requs

REQUS, White Paper

Yegor Bugayenko

December 19, 2021

Abstract

Requirements Definition and Query Language (requs) is an experimental set of tools and formats that enable definition of SRS documents in a human-friendly language at the same time making it understandable by computers. Plain text functional requirements are converted to second order logic predicates and then represented in MOF/XMI.

1 Introduction

IEEE (1998) says: "*Software Requirements Specification should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, traceable*". requs enables the creation of such documents in plain text format.

First, we convert plain English into objects (in terms of object-oriented programming). Next, we convert objects into second order logic predicates. Then, we resolve predicates on data scope (in terms of logical programming) and create a collection of all possible scope variants. Then, for every scope variant we build a model in terms of MOF, see Group (2006), converting them later into XMI and transferring to the destination. Next, XMI, see OMG (2007), can be rendered for an end-user or understood by a computer. Finally, the entire process is delivered to end-users in customizable XML, see Bray et al. (2008), reports. The workflow of the process explained is presented in Fig.1.

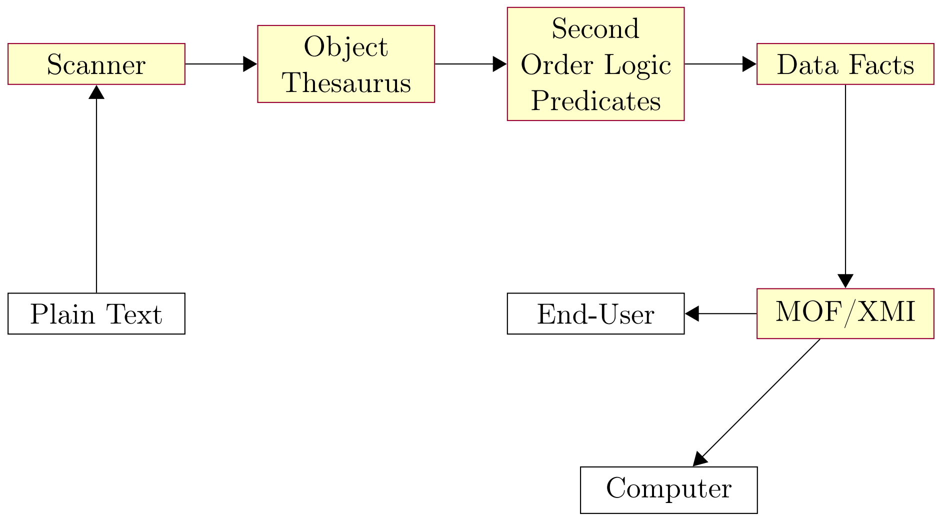


Figure 1: requs highest-level workflow description, where we start from plain text and finish with a strict formal description of a testing model, visible to an end-user and a computer.

Section 2 explains the process of converting plain text to "Object Thesaurus". We are using `antlr3` grammar analysis toolkit. Moreover, the entire requs product is written in Java.

Section 3 explains how objects from the Thesaurus are converted to second order logic "predicates" and validated. Here we also discuss the interconnection with Lisp, programming language explained by Graham (1993).

Section 4 explains the process of converting of second order predicates to Prolog-style data "facts", and reveals the internal structure of said facts. We discuss interconnection with Prolog, programming language explained by Shapiro and Sterling (1994).

Section 6 is about the interconnection between `requs-bin.jar` command-line interface tool and its users.

Section 5 explains the mechanism of conversion of data facts into test cases.

Section 7 contains the most important architecture and design decisions documented in diagrams.

2 Plain English to Object Thesaurus

Detailed explanation of the requs syntax is given on its website www.requs.org.

Fig.2 explains what Java classes we use to store objects retrieved from requs texts. All said Java classes are declared in the `org.requs.thesaurus` package.

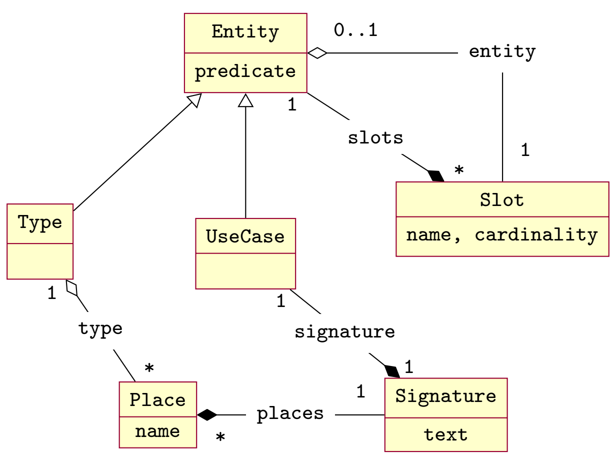


Figure 2: Java classes in the Object Thesaurus.

The instance of class `Entity`, which is a holder of all other types, is created by `InputText.toType()` method.

2.1 Types

A simplified verbal form of a requs type looks like this:

```
User is a "human being".
User includes:
  email: "email address";
  face-s: Photo "a collection of photos";
  manager-s?: User "user's managers, if any";
  address.
Address of User includes: city, country.
```

In Java the type is presented by class `Type` and the type written above will look like shown in Fig.3. Symbol `"face"/0..*` means an instance of class `Slot` with `name` of `"face"` and `cardinality` of `0..*`.

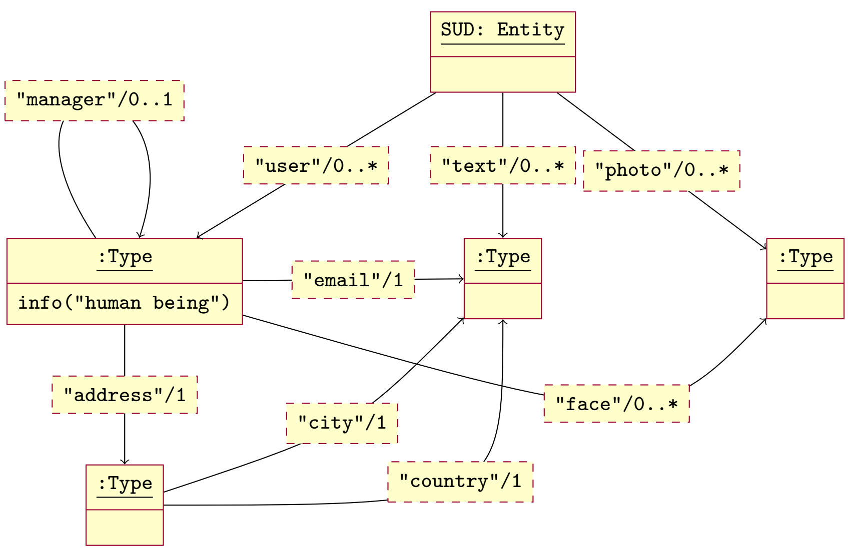


Figure 3: Simple requs type is translated into Thesaurus objects and relations between them.

2.2 Use Cases

There is a simple verbal form of an informal use case:

```
UC1: User (the user) personalizes himself: "the user
  uploads a new photo, SUD validates it and saves".
```

Such a simple use case is represented in the Thesaurus as explained in Fig.4

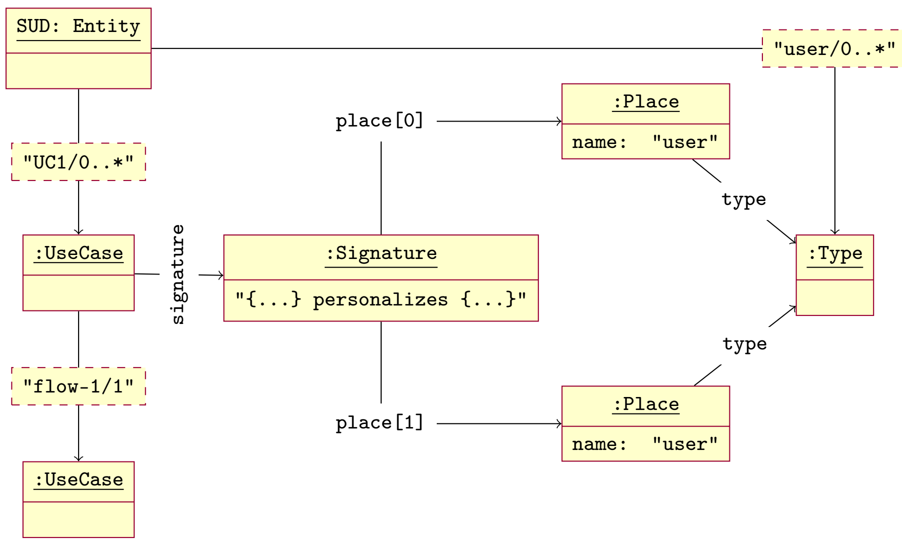


Figure 4: Simple requs use case is translated into Thesaurus objects and relations between them.

A signature is a text with "places" inside, for example `{...} personalizes {...}`. There are two places inside this signature, each of which has to be explained by an instance of class `Place`. Such an instance contains a link to `Type` and a name. The link can't be `null`, while the name can. If the name is not set it means that this is just a link to a type, and it is not instantiated.

There is a more complex example of the same use case:

```
UC1: User (the user) personalizes himself:
  1. The user creates photo of himself (the photo).
  2. We validate the photo "immediately".
UC1: 2a) If failed with "file format is not valid":
  1. We delete the photo.
  2. Fail with "only PNG images are accepted".
```

The use case is represented in the system by means of `UseCase` and `Slot`-s. The use case given above will be converted to the set of objects in Fig.5. For the sake of simplicity the diagram contains only `UseCase` types, and their relative signatures and predicates.

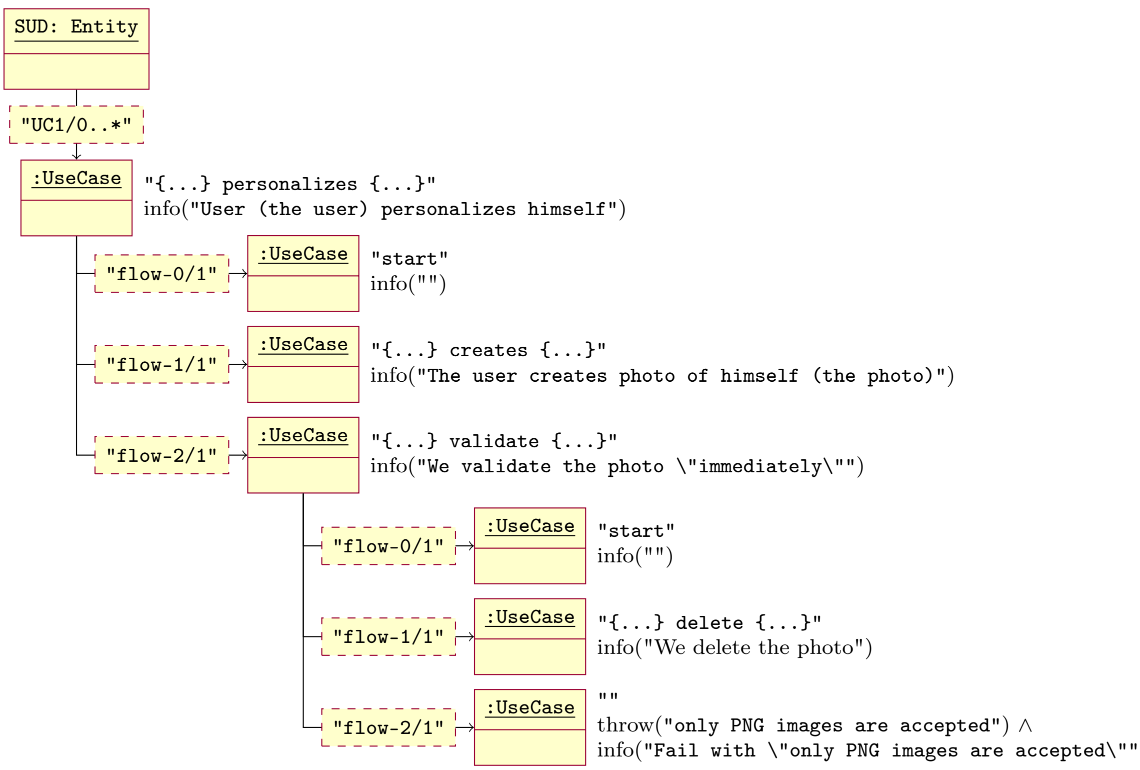
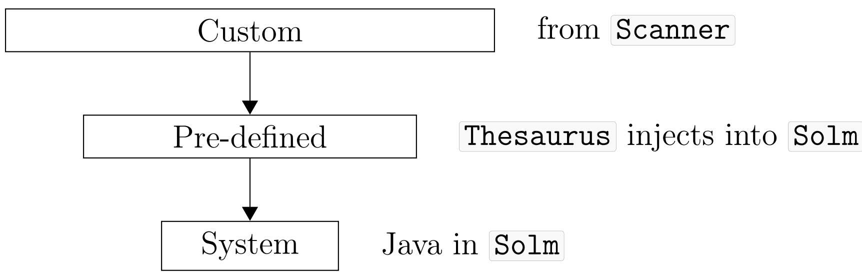


Figure 5: More complex requs use case translated.

3 Object Thesaurus to SOL Predicates

`org.requs.thesaurus.Type` class can be converted into second order logic predicate: `org.requs.solm.Predicate`. There are three groups of predicates. The first one is “system predicates”, which are Java encoded. The second group includes “pre-defined predicates”, which are injected into `Solm` by `Thesaurus`. The third group includes predicates defined by custom document, called “custom predicates”:



3.1 System Predicates

System predicates and formulas are (x_i is a variable, p_i is a predicate, X_i is a set of variables):

`true`	always true
$\alpha(x_1, x_2, \ldots, x_n) : p.$	declaration of a new predicate α
$\exists x(p)$	existence quantifier
$\forall x(p)$	universal quantifier
$p_1 \vee p_2 \vee \ldots \vee p_n$	logical disjunction
$p_1 \wedge p_2 \wedge \ldots \wedge p_n$	logical conjunction
$p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$	logical implication
$x \in X$	x belongs to set X
$\neg p$	predicate p is false
$x_1 = x_2$	variable x_1 equals to variable x_2
$x_1 < x_2$	variable x_1 is less than variable x_2
`kind`(x_1, x_2)	x_1 is a variable of kind x_2
`re`(x_1, x_2)	variable x_1 matches regular expression x_2
`ac`(x_1, x_2, x_3)	variable x_1 has access x_3 to variable x_2

A variable can be represented by an *atom*, which starts with a single quotation mark, like in Lisp Graham (1993), for example:

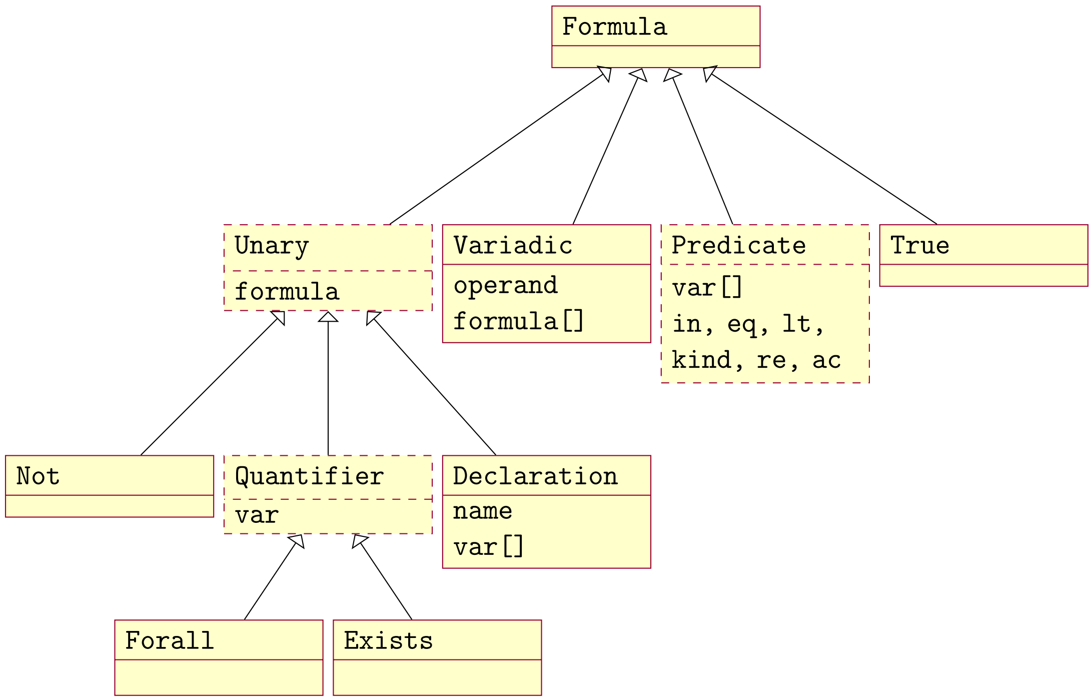
$$\exists X(\forall x(x \in X \rightarrow x < \texttt{'5}) \wedge X = \texttt{'3})$$

A variable can be represented by a *set*, which looks like:

$$\exists x(x = \{y, \texttt{'5}\})$$

Only declaration and quantifiers can declare a variable (create a new variable). All other predicates will report a fatal error if an undefined variable is passed to them.

Predicates are realized in `Solm` as a collection of *formulas* (classes inherited from `Formula`):



This structure is fixed and is not going to change. All other predicates are defined on top of these.

3.2 Pre-defined Formulas

These predicates are predefined in `Solm`:

`created`(x_1, x_2) : `ac`$(x_1, x_2,$ `'C`$)$.
`read`(x_1, x_2) : `ac`$(x_1, x_2,$ `'R`$)$.
`updated`(x_1, x_2) : `ac`$(x_1, x_2,$ `'U`$)$.
`deleted`(x_1, x_2) : `ac`$(x_1, x_2,$ `'D`$) \wedge \forall x(x \neq x_1)$.
`exception`(x) : $\exists x_2(x_2 = x \wedge$ `kind`$(x_2,$ `'ex`$))$
`throw`(x) : `kind`$(x,$ `'ex`$) \wedge \neg$`true`
`info`(x) : `kind`$(x_2,$ `'info`$)$.
`silent`(x) : `kind`$(x_2,$ `'silent`$)$.
`err`(x) : `kind`$(x_2,$ `'err`$)$.
`number`(x) : `kind`$(x,$ `'number`$) \wedge$ `re`$(x,$ `'[0-9]+`$)$.
`text`(x) : `kind`$(x,$ `'text`$)$.
`SUD`(x) : `kind`$(x,$ `'actor`$)$.

Also math binary predicates:

$x_1 > x_2 : \neg(x_1 < x_2) \wedge \neg(x_1 = x_2)$.
$x_1 \geq x_2 : \neg(x_1 < x_2)$.
$x_1 \leq x_2 : x_1 < x_2 \vee x_1 = x_2$.
$x_1 \neq x_2 : \neg(x_1 = x_2)$.

3.3 Types to Predicates

Let's convert a type defined above to second order logic predicates. First, we define predicates for slots inside the type (a few examples):

$$
\begin{aligned}
&\texttt{User.photo}(x,p) : \texttt{kind}(x, \texttt{'User}) \wedge \texttt{Photo}(p) \wedge \\
&\quad \exists r(\texttt{kind}(r, \texttt{'User.email}) \wedge r = \{x,p\}). \\
&\texttt{User.address.country}(x,p) : \texttt{kind}(x, \texttt{'User}) \wedge \texttt{text}(p) \\
&\quad \exists r(\texttt{kind}(r, \texttt{'User.address.country}) \wedge r = \{x,p\}).
\end{aligned}
$$

Then we create predicates for types (Π_i is a set):

$$
\begin{aligned}
&\texttt{User}(x) : \\
&\quad \texttt{kind}(x, \texttt{'User}) \bigwedge \\
&\quad \exists \Pi_1(\Pi_1 = \texttt{'1} \wedge \forall p(p \in \Pi_1 \rightarrow \texttt{User.email}(x,p)) \bigwedge \\
&\quad \exists \Pi_2(\Pi_2 \geq \texttt{'0} \wedge \forall p(p \in \Pi_2 \rightarrow \texttt{User.photo}(x,p)) \bigwedge \\
&\quad \exists \Pi_3(\Pi_3 < \texttt{'2} \wedge \forall p(p \in \Pi_3 \rightarrow \texttt{User.manager}(x,p)) \bigwedge \\
&\quad \exists \Pi_4(\Pi_4 = \texttt{'1} \wedge \forall p(p \in \Pi_4 \rightarrow \texttt{User.address.city}(x,p)) \bigwedge \\
&\quad \exists \Pi_5(\Pi_5 = \texttt{'1} \wedge \forall p(p \in \Pi_5 \rightarrow \texttt{User.address.country}(x,p)).
\end{aligned}
$$

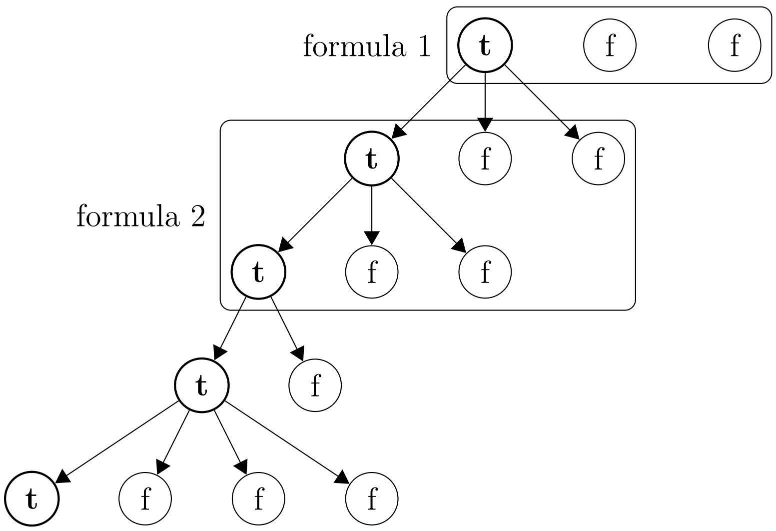
3.4 Use Case to Predicate

Every use case is a predicate, $\texttt{UC}_1(x) = \texttt{true}$ means that the user successfully extended his photo album:

$$
\begin{array}{l}
\texttt{UC}_1(x): \\
\quad \texttt{User}(x) \bigwedge \\
\quad \exists s(\texttt{SUD}(s)) \bigwedge \\
\quad \exists \Pi(\forall p(p \in \Pi \rightarrow \texttt{User.photo}(x,p))) \bigwedge \\
\quad \neg(\Pi > 5) \bigvee \\
\quad (\\
\qquad \texttt{info}(\texttt{'If number of photos of the user is greater than 5}) \bigwedge \\
\qquad \exists y(y \in \Pi) \bigwedge \\
\qquad \texttt{deleted}(y,x) \wedge \texttt{info}(\texttt{'The user deletes photo of himself}) \\
\quad) \bigwedge \\
\quad \exists p(p \in \Pi) \bigwedge \\
\quad \texttt{created}(p,x) \wedge \texttt{info}(\texttt{'The user creates photo of himself (the photo)}) \bigwedge \\
\quad \texttt{UC}_2(p) \wedge \texttt{info}(\texttt{'We validate the photo immediately}) \vee \\
\quad (\\
\qquad \texttt{exception}(\texttt{'file format is not valid}) \bigwedge \\
\qquad \texttt{deleted}(p,s) \wedge \texttt{info}(\texttt{'We delete the photo}) \bigwedge \\
\qquad \texttt{throw}(\texttt{'only PNG images are accepted}) \\
\quad) \bigwedge \\
\quad \texttt{silent}(\texttt{We protocol the operation in backlog}) \bigwedge \\
\quad \texttt{read}(p,x) \wedge \texttt{info}(\texttt{'The user reads the photo})
\end{array}
$$

4 Predicates to Snapshots

Every formula produces a number of “fact paths”. Every fact path is a vector of “facts”, and every fact is either a positive or negative:



Every fact includes a “snapshot” of persistent data. Snapshot includes: objects. Every object has a name, value, type, and may have a number of changes made by other objects:

```
class Object {
  int id; // could be zero
  string name; // could be empty
  string type; // mandatory
  Value* value;
  vector<AclRule> rules;
};
class Value {};
class ValueString : public Value {
  string value;
};
class ValueSet : public Value {
  vector<int> ids;
};
class ValueAssociation : public Value,
  public pair<AssociationMember, AssociationMember> {};
class AssociationMember {};
class AssociationMemberId : public AssociationMember {
```

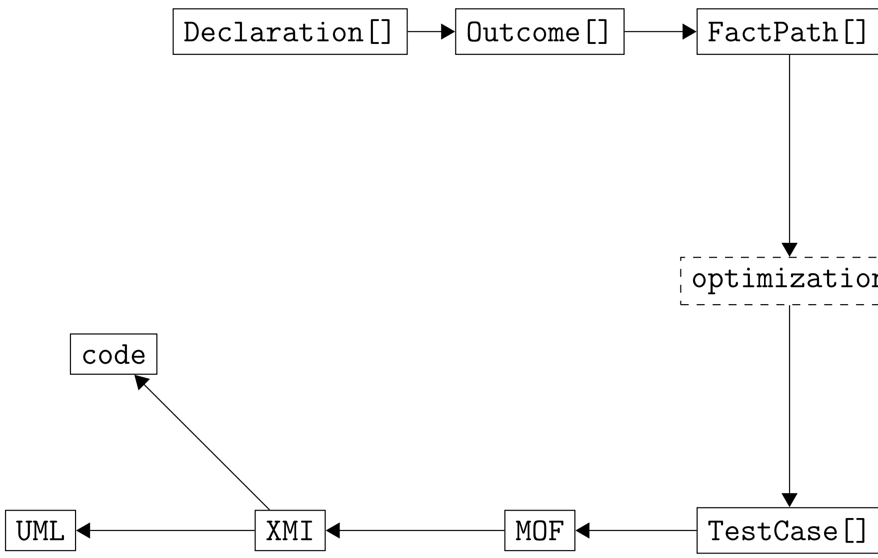
```
  int id;
};
class AssociationMemberName : public AssociationMember {
  string name;
};
class AclRule {
  enum {CREATE, READ, UPDATE, DELETE} operation;
  int id;
};
vector<Object> snapshot;
```

Consider this example:

SOLM formula	Snapshots				
	Name	Type	ID	Value	ACL Rules
`UC`$_1(x)$:					
`User`$(x) \rightarrow$	x	`User`	`1`		
$\exists\Pi($`User.photo`$(x,\Pi)) \rightarrow$	x	`User`	`1`		
	Π	`Photo`		`[?]`	
		`User.photo`		`1:`Π	
(					
$\lvert\Pi\rvert > 5 \rightarrow$	x	`User`	`1`		
	Π	`Photo`		`[2]`	
		`Photo`	`2`		
		`User.photo`	`3`	`1:2`	
$\exists y(y \in \Pi) \rightarrow$					
`deleted`(y)					
$) \rightarrow$	*skipped*				
$\exists p(p \in \Pi) \rightarrow$	x	`User`	`1`		
	Π	`Photo`		`[2]`	
	p	`Photo`	`2`		
		`User.photo`	`3`	`1:2`	
`created`$(p,x) \rightarrow$	x	`User`	`1`		
	Π	`Photo`		`[2,4]`	
	—	`Photo`	`2`		
	p	`Photo`	`4`		`CREATE:1`
		`User.photo`		`1:2`	
		`User.photo`	`5`	`1:4`	
`UC`$_2(p)\vee$					
(					
`exception`(“file format...”) $\rightarrow$					
`deleted`$(p) \rightarrow$					
`throw`(“only PNG...”)					
$) \rightarrow$	*skipped*				
`silent`(“We protocol...”)					
`read`(p,x)	x	`User`	`1`		
	Π	`Photo`		`[2,4]`	
		`Photo`	`2`		
	p	`Photo`	`4`		`CREATE:1`, `READ:1`
		`User.photo`		`1:2`	
		`User.photo`		`1:4`	
		`silent`	`6`	“We protocol...”	

5 Test Cases

We optimize vectors of snapshots, in order to produce the smallest collection. Then, we convert a vector of snapshots into MOF meta-model. The process looks like this:



6 Command Line Interface

CLI is a collection of components. Every component gets an associative array of configuration parameters and returns an XML element. The element is named `<report>` and has attributes equivalent to the configuration params provided.

`main()` returns an XML document that integrates all reports retrieved:

```
<?xml version="1.0" ?>
<requs>
  <errors>
    <report>
```

```
      <error line="23">This is an error</error>
    </report>
  </errors>
  <metrics>
    <report>
      <ambiguity>0.765</ambiguity>
    </report>
  </metrics>
  <uml>
    <report uc="UC6.5">
      <uml><[CDATA[....]]></uml>
    </report>
  </uml>
</requs>
```

The script shall be called from command line like this:

```
$ java -jar requs-bin.jar \
errors uml:uc=UC6.5 \
metrics \
< myscope.txt
```

Possible reports are:

- `errors`: full list of errors
- `metrics`: full analysis of the scope ambiguity, size, intensity, etc.
- `links`: report links between objects (line to line)
- `uml:uc=UC5,type=ActorUser,...`: description of types and use cases in UML
- `svg:uc=UC5,type=ActorUser,...`: description of types and use cases in SVG
- `tc:uc=UC5,...`: Test Cases for the given UC-s (or all)

7 Architecture and Design

Architectural and design decisions are documented at requs.org.

References

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible markup language (XML) 1.0 (fifth edition).

Graham, P. (1993). *On LISP: Advanced Techniques for Common LISP*. Prentice Hall.

Group, O. M. (2006). Meta object facility (MOF) core specification, version 2.0. Technical Report formal/06-01-01, Object Management Group.

IEEE (1998). Recommended practice for software requirements specifications. Technical Report IEEE Std 830-1998, The Institute of Electrical and Electronics Engineers.

OMG (2007). MOF 2.0/XMI mapping, version 2.1.1. Technical report, Object Management Group.

Shapiro, E. and Sterling, L. (1994). *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press.