Using Jaza to animate RoZ specifications of UML class diagrams

Y. Ledru
Université Joseph Fourier (Grenoble 1) - LSR-IMAG
BP 72, 38402 St-Martin-d'Hères, France
Yves.Ledru@imag.fr

Abstract

One of the goals of the integration of formal and graphical methods is to reuse tool support associated with formal methods. This paper reports on a combination of the Jaza Z animator with the RoZ tool. RoZ produces a Z specification from an annotated UML class diagram. It also generates the specification of basic operations associated to the diagram. The paper recalls the principles of the RoZ tool, gives a typical animation session, and discusses how RoZ and class diagrams must be adapted to support animation.

1. Introduction

In the last decade, numerous efforts have tried to integrate formal methods with graphical notations, such as UML[1]. The goal is to combine popular and intuitive notations with the precision of formal languages. A recent study [12] compares the use of the B method and the use of UML class and state-transition diagrams [1] on the same project. It concludes that B leads to a better precision than UML while UML produces more intuitive and readable documents. The authors advocate for a combination of both methods. Actually, a combination of both methods should bring more precise semantics to graphical formalisms, and favour a wider use of formal method tools. In the area of model-based languages such as Z, B or VDM, attempts have been made to link UML with VDM++ [8], B [13, 7, 9], Z [4, 3] or Object-Z [2, 6]. These works have shown that formal methods could make the semantics of graphical languages more precise, but using formal method tools to exploit the resulting formal model remains a difficult task.

The RoZ tool [3] results from one of these integration efforts. It is based on a set of rules which translate a UML class diagram, annotated with Z assertions, into a full Z specification. The tool also automatically generates the specification of basic operations associated to a data model. In [10], the Z-Eves tool[11] was used in conjunction with RoZ to establish some consistency properties and help identify operation preconditions. Still, while provers help us construct consistent specifications, they don't solve the validation problem: did we write the right specification? Animators can help us. They provide a way to experiment with the specification, and to compare its behaviour with the expected one. This paper reports on our attempt to combine RoZ with the Jaza Z animator [1].

This paper shows how Jaza can be used with RoZ, but also how RoZ had to be adapted to produce suitable input for Jaza. Additional rules were added to RoZ to make its specifications more executable. Using an animator also helped to clarify several issues related to the promotion of operations, and of their scope. Sect. 2 presents RoZ, on the basis of a small example. Sect. 3 then shows how the example can be animated using Jaza. Sect. 4 discusses how RoZ has evolved to adapt to Jaza, and how class diagrams must be adapted to fit the RoZ/Jaza needs. Finally, Sect. 5 discusses the current limitations of the tool, and draws the conclusions and perspectives of this work.

2. Roz

RoZ is a set of scripts for the Rational Rose^{TM} environment[2]. They translate the structure of UML class diagrams into Z specification skeletons, and fill the Z specification with several annotations of the class diagram. RoZ also supports the generation of basic operations and of several proof obligations (not covered here but discussed in [10]).

2.1. The access control example

Fig. 1 shows the class diagram of the access control example. In this diagram, *PERSON* models the set of people who might access a set of buildings (represented by class *BUILDING*). Each person has a first and lastname, and is identified uniquely by a number (*cardnb*) which corresponds to the smart card used to access the buildings. This

1 http://www.cs.waikato.ac.nz/~marku/jaza/
2 http://www.rational.com

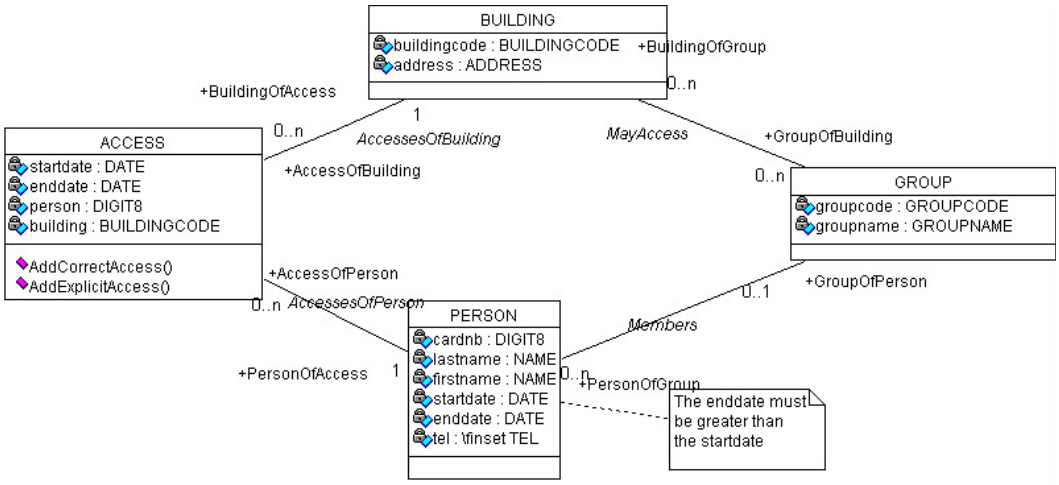

Figure 1. The class diagram of the access control example

card has a validity period characterized by start and end dates. Each person is also associated to a non-empty set of phone numbers. *BUILDING* is another class in this diagram, each building being characterized by some identifier (*buildingcode*) and its address.

Several constraints are associated to that class. In UML, these may be expressed in natural language, as illustrated on Fig. 1 where a constraint is stated on the start and end dates of a person. OCL provides another way to state these constraints. In RoZ, we choose to express these constraints in Z (see Sect. 2.3).

Each access of a person to a building is stored in an instance of *ACCESS*. An access corresponds to one and only one person, one and only one building, and the instant when the access begins (*startdate*). When the person exits the building, the end time of the access is recorded (*enddate*). Access of a person to a building is granted on the basis of the membership of this person to a group which may access the building. Class *GROUP* represents these groups, and associations *Members* and *MayAccess* record the members of a group, and the set of buildings a group may access.

2.2. Translation into Z

Types Fig. 1 is a classical class diagram, except that attribute *tel* of *PERSON* is declared of type `\finset TEL`. In RoZ, each attribute type must correspond to a Z set. These are declared in a separate file. Here are the set declarations for the access control example.

$[NAME, GROUPCODE, GROUPNAME]$
$[BUILDINGCODE, ADDRESS]$
$DATE == \mathbb{N}$
$TEL == 0 \,..\, 9999999999$
$DIGIT8 == 0 \,..\, 99999999$

tel being a multivaluated attribute, it is declared as an element of set $\mathbb{F}\, TEL$. Please note that in order to benefit from comparison operators, dates are natural numbers.

Some attributes may remain undefined for some time, e.g. *enddate* is undefined during the access. Therefore, we introduce a constant, named *undefineddate* which is one particular date that will be used to denote the undefined value. The axiomatic definition does not prescribe which value should be chosen; typically, it will be some very large value. This choice is left for later stages in the development.

$$undefineddate : DATE$$

Z skeleton Each class can then be translated into a pair of Z schemas. The first one, e.g. *PERSON*, is a schema type which corresponds to the structure of instances of the class. The second one, e.g. *PersonExt* declares the set of all instances of the class that are stored in the information system. The first schema corresponds to all possible instances; it is called the "intension" of the class. The second schema corresponds to the current instances of the class; it is called the "extension" of the class.

PERSON can be completed with constraints on the attributes of a given instance. *PersonExt* may feature more global constraints involving several instances. These will be discussed in Sect. 2.3.

PERSON
$cardnb : DIGIT8$
$lastname : NAME$
$firstname : NAME$
$startdate : DATE$
$enddate : DATE$
$tel : \mathbb{F}\, TEL$
...

PersonExt
$Person : \mathbb{F}\, PERSON$
...

Similarly, class *GROUP* is translated into:

GROUP

$groupcode : GROUPCODE$
$groupname : GROUPNAME$

...

GroupExt

$Group : \mathbb{F}\, GROUP$

...

The associations which link the classes, e.g. *Members*, are also translated into Z specifications.

MembersRel

$PersonExt;\ GroupExt$
$PersonOfGroup : GROUP \pfun \mathbb{F}\, PERSON$
$GroupOfPerson : PERSON \pfun GROUP$

$\mathrm{dom}\, GroupOfPerson \subseteq Person$
$\mathrm{ran}\, GroupOfPerson \subseteq Group$
$PersonOfGroup = \{g : \mathrm{ran}\, GroupOfPerson \bullet$
$\quad g \mapsto \{p : \mathrm{dom}\, GroupOfPerson \mid$
$\quad\quad GroupOfPerson(p) = g \bullet p\}\}$

...

In this schema, *Members* is translated as a pair of functions which correspond to both roles of the association. Function *PersonOfGroup* maps a group to a set of persons, because the role is multivaluated (0..n), while the monovaluated *GroupOfPerson* links a person to a single group. Three constraints are added to the diagram. The first two constraints express that the domain and range of *GroupOfPerson* are subsets of the extensions of classes *PERSON* and *GROUP*. In other words, you may not record a link between two instances that are not stored in the information system. Since these constraints refer to the extensions of the classes, schemas *PersonExt* and *GroupExt* are included in *MembersRel*.

The last constraint is a rather complex set comprehension expression, aimed at constructing *PersonOfGroup* from *GroupOfPerson*. It expresses that function *PersonOfGroup* links each group g, which appears in the range of *GroupOfPerson* to the set of persons who map to that group through function *GroupOfPerson*. Schema *MembersRel* also leaves space to express some constraints on the functions of the association.

The remaining classes and associations of the diagram are translated similarly, and finally grouped into a global schema which includes all associations, and transitively, all extensions of classes. Once again, this schema leaves space for adding constraints on the whole data structure.

GlobalView

MayAccessRel
AccessesOfBuildingRel
MembersRel
AccessesOfPersonRel

...

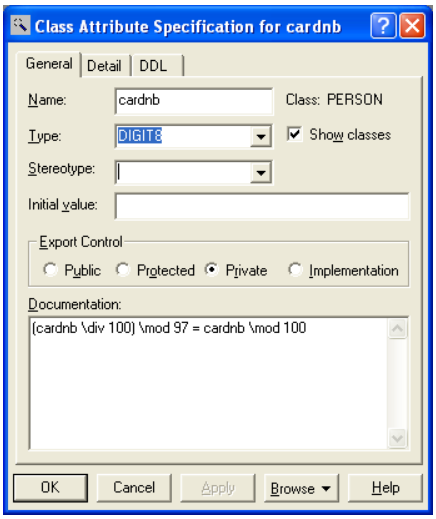

Figure 2. RoZ constraint for cardnb

2.3. Constraints

The graphical formalism has a limited expressiveness. Complex data structures usually include invariant constraints. In UML, such constraints can be expressed in OCL [15]. As shown in Fig. 2, RoZ allows to express these constraints directly in the Rational Rose environment, as Z annotations in the various "documentation" fields of the class diagram. In the access control example, we want to express the following constraints.

1. The last two digits of *cardnb* are a checksum.
2. When *enddate* is defined, it is always greater than *startdate* (for both classes *PERSON* and *ACCESS*).
3. The keys of classes are: *cardnb* for *PERSON*, *groupcode* or *groupname* for *GROUP*, *buildingcode* for *BUILDING*, and the combination of *person*, *building* and *startdate* for *ACCESS*.
4. Members of the same group have the same telephone prefix.
5. Finally, if a person performs an access to a building, then the person must be member of a group which may access the building.

Moreover, we want to express that the arity of attribute *tel* is 1..n, and that *startdate* is always defined. Such arity constraints can be expressed in the graphical formalism, but they are only partially translated into the Z skeletons.

An interesting feature of RoZ is that it allows to express constraints at several levels, depending on their scope. For example, constraints 1 and 2 can be evaluated for a given instance of a class, they will be expressed in the Z schema corresponding to the intension of the class. In RoZ, such con-

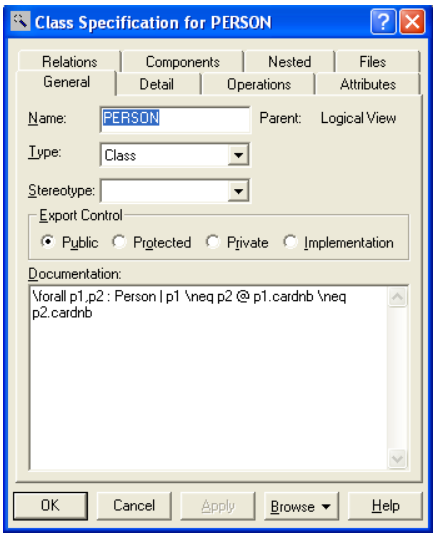

Figure 3. RoZ constraint for Person

straints are expressed in the documentation field of one of the attributes involved. For example Fig. 2 shows how constraint 1 (checksum of *cardnb*) is expressed in the documentation field of attribute *cardnb*. The constraint expresses that the last two digits of a card number are the six leading digits modulo 97.

RoZ automatically fills these annotations into the Z skeleton. Here is the resulting schema for *PERSON*:

PERSON

$$
\begin{array}{l}
cardnb : DIGIT8 \\
lastname : NAME \\
firstname : NAME \\
startdate : DATE \\
enddate : DATE \\
tel : \mathbb{F}\, TEL \\
\hline
(cardnb \operatorname{div} 100) \bmod 97 = cardnb \bmod 100 \\
tel \neq \{\} \\
startdate \neq undefineddate \\
(enddate \neq undefineddate) \Rightarrow (startdate \leq enddate)
\end{array}
$$

The schema includes constraint 1. It also expresses the arity constraints on attributes *tel* and *startdate*: *tel* is never empty and *startdate* is always defined, in other words it may not take the *undefineddate* value. The last line expresses constraint 2.

Fig. 3 shows how constraint 3, related to the keys of classes, is defined in the documentation fields of each class, here class *PERSON*. The constraint involves several instances of the class and must be defined at the level of the class extension. It expresses that two distinct instances of *PERSON* stored in the information system have different card numbers. The resulting Z schema is:

PersonExt

$$
\begin{array}{l}
Person : \mathbb{F}\, PERSON \\
\hline
\forall p1, p2 : Person \mid p1 \neq p2 \bullet p1.cardnb \neq p2.cardnb
\end{array}
$$

Similarly, *GroupExt* expresses that both *groupcode* and *groupname* can be used as keys for class *GROUP*.

GroupExt

$$
\begin{array}{l}
Group : \mathbb{F}\, GROUP \\
\hline
\forall g1, g2 : Group \mid g1 \neq g2 \bullet \\
\quad g1.groupcode \neq g2.groupcode \\
\quad \land\ g1.groupname \neq g2.groupname
\end{array}
$$

Constraint 4 requires information from the association between classes. It is expressed in the documentation field of *Members* and ends up in schema *MembersRel*.

MembersRel

$$
\begin{array}{l}
\ldots \\
\hline
\ldots \\
\forall p1, p2 : \operatorname{dom} GroupOfPerson \mid \\
\quad GroupOfPerson(p1) = GroupOfPerson(p2) \bullet \\
\qquad \forall t1 : p1.tel \bullet \forall t2 : p2.tel \bullet \\
\qquad\quad prefix(t1) = prefix(t2)
\end{array}
$$

The definition of *prefix* appears in the same file as type definitions and is expressed as follows:

$$
\begin{array}{l}
prefix : TEL \rightarrow TEL \\
\hline
\forall t : TEL \bullet prefix(t) = t \operatorname{div} 100
\end{array}
$$

Finally, constraint 5, which involves several classes and associations is expressed in the documentation field of the Rational Rose "Logical View" and completes the *GlobalView* schemas.

GlobalView

$$
\begin{array}{l}
MayAccessRel \\
AccessesOfBuildingRel \\
MembersRel \\
AccessesOfPersonRel \\
\hline
\forall a : Access \bullet \\
BuildingOfAccess(a) \in \\
BuildingOfGroup(GroupOfPerson(PersonOfAccess(a)))
\end{array}
$$

In summary, RoZ allows a smooth integration of Z annotations in the fields of the Rational Rose environment, and copies these in the related Z schemas. This brings several advantages. First, it improves the expressiveness of the graphical framework. Most contraints cannot be expressed graphically in UML. Typically they should be expressed in OCL. Compared to OCL, RoZ structures the constraints: in OCL, constraints may only be expressed in the context

of a given class. With RoZ, the context can be the intension or the extension of the class, but also an association or a view. From a methodological point of view, this structures the process of identification of constraints. From a documentation point of view, this provides an easier identification of the scope of a constraint. Finally, from a verification point of view, we will see that it allows to progressively take constraints into account and helps identify the operations which can be inconsistent with a constraint. Moreover, Z is supported by more tools, e.g. provers or animators, than OCL.

2.4. Operations

A UML class diagram only allows to specify the signature of the operations of a class. Their behaviour can be specified in OCL as pre- and post-conditions, or partially expressed through behavioural diagrams such as State-Transition or Sequence diagrams. RoZ allows to express their specification as Z annotations, in a similar way as constraints are included in the diagrams.

An additional feature of RoZ is the automatic production of the specification of basic operations. Basic operations are operations that appear systematically in class diagrams: they include operations to modify the value of each of the attributes of the class, operations to add or delete instances of the class, and operations to update the associations. Fig. 4 shows how these operations are automatically added to the class diagram by RoZ. They are then translated into Z. For example, operation *ChangeFirstname* modifies the *firstname* attribute of a given person, keeping other attributes unchanged.

PERSONChangeFirstname

$$\Delta PERSON$$
$$newfirstname? : NAME$$

$$firstname' = newfirstname?$$
$$cardnb' = cardnb \land lastname' = lastname$$
$$startdate' = startdate \land enddate' = enddate \land tel' = tel$$

This operation is an "intension operation", i.e. it is defined in the context of a schema which describes the intension of the class. Sect. 4.2 will discuss how this operation can be promoted at the level of the global view.

Other operations generated by RoZ affect the extension of classes. For example, the following operations add and remove instances of class *PERSON*.

AddPerson

$$\Delta PersonExt$$
$$person? : PERSON$$

$$Person' = Person \cup \{person?\}$$

RemovePerson

$$\Delta PersonExt$$
$$person? : PERSON$$

$$Person' = Person \setminus \{person?\}$$

Other operations modify the links of associations, such as *LinkPersonOfGroup* which defines a new member for a given group.

LinkPersonOfGroup

$$\Xi PersonExt;\ \Xi GroupExt$$
$$\Delta MembersRel$$
$$person? : PERSON$$
$$group? : GROUP$$

$$GroupOfPerson' = GroupOfPerson \oplus \{person? \mapsto group?\}$$

RoZ also generates an initialization schema (where all sets and associations are empty). This automatic generation of operation specifications is a major asset when combined with a Z animator, because it rapidly turns a data model, expressed as a class diagram, into a running prototype.

3. Animating it with Jaza

Jaza [14] is an animator that supports a wide subset of the Z language. It is based on a combination of proof (simplification, rewriting) and search (generate and test) techniques, which allow to handle some level of non-determinism in the specifications (provided the search space is not too large). In the case of non-deterministic specifications, it returns one of the valid solutions.

3.1. A small animation of the example

Here is a small sequence of animation steps for the access control example. We start with an initialisation of the data structures.

```
JAZA> do InitGlobalView

\lblot Access'==\{\}, AccessOfBuilding'==\{\},
       AccessOfPerson'==\{\}, Building'==\{\},
       BuildingOfAccess'==\{\}, BuildingOfGroup'==\{\},
       Group'==\{\}, GroupOfBuilding'==\{\},
       GroupOfPerson'==\{\}, Person'==\{\},
       PersonOfAccess'==\{\}, PersonOfGroup'==\{\} \rblot
```

Jaza returns each element of the state and its associated value. Fields are listed in alphabetical order. Here, every set and association is empty. Let us add a person to that state.

```
JAZA> ; AddPerson
Input person? = \lblot cardnb==1212,
lastname=="Smith",firstname=="John",startdate==1,
enddate==1000, tel ==\{0123456789\} \rblot
```

Jaza returns the following state.

```
\lblot Access'==\{\}, AccessOfBuilding'==\{\},
 AccessOfPerson'==\{\}, Building'==\{\},
 BuildingOfAccess'==\{\}, BuildingOfGroup'==\{\},
 Group'==\{\}, GroupOfBuilding'==\{\},
 GroupOfPerson'==\{\},
 Person'==\{\lblot cardnb==1212, enddate==1000,
            firstname=="John", lastname=="Smith",
            startdate==1, tel==\{123456789\} \rblot\},
 PersonOfAccess'==\{\}, PersonOfGroup'==\{\} \rblot
```

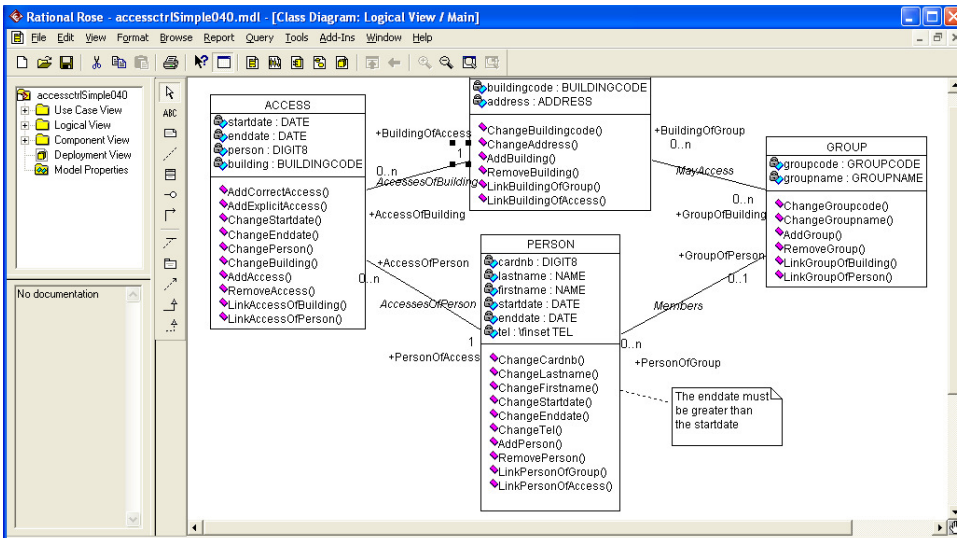

Figure 4. Generation of basic operations

Let us add a group (for brevity and clarity reasons, we omit some of the fields in the following transcipts).

```
JAZA> ; AddGroup
Input group? = \lblot groupcode=="G1",
               groupname=="GroupOne" \rblot

\lblot ...,
 Group'==\{\lblot groupcode=="G1",
           groupname=="GroupOne" \rblot\},
 GroupOfPerson'==\{\},
 Person'==\{\lblot cardnb==1212, enddate==1000,
            firstname=="John", lastname=="Smith",
            startdate==1,tel==\{123456789\} \rblot\},
 PersonOfGroup'==\{\} \rblot
```

We can now link the person to the group.

```
JAZA> ; LinkPersonOfGroup
Input person? = \lblot cardnb==1212,
                 lastname=="Smith",... \rblot
Input group? = \lblot groupcode=="G1",
               groupname=="GroupOne" \rblot

\lblot ...,
 Group'==\{\lblot groupcode=="G1",
           groupname=="GroupOne" \rblot\},
 GroupOfPerson'==
    \{(\lblot cardnb==1212, enddate==1000,
        firstname=="John", lastname=="Smith",
        startdate==1, tel==\{123456789\} \rblot,
        \lblot groupcode=="G1",
         groupname=="GroupOne" \rblot)\},
 Person'==\{\lblot cardnb==1212, enddate==1000,
            firstname=="John", lastname=="Smith",
            startdate==1, tel==\{123456789\} \rblot\},
 PersonOfGroup'==
    \{(\lblot groupcode=="G1",
          groupname=="GroupOne" \rblot,
       \{\lblot cardnb==1212, enddate==1000,
          firstname=="John", lastname=="Smith",
          startdate==1,tel==\{123456789\} \rblot\})\}
\rblot
```

Association links are represented as a pair of values. The translation rules of RoZ lead to a lot of redundancy. First, every role is stored, resulting in two redundant functions (here *GroupOfPerson* and *PersonOfGroup*). Also, using schema types in associations leads to repeat all attributes of an instance in all links where it is involved. This redundancy makes it more complex to modify the value of a class attribute. But we benefit from the fact that RoZ is able to generate appropriate promotion operations (see Sect. 4.2). Here, operation *PersonChangeFirstnameandRels* corresponds to the promoted version of *PERSONChangeFirstname*. We can use it to change the firstname of John Smith.

```
JAZA> ; PersonChangeFirstnameandRels
Input x? = \lblot cardnb==1212,lastname=="Smith",
            firstname=="John", ... \rblot
Input newfirstname? = "Philip"

\lblot ...,Group'==\{\lblot groupcode=="G1",
                    groupname=="GroupOne" \rblot\},
GroupOfPerson'==
       \{(\lblot cardnb==1212,enddate==1000,
           firstname=="Philip", lastname=="Smith",
           startdate==1, tel==\{123456789\} \rblot,
          \lblot groupcode=="G1",
           groupname=="GroupOne" \rblot)\},
 Person'==\{\lblot cardnb==1212, enddate==1000,
            firstname=="Philip", lastname=="Smith",
            startdate==1,tel==\{123456789\} \rblot\},
 PersonOfGroup'==
         \{(\lblot groupcode=="G1",
            groupname=="GroupOne" \rblot,
            \{\lblot cardnb==1212, enddate==1000,
               firstname=="Philip", lastname=="Smith",
               startdate==1,
               tel==\{123456789\} \rblot\})\}
\rblot
```

Our combination of RoZ and Jaza has conveniently handled the redundancy of the data structure and has replaced the firstname of John Smith everywhere it appears.

As class diagrams scale up, the states resulting from an animation quickly become unreadable. Therefore, we have developed a small visualizer, based on dotty [5], which turns

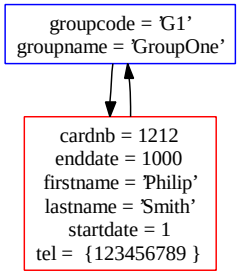

Figure 5. A small object diagram

a Jaza/RoZ state into a graphical representation. For example, Fig. 5 shows the current state of our animation. Fig. 6 shows a more complex state, with several persons, buildings and groups, and a single access; the corresponding textual representation of the state is 89 lines long.

3.2. Using non-promoted operations

One of the strengths, but also one of the difficulties of Z, is to clearly define the scope of its operations. For example, operation *AddPerson* of Sect. 2.4 has its scope restricted to set *Person*, defined in schema *PersonExt*. The schema clearly states how some fields of the state, here *Person*, must evolve to conform with the specification. What happens to fields such as *Group*, *Building* or *GroupOfPerson* when applying the operation *AddPerson* to a state that includes them? In early versions of Jaza, such fields did no longer appear in the resulting state, because they were out of the frame of *AddPerson* and hence there was no information on the existence of a $Group'$, $Building'$ or $GroupOfPerson'$ value. With this earlier version of Jaza, the *AddPerson* schema had to be completed as follows:

AddPersonPromoted

$\Delta PersonExt$
$\Delta GlobalView$
$\Xi MayAccessRel$; $\Xi AccessesOfBuildingRel$
$person? : PERSON$

$Person' = Person \cup \{person?\}$
$AccessOfPerson' = AccessOfPerson$
$GroupOfPerson' = GroupOfPerson$

The additional lines of the specification extend its scope ($\Delta GlobalView$) and express that everything else than *Person* does not change (using Ξ or additional equality constraints).

The latest versions of Jaza adopt a more flexible approach: everything outside the scope of an operation is kept unmodified in the final state. Operations such as *AddPerson* can be used without worrying about the rest of the state. Unfortunately, this approach may lead to inconsistent states, with respect to the *GlobalView* constraints. For example, let us call *RemovePerson* on the last state.

```
JAZA> ; RemovePerson
Input person? = \lblot cardnb==1212,
                lastname=="Smith",... \rblot


\lblot ...,
 Group'==\{\lblot groupcode=="G1",
           groupname=="GroupOne" \rblot\},
 GroupOfPerson'==
        \{(\lblot cardnb==1212, enddate==1000,
            firstname=="Philip", lastname=="Smith",
            startdate==1, tel==\{123456789\} \rblot,
           \lblot groupcode=="G1",
            groupname=="GroupOne" \rblot)\},
 Person'==\{\},
 PersonOfGroup'==
        \{(\lblot groupcode=="G1",
            groupname=="GroupOne" \rblot,
           \{\lblot cardnb==1212, enddate==1000,
              firstname=="Philip", lastname=="Smith",
              startdate==1,
              tel==\{123456789\} \rblot\})\}
\rblot
```

The resulting state has excluded Smith from set *Person*, but it remains in both *GroupOfPerson* and *PersonOfGroup*. The resulting state does not fulfill the constraints of *MembersRel* because the animator was not instructed to take these constraints into account. This has motivated the generation of the following operation:

CheckGlobalInvariant

$\Xi GlobalView$

CheckGlobalInvariant does nothing: it simply copies the initial state into a final state. But it is also instructed to compute a final state which fulfills all the constraints included recursively in *GlobalView*, which includes the constraints of *MembersRel*. Executing *CheckGlobalInvariant* on the current state yields the following result:

```
JAZA> ; CheckGlobalInvariant
No solutions
```

Systematic calls to *CheckGlobalInvariant* during an animation session allow to identify those operations calls which result in invariant violations, and avoids the systematic production of promoted versions of all operations.

3.3. Invalid execution sequences

Invalid preconditions Sect. 3.1 has shown a succesful animation. Executing *RemovePerson* led to an erroneous state. Some executions may also fail to return a solution.

```
JAZA> do InitGlobalView
\lblot ...,Person'==\{\}, ...\rblot
JAZA> ; AddPerson
Input person? =
  \lblot cardnb==1213, lastname=="Smith",
   firstname=="John", startdate==1,
   enddate==1000, tel ==\{0123456789\} \rblot
No solutions
```

Jaza provides an explanation facility (`why` command).

```
JAZA> why
\begin{schema}{AddPerson}
1  1213 \in 0 \upto 99999999
2  1213 \div 100 \mod 97 = 1213 \mod 100
3  ...
\end{schema}
The maximum number of true constraints was 1.
```

Here the failure results from the fact that constraint number 2 was false: 13 is not the checksum of 12! The input of *AddPerson*, which must be of type *PERSON*, does not verify one of the constraints of the schema.

Failures appear when Jaza finds an inconsistent set of constraints. Inconsistencies may result from a failed precondition: the user has chosen inadequate inputs, or has called the operation in an initial state which is not appropriate. Using the animator helps to identify more explicitly the pre-conditions which are implicitly stated by the specification. Animation may also be used as a way to test constraints and validate that they correspond to the intended meaning. Failed execution may also result from an inconsistent operation definition, i.e. an operation whose precondition is always false. For such operations, several animation sequences will reveal that the operation always fails, and the `why` construct may give hints on how to correct it.

Failed invariant In cases where the basic operation fails to preserve invariant properties, a new version of the operation must be written by the user. For example, *RemovePersonPromoted* makes the promotion of the operation at the level of the global view and explicitly states its precondition. Another possibility is to specify an operation that will delete the links where the person appears. But this is more tricky, since removing a person may also lead to remove the accesses he made.

RemovePersonPromoted

$\Delta PersonExt$
$\Delta GlobalView$
$\Xi MayAccessRel$; $\Xi AccessesOfBuildingRel$
$person? : PERSON$

$Person' = Person \setminus \{person?\}$
$person? \notin \mathrm{dom}\, AccessOfPerson$
$person? \notin \mathrm{dom}\, GroupOfPerson$
$AccessOfPerson' = AccessOfPerson$
$GroupOfPerson' = GroupOfPerson$

An attempt to *RemovePersonPromoted* does no longer lead to an inconsistent state, but returns `no solutions`.

```
JAZA> ; RemovePersonPromoted
Input person? = \lblot cardnb==1212, ... \rblot
No solutions
```

Non-determinism Jaza may refuse to animate non-deterministic operations, such as *GenTel*, when the search space is too large.

GenTel

$t! : TEL$

Its execution produces the following result.

```
JAZA> do GenTel
Problem: set is too big to enumerate
           (~ 10000000000 but current limit=1000)
0 \upto 9999999999
```

Since RoZ only produces deterministic operations, such situations may only arise when the model also includes user-defined operations.

4. Adapting RoZ to Jaza

4.1. Adapting the annotated class diagram

Adapting the types and definitions Some Z constructs are not supported by Jaza, this is the case of axiomatic definitions. In our example, *undefineddate* and *prefix* are defined axiomatically. These definitions had to be redefined as:

$undefineddate == 999999$
$prefix == \{t : TEL \bullet t \mapsto (t \operatorname{div} 100)\}$

This leads to choose a value for *undefineddate* and results in a slightly less abstract specification. Also, when non-deterministic operations over given sets are specified, it may be required to transform a given set into an enumerated set, in order to provide Jaza with a defined search space.

Arities in the diagram In our original UML model, the *Members* association mandated every person to be member of one and only one group, and each group to have at least one member. With such a model, it is mandatory to simultaneously create the first person and the first group. The corresponding operation is more difficult to specify and to implement. It does not correspond to the basic operations generated by the tool. Using the animator helps locate such difficulties in the diagram. It is then up to the analyst to decide whether he manually writes a complex operation, or he simply modifies the arities, as was done for *Members*.

Supported UML constructs RoZ does not support all constructs of the class diagram. For example, stereotypes will never be supported by the tool. Other constructs, such as agregation or composition, are partially translated: they are considered as associations, but additional checks or proof obligations are generated. Finally, some constructs, like inheritance, are translated by RoZ using non-deterministic constructs that are not executable by Jaza[3].

3 The translation is based on the disjunction of the schemas corresponding to subclasses. As a result, the contents of fields specific to a given class are not handled deterministically by the operations of the other subclasses.

4.2. Evolutions of the RoZ rules

Several rules have been added or modified to fit with Jaza.

Initialisation and CheckGlobalInvariant The original RoZ asked the user to provide explicitly an initialisation operation. Some rules have been added to automatically create an initialisation operation that creates a state whose fields are all empty sets. Still, this leaves the opportunity for the user to define other initialisation schemas if needed. Similarly, rules produce *CheckGlobalInvariant* which did not make sense before RoZ was used with Jaza.

Redundant definitions In Sect. 2.2, the translation of associations, like *Members*, describes how one of the roles, i.e. *PersonOfGroup*, can be constructed from the other one, i.e. *GroupOfPerson*. Jaza requires a more symetrical definition and new RoZ rules redundantly express how *GroupOfPerson* can be constructed from *PersonOfGroup*.

MembersRel

$$
\begin{array}{l}
PersonExt;\ GroupExt \\
PersonOfGroup : GROUP \pfun \mathbb{F}\, PERSON \\
GroupOfPerson : PERSON \pfun GROUP \\
\hline
\mathrm{dom}\, GroupOfPerson \subseteq Person \\
\mathrm{ran}\, GroupOfPerson \subseteq Group \\
PersonOfGroup = \{x : \mathrm{ran}\, GroupOfPerson \bullet \\
\qquad x \mapsto \{y : \mathrm{dom}\, GroupOfPerson \mid \\
\qquad\quad GroupOfPerson(y) = x \bullet y\}\} \\
GroupOfPerson = \bigcup\{x : \mathrm{dom}\, PersonOfGroup \bullet \\
\qquad \{y : PersonOfGroup(x) \bullet y \mapsto x\}\} \\
\ldots
\end{array}
$$

Promotion Operations, such as *PERSONChangeFirstname*, are defined on the intension of the class and do not refer to fields that appear in the global state during the animation. They refer to values that may be elements of the sets or in the domain or range of functions. Therefore promotion mechanisms are needed to use such operations at the level of extensions or associations.

All intension operations of class *PERSON* can be promoted at the level of the class by the following schema:

ChangePerson

$$
\begin{array}{l}
\Delta PersonExt;\ \Delta PERSON \\
x? : PERSON \\
\hline
x? \in Person \\
\theta PERSON = x? \\
Person' = Person \setminus \{x?\} \cup \{\theta PERSON'\}
\end{array}
$$

This operation modifies both an instance of *PERSON* and the extension of the class. It takes as input the instance whose attributes are modified and specifies how the extension of the class evolves. The schema must be combined with an intension operation to constraint the evolution of the fields of the instance. For example, here is the combination of *ChangePerson* with *PERSONChangeFirstname*.

$$
\begin{array}{l}
PersonChangeFirstname == \\
\quad (ChangePerson \land PERSONChangeFirstname) \\
\quad \backslash (cardnb, cardnb') \setminus (lastname, lastname') \\
\quad \backslash (firstname, firstname') \setminus (startdate, startdate') \\
\quad \backslash (enddate, enddate') \setminus (tel, tel')
\end{array}
$$

Basically, this is a conjunction of the operation definitions. Hiding operators are used to hide the fields of the instance. Otherwise, the resulting Jaza state would also refer to these fields, which don't make sense in the global view.

Similarly, the following schemas promote the operation at the level of associations. RoZ looks for all associations that involve the class of the modified instance and updates the functions whose domain is included in *PERSON* (here *GroupOfPerson* and *AccessOfPerson*). The remaining roles (*PersonOfGroup* and *PersonOfAccess*) are automatically updated by Jaza based on the constraints which describe how to construct these functions.

SubstitutePersonInRels

$$
\begin{array}{l}
\Delta MembersRel;\ \Delta AccessesOfPersonRel;\ \Delta PERSON \\
\Xi GroupExt;\ \Xi AccessExt \\
\hline
GroupOfPerson' = (\{x : Person \setminus \{\theta PERSON\} \bullet x \mapsto x\} \\
\quad \cup \{\theta PERSON' \mapsto \theta PERSON\}) \\
\qquad {}_{\S} GroupOfPerson \\
AccessOfPerson' = (\{x : Person \setminus \{\theta PERSON\} \bullet x \mapsto x\} \\
\quad \cup \{\theta PERSON' \mapsto \theta PERSON\}) \\
\qquad {}_{\S} AccessOfPerson
\end{array}
$$

$$
\begin{array}{l}
PersonChangeFirstnameandRels == \\
\quad (PERSONChangeFirstname \land \\
\qquad ChangePerson \land SubstitutePersonInRels) \\
\quad \backslash (cardnb, cardnb') \setminus (lastname, lastname') \\
\quad \backslash (firstname, firstname') \setminus (startdate, startdate') \\
\quad \backslash (enddate, enddate') \setminus (tel, tel')
\end{array}
$$

Producing these promotion schemas is a crucial element in our translation scheme from UML to Z, because we make a wide use of schema types to translate the intension of a class. Writing such complex schemas by hand is a clerical and error-prone process. It is therefore interesting to have a tool such as RoZ to produce these automatically.

5. Conclusion

This paper has presented a combination of RoZ with Jaza to animate annotated UML class diagrams. It has shown how an analyst can focus on the specification of the data structures of an information system, and of the associated constraints. RoZ automatically produces the specification

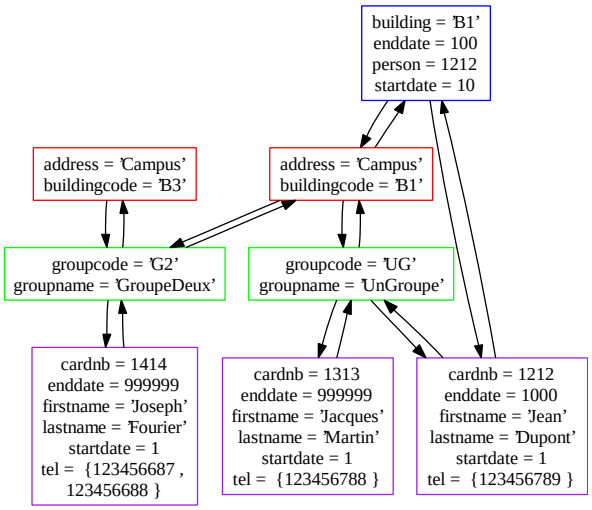

Figure 6. A larger object diagram

of basic operations associated to these data structures and translates the whole model into a Z specification. The model can then be exploited by standard Z tools such as provers or animators. Using an animator is a simple way to experiment with the model, to find errors in the specified constraints, or to identify operations which need more attention.

Limitations Today, the combination of RoZ and Jaza does not cover all constructs of the UML class diagram, and some of them (agregation/composition) are only partially supported. It is our intent to try to support some level of inheritance, which is an intrinsic construct in object-oriented modeling. Actually RoZ already translates inheritance into Z, but in a way that is difficult to animate with Jaza.

Schema types The translation rules of RoZ make a wide use of schema types to translate classes. This makes the translation and the animation more complex, but adequate promotion operations can be synthesized to support it suitably. Alternate translations exist, which ease the specification of operations and the translation of inheritance, but make invariant constraints more difficult to express. It was our intent in this work to evaluate to which extent RoZ and Jaza could support the translation on the basis of schema types, and the results are positive, for both RoZ and Jaza. On the one hand, Jaza has proven to be able to support a wide number of Z constructs, and to handle complex state structures such as the ones produced by RoZ. On the other hand, we were able to adapt the RoZ translation rules to support the promotion operations required by the schema types.

Future work Future work includes the support for some level of inheritance, it also involves the development of a better user interface for the use of Jaza in the context of RoZ. Transforming RoZ/Jaza states into dotty graphs is one element of such an environment. Additional facilities could include graphical support for operation calls, or links with tests cases expressed as UML sequence diagrams.

Acknowledgments This work would not have been possible without Mark Utting who helped me understand the subtleties of Jaza, and has performed some adaptations to the tool to make it more usable in the RoZ context. I also thank the LIFC laboratory of the Université de Franche-Comté which has hosted this collaboration. This work is supported by the EDEMOI project, sponsored by the ACI Sécurité Informatique of the French Ministry of Research.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley, 1999.

[2] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT dynamic model into Object-Z. In *ZUM'98*, LNCS 1493. Springer, 1998.

[3] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ : a Tool for Integrating UML and Z Specifications. In *CAiSE'2000*, LNCS 1789. Springer, 2000.

[4] R. France, J.-M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In *2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Chapman and Hall, 1997.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11), 2000.

[6] S.-K. Kim, D. Burger, and D. Carrington. An MDA Approach Towards Integrating Formal and Informal Modeling Languages. In *FM 2005*, LNCS 3582. Springer, 2005.

[7] R. Laleau and F. Polack. Coming and going from UML to B: A proposal to support traceability in rigorous IS development. In *ZB'2002*, LNCS 2272. Springer, 2002.

[8] K. Lano, H. Houghton, and P. Wheeler. Integrating Formal and Structured Methods in Object-Oriented System Development. In *Formal Methods and Object technology*, chapter 7. Springer, 1996.

[9] H. Ledang and J. Souquières. Contributions for modelling UML state-charts in B. In *IFM'2002: 3rd Int. Conf. on Integrated Formal Methods*, LNCS 2335. Springer, 2002.

[10] Y. Ledru. Identifying pre-conditions with the Z/EVES theorem prover. In *13th Int. Conf. on Automated Software Engineering*. IEEE CS Press, 1998.

[11] M. Saaltink. The Z/EVES system. In *ZUM'97*, LNCS 1212. Springer, 1997.

[12] M. Satpathy, R. Harrison, C. Snook, and M. Butler. A Comparative Study of Formal and Informal Specifications through an Industrial Case Study. In *FSCBS'01: IEEE/IFIP Joint Workshop on Formal Specification of Computer Based Systems*, 2001.

[13] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Trans. on Software Engineering and Methodology*, to appear.

[14] M. Utting. Data Structures for Z Testing Tools. In *FM-TOOLS 2000, TR 2000-07, Information Faculty, Univ. of Ulm*, 2000.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language (Second Edition) - Getting your models ready for MDA*. Addison-Wesley, 2003.