialization) and returns actual RModel

`compare(another_model)`
compares current model with another models and prints out the differences; detailed explanation is given further

`flatten()`
returns flattened version of the model; detailed explanation is given further

`abstract(levels)`
returns abstracted version of the model; detailed explanation is given further

Operations

Abstraction

Abstraction takes a model and a `levels` parameter that is a number that defines how many levels in deep we want to see and returns a new model without objects and relations with the nesting level superior to `levels`.

The goal of this operation is having a simple representation of the model. You mustn't use the returned model for any reasons other than printing because this operation can easily introduce inconsistency to the model.

Pseudo-code:

```
abstract(model, levels)
      model_copy ← clone(model)
      abstract_helper(model_copy)
      return model_copy

abstract_helper(model, level)
      if level > 0
            foreach obj in model.objects
                  abstract_helper(obj, level - 1)
            foreach rel in model.relations
                  abstract_helper(rel, level - 1)
      else
            model.objects ← {}
            model.relations ← {}
            model.properties ← {}
```

Flattening

Flattening takes a model and returns a new model with all nesting removed that is all object, relations and properties become the immediate descendants of the root object.

The goal of this operations is having a simple representation of the model. You shouldn't use the returned model for any reasons other than printing because this operation may easily introduce inconsistency to the model.

Pseudo-code:

```
flatten(model)
      model_ref = clone(model)
      model ← empty_model
      flatten_helper(model_ref, model)
```

```
    return model

flatten_helper(entity_ref, model)
    foreach obj in entity_ref.objects
        flatten_helper(obj, model)
        model.objects ← obj
    foreach rel in entity_ref.relations
        flatten_helper(rel, model)
        model.relations ← rel
    foreach prop in entity_ref.properties
        model.properties ← prop
```

Comparison

Comparison takes two models and prints out differences found between two models. Differences include added or removed objects, relations, properties. The output format is aimed for human analysis.

Pseudo-code:

```
key_changes(keys1, keys2)
    added, deleted, removed ← [ ], [ ], [ ]

    foreach key in keys1
        if k in keys2
            common ← key
        else
            deleted ← key

    foreach key in keys2
        if k not in keys1
            added ← key

    return added, deleted, common

compare_relation(rel1, rel2)
    to_added, to_removed, to_common ← key_changes(rel1.to,  rel2.to)
    from_added,  from_removed,  from_common ← key_changes(rel1.from,
rel2.from)

    foreach k in to_added
        print(k to added)

    foreach k in from_added
```

```
            print(k from added)

      foreach k in to_removed
            print(k to removed)

      foreach k in from_removed
            print(k from removed)

      compare_properties(rel1.properties, rel2.properties)

compare_properties(props1, props2)
      prop_added,  prop_removed,  prop_common  ←  key_changes(props1,
props2)

      foreach k in prop_added
            print(k added)

      foreach k in prop_removed
            print(k removed)

      foreach k in prop_common
            if props1[k] <> props2[k]
                  print(k changed)

compare_object(obj1, obj2)
      obj_added,  obj_removed,  obj_common  ←  key_changes(obj1.objects,
obj2.objects)
      rel_added, rel_removed, rel_common ← key_changes(obj1.relations,
obj2.relations)

      foreach k in obj_added
            print(k added)

      foreach k in obj_removed
            print(k removed)

      foreach k in rel_added
            print(k added)

      foreach k in rel_removed
            print(k removed)

      foreach k in obj_common
```

```
        compare_object(obj1.objects[k], obj2.objects[k])

    foreach k in rel_common
        compare_relation(obj1.relations[k], obj2.relations[k])

    compare_properties(obj1.properties, obj2.properties)
```

Miscellaneous

RPickle

RPickle class with its static methods is a façade to the persistence layer operations like reading, converting and saving.

`text_to_dict(text)`
converts JSON-represented model to Python dictionary

`file_to_text(filename)`
returns contents of a file filename

`file_to_dict(filename)`
returns dictionary build from JSON-represented model stored in file filename

`to_text(rentity)`
converts Python dictionary to JSON-representation of model

`text_to_file(filename, text)`
saves given text to a file filename

`JSON_pretty_format(js)`
pretty formats Python object to be printed on the screen

RException

RException is an exception that is thrown in case of logical errors found in modeling activity

---

Examples

In this section you can find several examples to illustrate how to load a model, make some changes to it, abstract, flatten, compare and save.

Load model

Create model in-place

```
from rauzy import RModel

root = """
{
    "nature": "object",
    "extends": null,
    "objects": {},
    "relations": {},
    "properties": {
        "prop1": "value1",
        "prop2": "value2"
        },
    "library": "./inputFileExamples/library.json"
}
"""
model = RModel.parse(RPickle.text_to_dict(root))
print(model)
```

Load model from file

```
from rauzy import RModel, RPickle

root = RPickle.file_to_text("./inputFileExamples/geo.json")
model = RModel.parse(RPickle.text_to_dict(root))
print(model)
```

Change model

```
from rauzy import RModel, ROject, RRelation, RPickle

root = RPickle.file_to_text("inputFileExamples/geo.json")

model1 = RModel.parse(RPickle.text_to_dict(root))
model2 = RModel.parse(RPickle.text_to_dict(root))

# changing property in an object
model1.properties["prop1"] = "old_prop"
model2.properties["prop1"] = "changed_prop"

# adding property in an object
model2.properties["prop2"] = "new_prop"

# removing property from an object
```

```
del model2.objects["Europe"].properties["population"]

# changing property in a relation
model1.objects["Europe"].objects["France"]\
    .relations["postIndependencePeace"]\
    .properties["duration"] = "230 years"

model2.objects["Europe"].objects["France"]\
    .relations["postIndependencePeace"]\
    .properties["duration"] = "100 years"

# adding property in a relation
model2.objects["Europe"].objects["France"]\
    .relations["postIndependencePeace"]\
    .properties["duration2"] = "300 years"

# removing property in a relation
del model2.objects["Europe"].objects["France"]\
    .relations["postIndependencePeace"].properties["treaty"]

# objects
obj_repr = \
"""
    {
        "nature": "object"
    }
"""

model2.objects["added_object"] = \
    RObject.parse(RPickle.text_to_dict(obj_repr))

model1.objects["deleted_object"] = \
    RObject.parse(RPickle.text_to_dict(obj_repr))

# relations
relation_repr = \
"""
    {
        "nature": "relation"
    }
"""

model2.relations["added_relation"] = \
```

```
            RRelation.parse(RPickle.text_to_dict(relation_repr))

        model1.relations["deleted_relation"] = \
            RRelation.parse(RPickle.text_to_dict(relation_repr))

        model1.relations["common_relation"] = \
            RRelation.parse(RPickle.text_to_dict(relation_repr))

        model2.relations["common_relation"] = \
            RRelation.parse(RPickle.text_to_dict(relation_repr))

        model1.relations["common_relation"].from_ids.append("France")
        model2.relations["common_relation"].from_ids.append("Europe")
        model2.relations["common_relation"].to_ids.append("North
America")

        model1.update_references()
        model2.update_references()
```

Abstract model

```
        from rauzy import RModel, RPickle
        root = RPickle.file_to_text("inputFileExamples/geo.json")
        model = RModel.parse(RPickle.text_to_dict(root))
        print(model.abstract(1))
```

Flatten model

```
        from rauzy import RModel, RPickle
        root = RPickle.file_to_text("inputFileExamples/geo.json")
        model = RModel.parse(RPickle.text_to_dict(root))
        print(model.flatten())
```

Compare models

```
        from rauzy import RModel, RPickle
        root1 = RPickle.file_to_text("inputFileExamples/geo.json")
        root2 = RPickle.file_to_text("inputFileExamples/zoo.json")

        model1 = RModel.parse(RPickle.text_to_dict(root1))
        model2 = RModel.parse(RPickle.text_to_dict(root2))

        model1.compare(model2)
```

Save model

```
from rauzy import RModel, RPickle
filepath = '/tmp/rtest.txt'
root = RPickle.file_to_text("inputFileExamples/geo.json")

model1 = RModel.parse(RPickle.text_to_dict(root))
RPickle.text_to_file(filepath, RPickle.to_text(model1))
```