Identifying pre-conditions with the Z/EVES theorem prover

Yves Ledru
Laboratoire Logiciels Systèmes Réseaux
IMAG
B.P. 72, F-38402 St-Martin-d'Hères cedex, FRANCE
Yves.Ledru@imag.fr

Abstract

Starting from a graphical data model (a subset of the OMT object model), a skeleton of formal specification can be generated and completed to express several constraints and provide a precise formal data description. Then standard operations to modify instances of this data model can be systematically specified. Since these operations may invalidate the constraints, it is interesting to identify their preconditions.
In this paper, the Z-EVES theorem prover is used to calculate and try to simplify the pre-conditions of these operations. Then, the developer may identify a set of conditions and use the prover to verify that they logically imply the pre-condition.

Y. Ledru. Identifying pre-conditions with the Z/EVES theorem prover. In *Proceedings of the 13th International Conference on Automated Software Engineering*. pp. 32-41, IEEE Computer Society Press, Honolulu, 1998.

Identifying pre-conditions with the Z/EVES theorem prover

Yves Ledru
Laboratoire Logiciels Systèmes Réseaux
IMAG
B.P. 72, F-38402 St-Martin-d'Hères cedex, FRANCE
Yves.Ledru@imag.fr

Abstract

Starting from a graphical data model (a subset of the OMT object model), a skeleton of formal specification can be generated and completed to express several constraints and provide a precise formal data description. Then standard operations to modify instances of this data model can be systematically specified. Since these operations may invalidate the constraints, it is interesting to identify their pre-conditions. In this paper, the Z-EVES theorem prover is used to calculate and try to simplify the pre-conditions of these operations. Then, the developer may identify a set of conditions and use the prover to verify that they logically imply the pre-condition.

1. Introduction

In the recent years, several research works have been devoted to the integration of semi-formal specification languages (like OMT [19] or UML [10]) with formal specification techniques. On the one hand, semi-formal languages offer intuitive graphical notations that favour the communication within the specification team and with the customer. They put emphasis on the structure of the specification while details are often specified in natural language. The semantics of these notations are not always precise. This semantical imprecision and the informal character of natural language may lead to ambiguities and misunderstandings of such specifications. On the other hand, formal methods offer mathematical notations for the specification process. They have a precise semantics that removes ambiguities from specifications and offers a potential for reasoning and automation. Formal reasoning can be used to detect inconsistencies in specifications and to guarantee the existence of an implementation. Tools can take advantage of the formal semantics to support this reasoning process but also to help synthesize efficient implementations or test suites.

The integration of semi-formal and formal techniques aims at combining the advantages of both approaches, i.e. to have intuitive structured notations with a precise semantics that gets rid of ambiguities. Efforts have been made to integrate structured methods such as SSADM [17] or data flow diagrams[15, 18] with model-based languages like Z [22], B [1] or VDM [11]. More recently, these techniques have been used for object-oriented methods like Fusion [7], OMT [5, 14] or UML [8].

From our experience in this domain, we have learned that three benefits can be obtained from this integration:

Semantical awareness In OMT two alternate semantics can be given to the aggregation construct. In [4], precise specifications have been stated for both, it is then the analyst's responsibility to be aware of the ambiguity of the construct and to choose the appropriate meaning for his model.

Production of a formal specification Based on this precise semantics, a translation scheme can be defined to produce a skeleton of formal specification from the semi-formal diagrams. In [16], we have discussed such a translation schema from a subset of OMT into Z. That technique is successfully taught since several years as an introduction to formal methods for undergraduate. We are currently working on a more complete translation that takes into account the object oriented constructs of the method (aggregation, inheritance, encapsulation) [4] and maps them into Object-Z [3] constructs.

Natural language complements The translation of the semi-formal diagrams only provides a specification skeleton. This skeleton must be completed with information that is expressed in the informal natural language comments of the diagrams. Proposing a translation schema helps thus identify the nature of these comments. In the coming years, we expect to exploit this classification of comments to design a set of forms that will organize the natural language complements of OMT diagrams.

Once a formal model has been developed with these techniques, how can the resulting specification be exploited? On the one hand, one may expect that developing a precise specification leads the analysts to a finer under-

standing of the problem; this was demonstrated by several studies [6, 2]. Moreover, the specification provides an unambiguous reference document for the subsequent stages of the development. On the other hand, a formal specification can be the starting point of a reasoning process to either verify properties or to construct a program. Often, tool support is mandatory to guarantee the correctness of reasoning (e.g. critical applications), to help manage the size and complexity of the specification, or to cope with industry standards in productivity. Nowadays, tool support is still insufficient to match these requirements; formal methods are mainly used in critical applications where they are required by certification authorities, or in restricted application domains where sufficient domain knowledge has been developed.

This paper reports on an attempt to use a theorem prover [20] to exploit the formal specification in order to identify pre-conditions of systematically synthesized operation specifications.

2. Overview of the approach

The proposed development approach starts from a semi-formal specification of the data model of an information system (Fig. 1).

1. The graphical model is edited and completed with natural language annotations that state several constraints on the data (Sect. 3).
2. From this data model, commercial tools can generate a prototype application with standard operations to modify objects and their links.
3. The data model is also the starting point of a translation process which produces a formal specification skeleton (Sect. 4). Our process usually translates OMT diagrams into Object-Z. But here, in order to exploit a theorem prover, Z specifications were produced. Therefore, only a subset of OMT is considered with classes and relations.
4. The skeleton is completed with the formal expression of the natural language constraints.
5. Standard operation specifications are generated from the data model. They correspond to the operations of the prototype (Sect. 5).
6. Since these standard operations do not take into account the constraints, it is important to evaluate under which conditions these may be called safely. This is performed by identifying the pre-conditions with the help of the theorem prover (Sect. 6).
7. The pre-conditions provide information to complete the prototype and turn it into a robust and efficient implementation. This last step is discussed shortly in the conclusion of the paper (Sect. 7).

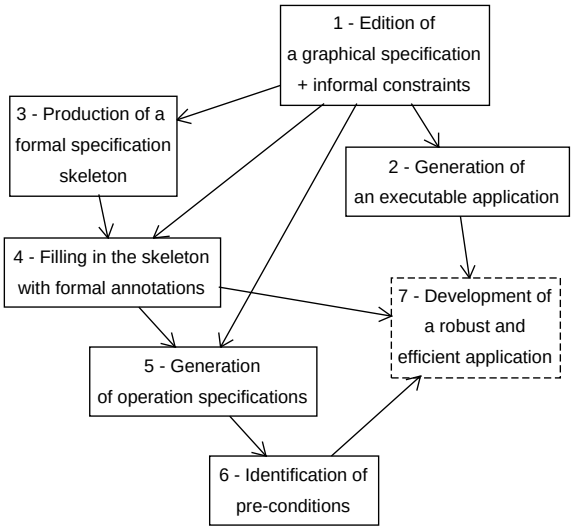

Figure 1. The proposed development process

3. Semi-formal specification

To illustrate the approach, we will use a variant of the data model of the access control system presented in [16]. This model (Fig. 2) features two classes: persons and groups. Each person has four attributes: his last and first names, the set of his telephone numbers, and the number of his magnetic access card. Each group is characterized by a name (e.g. "Staff") and a code (e.g. "ST"). The *Members* relation links each person to one and only one group.

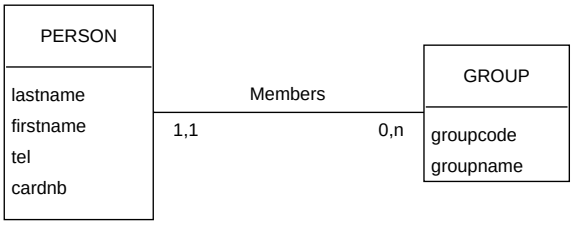

Figure 2. A simple data model

This model does not express the full specification of the application's data. Five constraints complement this diagram:

1. The card number has 8 digits. The first 6 digits are the actual number while the last 2 digits are a checksum (remainder of the division of the number by 97).
2. Every person has at least one telephone number.
3. The telephone numbers of the members of a given group have the same prefix.
4. The card number is a key for persons.
5. Both group code and name are keys for groups.

4. Translation of the data model into Z

4.1. Translating the classes

Systematic translation of this diagram into Z gives the following skeleton for PERSON. This skeleton must be completed with type information about the object's attributes and the constraints on each instance of the class.

PERSON

$lastname : \ldots$
$firstname : \ldots$
$tel : \ldots$
$cardnb : \ldots$

$\ldots$

The type for *cardnb* is a 8-digit number:

$$DIGIT8 == 0 \,..\, 99999999$$

Names and telephones are introduced as "given types", a more abstract type definition. Their specification is limited to the name of the type. More details about the type are left for subsequent refinements of the specification.

$$[NAME, TEL]$$

The *PERSON* schema may now be filled in:

PERSON

$lastname : NAME$
$firstname : NAME$
$tel : \mathbb{F}\, TEL$
$cardnb : DIGIT8$

$tel \neq \emptyset$
$(cardnb \text{ div } 100) \text{ mod } 97 = cardnb \text{ mod } 100$

The *tel* attribute is multi-valuated, i.e. it corresponds to a finite set of telephone numbers ($\mathbb{F}\, TEL$). Two constraints have been expressed on the elements of this schema: the first one expresses constraint 2 (the set of telephone numbers may not be empty), the second one is the checksum constraint on *cardnb* (constraint 1).

PERSON can now be used as a type for variables or constants. In [16], the following line introduces an undefined element as a constant of type PERSON.

$undefperson : PERSON$

Actually, in OMT or UML data models, the class corresponds to two notions: it introduces both the type of objects and an object "tank", i.e. the collection of objects of the class present in the information system. This notion, that we define as the "extension" of the class, must also be specified. The *PersonExt* schema introduces this extension (*Person* as a finite set of *PERSON*). Upper and lower cases are significant in Z, so we take advantage of this to distinguish the class type (*PERSON*) from its extension (*Person*).

PersonExt

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

$\forall p1, p2 : Person \mid p1 \neq p2 \bullet$
$\quad p1.cardnb \neq p2.cardnb$
$undefperson \notin Person$

The first constraint of this schema expresses that *cardnb* is the key of persons (constraint 4), i.e. two different persons have distinct card numbers; the second one expresses that the extension does not include the undefined element. This specification of *PersonExt* can be generated automatically from the data model as soon as *cardnb* has been identified as the key of the class.

The specification of the groups is produced similarly. Here the first constraint expresses that both *groupcode* and *groupname* are potential keys (constraint 5).

$$[GROUPCODE, GROUPNAME]$$

GROUP

$groupcode : GROUPCODE$
$groupname : GROUPNAME$

$undefgroup : GROUP$

GroupExt

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

$\forall g1, g2 : Group \mid g1 \neq g2 \bullet$
$\quad g1.groupcode \neq g2.groupcode$
$\quad \wedge\ g1.groupname \neq g2.groupname$
$undefgroup \notin Group$

4.2. Translating the relation

The relation between persons and groups is translated as a pair of functions: *GroupOfPerson* links a group to each person and *Members* gives the members of a given group. The first constraint on this schema expresses that the domain of *GroupOfPerson* is identical to set *Person*, i.e. every element of the extension is member of a group. The second constraint expresses that the range of this function is a subset of the extension of groups, i.e. some groups may have no members. The third constraint expresses *Members*

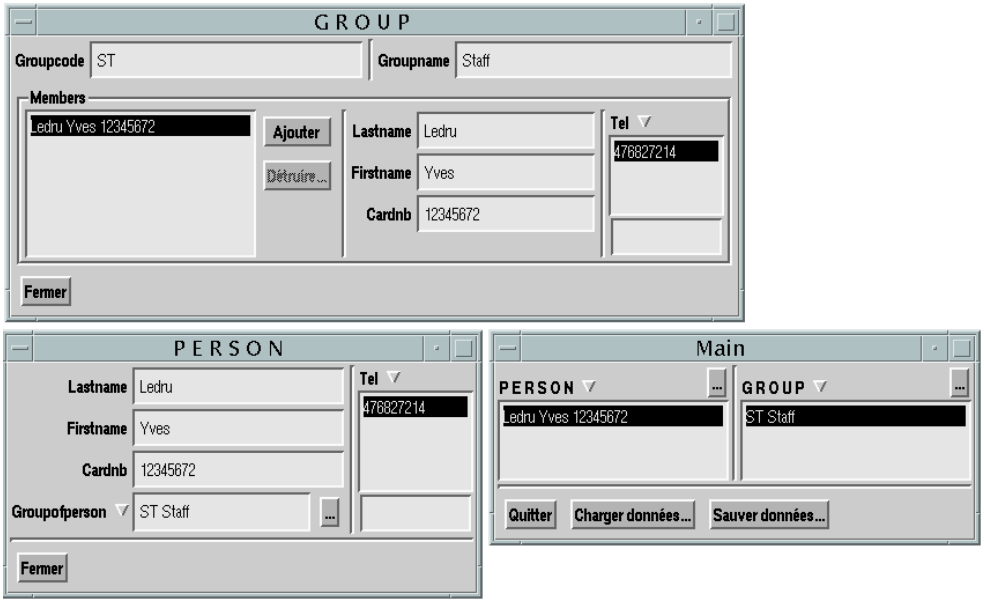

Figure 3. Application generated by the Delphia Object Modeler

as a set comprehension in terms of *GroupOfPerson*, i.e. the members of a group must be linked to that group. Actually, schema *PersonGroupRel*2 is a pure translation of the arities of the relation on the graphical model (1,1 to 0,n).

*PersonGroupRel*2

$$\begin{array}{l} \textit{PersonExt};\ \textit{GroupExt} \\ \textit{GroupOfPerson} : \textit{PERSON} \pfun \textit{GROUP} \\ \textit{Members} : \textit{GROUP} \pfun \mathbb{F}\, \textit{PERSON} \\ \hline \mathrm{dom}\, \textit{GroupOfPerson} = \textit{Person} \\ \mathrm{ran}\, \textit{GroupOfPerson} \subseteq \textit{Group} \\ \textit{Members} = \{g : \mathrm{ran}\, \textit{GroupOfPerson} \bullet \\ \qquad g \mapsto \{p : \mathrm{dom}\, \textit{GroupOfPerson} \mid \\ \qquad\quad \textit{GroupOfPerson}(p) = g \bullet p\}\} \end{array}$$

An additional constraint on this relation is that all members of a group have the same telephone prefix (constraint 3). It is introduced as a *prefix* function that returns the prefix of a given telephone number. This function is total (every telephone number has a prefix) and surjective (every prefix corresponds to at least one telephone number).

$[\textit{PREFIX}]$

$$\textit{prefix} : \textit{TEL} \tsurj \textit{PREFIX}$$

The constraint is added to the specification by first including the elements of the *PersonGroupRel*2 schema and then expressing the constraint.

PersonGroupRel

$$\begin{array}{l} \textit{PersonGroupRel}2 \\ \hline \forall\, p1, p2 : \textit{Person} \mid \\ \qquad \textit{GroupOfPerson}(p1) = \textit{GroupOfPerson}(p2) \bullet \\ \qquad\quad \forall\, t1 : p1.\textit{tel} \bullet \forall\, t2 : p2.\textit{tel} \bullet \\ \qquad\qquad \textit{prefix}(t1) = \textit{prefix}(t2) \end{array}$$

In summary, the specification is structured at 3 levels: class, extension, and relation. In more elaborate case studies, one or two additional levels appear in order to structure relations into views (4th level) and to have a global view of the whole data model (5th level). But these fourth and fifth levels do not introduce additional conceptual difficulties. It is also interesting to notice that constraints appear at each level: constraints 1 and 2 appear at the class level, 4 and 5 at the extension level, 3 at the relation level.

5. Generating operation specifications

Commercial tools (in particular 4th Generation tools) exploit the graphical data model to automatically synthesize a program. For example, Fig. 3 shows the user interface of an application automatically generated by one such tool (D•OM - Delphia Object Modeler [21]). The application allows to edit and modify groups and persons, and to add (*ajouter*) or remove (*détruire*) links between these classes. In D•OM, a trigger mechanism provides support for the constraints. These are expressed in an algorithmic way and can be checked before or after modifications of the data.

The following operations can be synthesized from the data model.

- at the class type level: modification of the attributes of an object;
- at the extension level: creation, deletion of objects of the extension;
- at the relation level: creation, deletion, modification of links between objects.

One may also define "find" operations that retrieve an instance on basis of a combination of its attributes. Such operations do not modify the data structure and will not be addressed in the rest of this paper. Only standard operations are generated from the data model. In many cases, they constitute a substantial part of the software. Application specific operations require additional human guidance. For example, they can be specified using other OMT models.

Operations on the class type. For each attribute of the abstract entity, a modification operation corresponds to a simple assignment to one of the attributes of the entity which does not affect the other attributes. For example, *ChangeCardnb* corresponds to the following instruction:

```
cardnb := newcardnb
```

It is expressed in Z as:

$$
\begin{array}{|l}
\hline
\textit{ChangeCardnb} \\
\Delta \textit{PERSON} \\
\textit{newcardnb}? : \textit{DIGIT}8 \\
\hline
\textit{lastname}' = \textit{lastname} \\
\textit{firstname}' = \textit{firstname} \\
\textit{tel}' = \textit{tel} \\
\textit{cardnb}' = \textit{newcardnb}? \\
\hline
\end{array}
$$

In this specification, *newcardnb*? is an input parameter. The last predicate constraints the new value of the *cardnb* attribute (denoted by a ') to be equal to this input parameter. The remaining predicates express that the other attributes keep their initial values. The first line of the Z schema (Δ*PERSON*) expresses that the effects of this operation are limited to changing an object of type *PERSON*. Other schemas are needed to "promote" this operation to the higher levels and to propagate the object modification to the extension of the class and the relations.

Operations on the class extension. The class extension is a set. Adding and removing an element are typical operations on sets. For example, the *AddPerson* operation receives a *PERSON* as input (*person*?) and its specification states that the new extension is the old one with this input

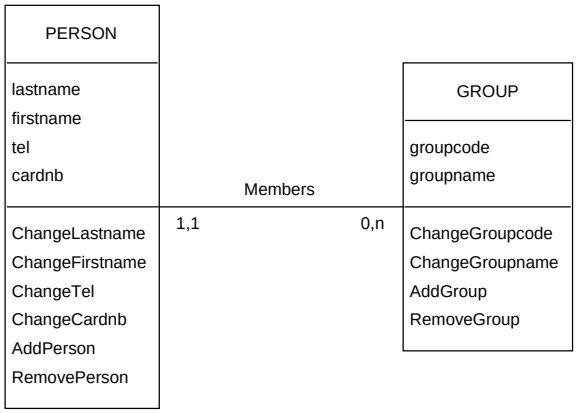

Figure 4. A data model enriched with methods

parameter. This operation is defined at the level of the extension and also needs to be "promoted" at the level of the relation.

$$
\begin{array}{|l}
\hline
\textit{AddPerson} \\
\Delta \textit{PersonExt} \\
\textit{person}? : \textit{PERSON} \\
\hline
\textit{Person}' = \textit{Person} \cup \{\textit{person}?\} \\
\hline
\end{array}
$$

Operations on Relations. Similarly, operations can be defined to create, remove or modify links between objects. The following specification can be synthesized to modify the link between a person and his group.

$$
\begin{array}{|l}
\hline
\textit{ChangeGroupofPerson} \\
\Delta \textit{PersonGroupRel} \\
\Xi \textit{PersonExt};\ \Xi \textit{GroupExt} \\
\textit{person}? : \textit{PERSON} \\
\textit{group}? : \textit{GROUP} \\
\hline
\textit{GroupOfPerson}' = \\
\quad \textit{GroupOfPerson} \oplus \{\textit{person}? \mapsto \textit{group}?\} \\
\hline
\end{array}
$$

A person and a group are passed as parameters, and the operation changes the entry of *person*? in *GroupOfPerson* to replace it with the new group. The second line of the specification expresses that this operation may access but does not modify (Ξ) the extensions of persons and groups.

Violating the constraints Operations *ChangeCardnb*, *AddPerson*, and *ChangeGroupofPerson* have been synthesized by assigning new values to the attributes or by adding or removing elements of a set. Applying this technique systematically to each attribute and each extension produces

the data model of Fig. 4 where methods have been added to each class.

But this synthesis process does not take into account the constraints expressed on the data model. Changing the card number may violate constraints 1 (checksum) and 4 (key of person); constraint 4 may also be violated by *AddPerson*; finally, constraint 3 (prefix) is likely to be violated by *ChangeGroupofPerson* (except if the new group has the same telephone prefix than the previous one).

6. Computing pre-conditions

One way to evaluate the impact of operations on constraints is to compute their pre-condition, i.e. the condition that must be satisfied initially to guarantee that the constraints are preserved at the end of the operation.

Identifying the pre-conditions of operations can help the developer in several ways:

- Some operations have a *true* pre-condition. They may be used without care.
- Some operations have a pre-condition that only refers to input parameters. A test that the input parameter meets the pre-condition provides a suitable guard for the operation. For example, *ChangeCardnb* verifies constraint 1 (checksum) if the *newcardnb*? parameter meets this constraint.
- Other operations require to test a combination of input parameters and state variables. It is the developer's responsibility to evaluate whether it is worth to access the state variables and perform the test or if some alternate strategy must be used. For example, *AddPerson* may violate the key constraint (4). To check this pre-condition, the new key can be compared to all keys of the extension. An alternate way to implement this is to add a variable which records the highest key number, and to impose that the new key is greater than this maximum. Of course, finding out this alternate implementation requires invention from the developer.
- Finally, some operations have an intricate pre-condition which is very often (or always) false. Identifying this pre-condition helps the developer figure out which operations don't make sense. These operations are not included in the final system or, if such operations are definitely needed, they may require to change the data model and its constraints. For example, the pre-condition of *ChangeGroupofPerson* requires that the person keeps his telephone prefix. Since, it is rather legitimate to allow group changes, this may mean that some constraints should be weakened, for example by allowing a person to have an empty set of telephone numbers.

In other words, identifying pre-conditions not only helps classify the operations generated automatically, but it helps the developer understand his application and the resulting data model. But the automatic generation of operations only provides basic blocks to construct an application. Exploiting these blocks is not always trivial and out of the scope of this study. For example, the basic operations can be used in a context specified by a dynamic diagram (Statechart) or a use case. In that case, the identified precondition will be used by the developer to check that it is satisfied in the calling context of the operation.

6.1. Pre-conditions in Z

Unlike VDM or B, Z does not distinguish a pre-condition in operation specifications. A single predicate links input and output parameters, initial and final states. But Z provides the pre *Op* operator to extract the weakest pre-condition of an operation [9]. For operation *Op* which modifies the schema *State* and has $i?$ and $o!$ as parameters, the pre-condition is defined as:

$$\exists\, State';\ o! : OUT \bullet Op$$

This operator quantifies the final state and the output parameters of operation *Op*. The resulting predicate is a condition on the initial state and the input parameters. When this condition is verified, the existence of a final state and output parameters that fulfill the specification is guaranteed.

Applying the pre operator to *ChangeCardnb* gives:

$$\begin{array}{l}\exists\, cardnb' : \mathbb{Z};\ firstname' : NAME;\\ \quad lastname' : NAME;\ tel' : \mathbb{P}\, TEL \bullet\\ \qquad ChangeCardnb\end{array}$$

By expanding the schema and removing the quantifier, this predicate can be simplified into the following condition:

$$\begin{array}{l} newcardnb? \in \mathbb{Z}\\ \land\ 0 \leq newcardnb? \land newcardnb? \leq 99999999\\ \land\ newcardnb? \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = newcardnb? \ \mathsf{mod}\ 100\\ \land\ firstname \in NAME \land lastname \in NAME\\ \land\ tel \in \mathbb{F}\, TEL\\ \land\ \neg\ tel = \{\}\\ \land\ 0 \leq cardnb \land cardnb \leq 99999999\\ \land\ cardnb \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = cardnb \ \mathsf{mod}\ 100\end{array}$$

In other words, the pre-condition expresses the type of the input parameters and the state variables, associated to constraints 1 and 2.

6.2. Identifying pre-conditions

One may assume that the initial state will satisfy the constraints stated in its schema, so the condition can be rewritten as:

$$\forall\, PERSON \bullet \mathrm{pre}\ ChangeCardnb$$

Moreover, it is also reasonable to assume that the input parameters will match their type:

$$\forall PERSON;\ newcardnb? : DIGIT8 \bullet \mathrm{pre}\ ChangeCardnb$$

This formula simplifies into:

$$\begin{array}{l} \quad lastname \in NAME \land firstname \in NAME \\ \quad \land\ tel \in \mathbb{F}\ TEL \\ \quad \land\ \neg\ tel = \{\} \\ \quad \land\ 0 \leq cardnb \land cardnb \leq 99999999 \\ \quad \land\ cardnb \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = cardnb \ \mathsf{mod}\ 100 \\ \quad \land\ 0 \leq newcardnb? \land newcardnb? \leq 99999999 \\ \Rightarrow \\ newcardnb? \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = newcardnb? \ \mathsf{mod}\ 100 \end{array}$$

which means that if the initial state satisfies types and constraints, and if the input parameters type-check, we only have to ensure the consequent of the implication:

$$newcardnb? \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = newcardnb? \ \mathsf{mod}\ 100$$

which is the actual pre-condition of *ChangeCardnb* at the class level. In summary, the pre-condition is obtained by simplification of the formula:

$$\forall State, i? : IN \bullet \mathrm{pre}\ Op$$

6.3. Automating this process

Computing and simplifying pre-conditions requires to manipulate long formulas. This is obviously a tedious and error-prone process. Therefore, tool support is mandatory if one wants to be more rigorous than informal reasoning. The major theorem provers for Z are ICL Proofpower [12], a commercial tool based on HOL, and Z/EVES [20]. Z/EVES was used to compute the above simplifications of pre-conditions.

Z/EVES provides support for general theorem proving for Z. It can be used to prove the consistency of specifications or refinements. It also supports schema manipulation and can be used to compute pre-conditions. It features both interactive and automatic mode, automatic mode being used for simple theorems.

Actually, the simplifications of the last section were performed with this prover. For example, the last pre-condition theorem is computed automatically using the following commands:

```
try \forall PERSON; newcardnb? : DIGIT8 @
      \pre ChangeCardnb;
prove by reduce;
```

The first command states the theorem to prove, the second one invokes an automatic strategy where schema expansion is performed.

6.4. Validating pre-conditions

Unfortunately, this process of pre-condition calculation and automatic simplification only works with simple examples. Often, the simplified formula is too long to be understood. This is quite normal: Z/EVES is aimed at showing that theorems are true, not at simplifying arbitrary formulas in a form suitable to be read. Moreover, we don't know of any Z tool that performs such simplifications.

The calculation and simplification process is not the only way to identify pre-conditions. One may also analyze the specification and report all conditions that apply to the relevant subset of the data model. But this second technique has also its limits: some of the collected constraints are not invalidated by the operation and are not involved in its pre-condition.

Therefore, we adopted a pragmatic approach:

- First, the developer figures out what the pre-condition is. Here, no systematic technique is proposed to carry out this activity. Several techniques may help him: calculating and simplifying (i.e. trying to use the approach of Sect. 6.3), browsing through the specification (as suggested in the beginning of this section), or also informal reasoning.
- Then, the theorem prover is used to formally verify that the proposed pre-condition is actually a pre-condition of the operation.

With this second approach, one tries to **prove** the following theorem:

$$\forall State, i? : IN \mid precondition(State, i?) \bullet \mathrm{pre}\ Op$$

The $precondition(State, i?)$ predicate is not necessarily the weakest pre-condition of the operation, but any condition that is strong enough to logically imply $\mathrm{pre}\ Op$.

For *ChangeCardnb*, this results into the following theorem, which is proved automatically by Z/EVES.

theorem ChangeCardnb_Pre
$$\begin{array}{l} \forall PERSON;\ newcardnb? : DIGIT8 \mid \\ \quad newcardnb? \ \mathsf{div}\ 100 \ \mathsf{mod}\ 97 = newcardnb? \ \mathsf{mod}\ 100 \bullet \\ \quad\quad \mathrm{pre}\ ChangeCardnb \end{array}$$

Using this process, the following pre-condition theorem is proposed for *AddPerson*. The proposed pre-condition states that the new element is not the undefined element, and that the key of the new element is not used in the extension.

theorem AddPerson_Pre
$$\begin{array}{l} \forall PersonExt;\ person? : PERSON \mid \\ \quad (person? \neq undefperson) \\ \quad \land\ (\forall p : Person \bullet person?.cardnb \neq p.cardnb) \bullet \\ \quad\quad \mathrm{pre}\ AddPerson \end{array}$$

This condition is actually stronger than the weakest precondition of the operation which also allows to add an element that was already in the set. Here the condition on the card number does no longer allow this possibility.

The first part of the pre-condition, which refers to the undefined element was not identified as a constraint in Sect. 3. It results from the systematic translation and from the automatic construction of some operations. Actually, an undefined element is needed for "find" operations that fail. Experimentally, this part of the pre-condition was found by trying and failing to prove a simpler pre-condition.

Proving *AddPerson_Pre* is no longer an automatic process. As stated earlier, the automatic strategies of Z/EVES are only successful for simple theorems. Here, the proof is more elaborate and involves about 15 proof steps. The proof is structured as a case analysis where each case is not longer than 5 or 6 steps. Figuring out this proof requires some training with the prover. Hopefully many proofs are very similar. For example, the proof of *AddGroup_pre* is the same proof as *AddPerson_pre*. One may thus expect that since operations are generated systematically, reuse of proofs or of proof structure is often possible.

A more elaborate pre-condition is obtained by promoting *ChangeCardnb* at the level of the class extension. This promoted operation (*ModifyCardnb*) has the following pre-condition theorem:

theorem ModifyCardnb_pre
$\forall$ *PersonExt*; *PERSON*; *person*? : *PERSON*;
newcardnb? : *DIGIT*8 |
θ*PERSON* = *person*?
$\wedge$ *newcardnb*? div 100 mod 97 = *newcardnb*? mod 100
$\wedge$ *newcardnb*? $\neq$ *undefperson.cardnb*
$\wedge$ ($\forall$ *p* : *Person* • *p.cardnb* $\neq$ *newcardnb*?)
• pre *ModifyCardnb*

ModifyCardnb has two input parameters: *newcardnb*? and *person*? which designates the object whose card number will be changed. The first line of the pre-condition is purely technical (θ*PERSON* = *person*?): it expresses that the person given as input is the one that is modified by the promoted operation. The last three lines of the pre-condition express constraints 1 and 4 and ensure that the resulting object is not *undefperson*. The last two lines are expressed in terms of the input parameter (*newcardnb*?). The condition *newcardnb*? $\neq$ *undefperson.cardnb* is too strong (nothing forbids that an element of *Person* has the same key as *undefperson*). A weaker precondition would combine *newcardnb*? with the attributes of *person*? to check that the resulting element is not the undefined one.

Proving this theorem also requires an interactive proof of about 20 steps. But the level of difficulty of the proof is similar to the one of *AddPerson* and once again, there are numerous similarities for all proofs corresponding to the promotion of attributes modifications.

7. Conclusion

7.1. Summary

A development process has been proposed which starts from a data model, produces a formal specification, generates operation code and specifications, and finally identifies the pre-conditions of operations. The translation process from the data model to Z has now been successfully experimented for several years. The generation of operation specifications is still very basic and can be improved to take into account the arity constraints of the relations as done in [13]. This would probably avoid to generate some useless operations that turn out to have a false pre-condition.

The main contribution of this paper is the definition of an identification process for pre-conditions. First, we expected that a general-purpose theorem prover could help us calculate and simplify these pre-conditions. This first attempt failed and the process was refined into a first phase where a pre-condition is "guessed" by the developer and a second phase where the theorem prover is used to validate this guess. "Guessing" precondition is not systematic and requires interaction with the developer.

Finally, it turns out that identifying the pre-condition is an iterative process. One starts with an empty pre-condition and tries prove the theorem $\forall$ *State*; *i*? : *IN* • pre *Op*. If the proof does not succeed, some insight is required from the developer. Usually, a careful examination of the "simplified" formula and some informal reasoning on the formal specification help identify all or parts of the pre-condition. These are injected in the theorem ($\forall$ *State*; *i*? : *IN* | *precondition*(*State*, *i*?) • pre *Op*). And the process is iterated until a complete pre-condition has been identified.

Validating pre-conditions, instead of calculating these, allows to identify a pre-condition that is not the weakest. It also allows to re-express the pre-condition in terms that are easier to check, e.g. expressing it only in terms of input parameters. But this flexibility has its limits, when the pre-condition is too different from the statement of the constraints, interactive theorem proving is required.

Fig. 5 gives the number of proof steps needed for the pre-condition theorems of the operations of Fig. 4. Level 1 operations which only modify the class are proved automatically; level 2 operations may be easy to prove (3 proof steps for *Remove* operations whose pre-conditions are *true*) or require case analysis. It turns out that the proofs of *AddPerson* and *AddGroup*, *RemovePerson* and *RemoveGroup* are respectively identical. We did not succeed to prove operations of level 3 (like *ChangeGroupofPerson*, or the promotion of level 2 operations). The problem lies with our small experience with the prover. Nevertheless, based on our first attempts, we are convinced that the approach is still valid and expect to find out other systematic proofs at this level.

Operation	# steps	Operation	# steps
ChangeLastname	1	ChangeGroupcode	1
ChangeFirstname	1	ChangeGroupname	1
ChangeTel	1		
ChangeCardnb	1		
AddPerson	15	AddGroup	15
RemovePerson	3	RemoveGroup	3
ModifyCardnb	20		

Figure 5. Proof effort for each operation

	Activity	Mode	Support tools
1	Edition of graphical specification	Interactive	Standard OMT tools, D•OM
2	Generation of executable application	Automatic	Commercial tools, D•OM
3	Production of a specification skeleton	Mainly automatic	tool to develop
4	Filling in with formal annotations	Interactive	
5	Generation of operation specifications	Automatic	tool to develop
6	Identification of preconditions	Interactive	Z/EVES prover

Figure 6. Tool support

7.2. The choice of a theorem prover

The process of identifying and validating pre-conditions is not specific of a given theorem prover. But several requirements exist for the prover:

Support for Z and the Z syntax. Using the Z syntax is mandatory because the iterative identification process involves both browsing the specification and proving. Also, it is not reasonable for the developer to learn languages for graphical specification, formal specification, proof, and programming. Having the same language for proofs and specification helps!

Support for automatic proofs. In the context of model-based specifications, theorems are often long and verbose but simple. This is the case here where pre-conditions involve several schema definitions. Unfolding these schemas in the original pre-condition theorems leads to long formulas. But these are often simple to demonstrate because the identified pre-condition that appears as an hypothesis is often the same or close to the statement of some corresponding constraint. Long but simple theorems often leads to long but simple proofs that can be handled automatically. Hence, automatic mode is required for the approach.

Z/EVES meets both requirements. ICL ProofPower may also meet these, but we don't have any experience with it. The limitations we have experimented with Z/EVES are its learning curve for demonstrating arbitrary theorems and performance when numerous schemas are involved by the theorem. As far as the learning curve is concerned, we expect that proofs or proof structures for standard operations can be reused (as shown by the proofs of level 2 operations). This should be exploited to guide the learning process and we intend to experiment it with undergraduate students. As far as performance is concerned, problems arise for operations at the third level and higher (relations). Solutions to this problem may lie in a better proof structure where automatic steps are performed after some preliminary work, or in the definition of theorems and rewrite rules that are specific to the approach. Also, experience with other methods (e.g. B) that involve theorem provers has shown that the way operations are stated may simplify their automatic proof. Further investigations should thus be performed on alternate expressions of the operation specifications.

7.3. Potential for tool support and automation

Fig. 6 summarizes the main phases of the development process, their automatic/interactive character and their tool support. The edition of the specification and generation of an executable application (1,2) can take advantage of commercial tools. The construction of the specification skeleton (3) and the generation of operation specifications (5) can be supported by automatic tools. Before undertaking the development of these tools, we prefer to investigate the identification of pre-conditions. It also turned out that this theorem proving activity helps validate and improve the rules we expect to use for automatic translation and generation. The production of a formal specification also involves the translation of constraints expressed in natural language (4). This translation is a human process that can benefit from the support of type-checking or proof tools. Finally, the identification of pre-conditions (6) is an interactive process.

In order to scale up from small examples to more significant applications, the process should be automated as much as possible. This would allow the developer to concentrate on insightful specification activities (1, 4). This is why the identification of pre-conditions must exploit the automatic mode of the prover and try to reuse proof structure.

7.4. Towards robust and efficient applications

Integrating formal methods into an industrial development process is not an easy task. It requires improvements in either productivity or quality, preferably both. In order to meet the productivity constraints, many phases of this approach offer a potential for automation. Still the translation of natural language into Z and the identification of pre-

conditions have an inherent interactive character. Therefore, the approach must show how it improves quality.

Actually, quality can be improved by taking into account the information gathered in the pre-condition identification process, to turn the executable prototype into a robust and efficient implementation. Robustness is achieved by calling operations within their pre-conditions. This guarantees that the properties of the data model are preserved. Efficiency is achieved because it is no longer needed to check that the data meet the constraints. For example, in the D•OM system, constraints are enforced by several demons that are triggered at each transaction. If the new state does not meet the constraints, the transaction is cancelled and the data are restored into their initial state. This way of enforcing constraints is acceptable for a prototype or a first version of the software but it requires additional computing resources. By enforcing pre-conditions, transactions are never cancelled and demons are no longer needed which improves the efficiency of the application.

This work is still far from industrial transfer. It requires to better integrate the object-oriented constructs in the specification as we try to do with OMT and Object-Z [4]. But we hope that it gives clues on how the additional precision of a formal specification can be exploited by adequate tools.

Acknowledgments Thanks to the members of the CuLARo Project for fruitful discussions. This work was partly supported by the EMERGENCE programme of the Région Rhône-Alpes.

References

[1] J. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, U.S. National Institute of Standards and Technology, 1993.

[3] R. Duke, P. Kin, G. Rose, and G. Smith. The Object-Z Specification Langage: Version 1. Technical Report 91–1, Departement of Computer Science, University of Que ensland, Australia, Software Verification Research Centre, April 1991.

[4] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Integrating OMT and Object-Z. In K. L. e. A. Evans, editor, *Proceedings of BCS FACS/EROS ROOM Workshop, technical report GR/ K67311-2, Dept. of Computing, Imperial College, 180 Queens Gate*, London, UK, June 1997.

[5] P. Facon, R. Laleau, and H. Nguyen. Mapping Object Diagrams into B Specifications. In A. Bryant and L. Semmens, editors, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, England, March 1996. Springer-Verlag.

[6] J. Fitzgerald, T. Brookes, M. Green, and P. Larsen. Formal and informal specifications of a secure system component : First results in a comparative study. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[7] R. France and J.-M. Bruel. Using Formal Techniques to Strengthen Informal Object-Oriented Modeling Techniques: The FuZE Experience. Technical report, Florida Atlantic University, Boca Raton, USA, Departement of Computer Science and Engineering, 1997.

[8] R. France, J.-M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 247–260, Canterbury, UK, 1997. Chapman and Hall, London.

[9] D. Gries. *The science of programming*. Springer-Verlag, 1981.

[10] T. U. Group. UML notation guide version 1.1. Technical report, Rational Software Corporation, September 1997. http://www.rational.com/uml/documentation.html.

[11] C. B. Jones. *Systematic Software Development Using VDM (Second Edition)*. Prentice-Hall, London, 1990.

[12] R. B. Jones. ICL ProofPower. *BCS-FACS FACTS*, Series III, 1(1):10–13, Winter 1992.

[13] R. Laleau and N. Hadj-Rabia. Génération automatique de spécifications VDM à partir d'un schéma conceptuel de données. In *Actes du XIIIe congrès INFORSID*, 1995.

[14] K. Lano and S. Goldsack. Intregrated Formal and Object-Oriented Methods: The VDM++ Approach. In A. Bryant and L. Semmens, editors, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, England, March 1996. Springer-Verlag.

[15] P. G. Larsen, N. Plat, and H. Toetenel. A Formal Semantics of Data Flow Diagrams. *Formal Aspects of Computing*, 6(6), Dec. 1994.

[16] Y. Ledru. Complementing semi-formal specifications with Z. In *Proceedings of the 11th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, September 1996.

[17] F. Polack, M. Whiston, and K. C. Mander. The SAZ project: Integrating SSADM and Z. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 541–557. Springer-Verlag, 1993.

[18] G. P. Randell. Translating data flow diagrams into Z (and vice versa). Report no. 90019, RSRE, Ministry of Defence, Malvern, Worcestershire, UK, Oct. 1990.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[20] M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88, Reading, UK, Apr. 1997. Springer-Verlag, Berlin.

[21] SLIGOS. *Delphia Object Modeler - Software production by object modeling*. SLIGOS, Agence Delphia, Seyssinet, France, 1995.

[22] J. Spivey. *The Z notation - A Reference Manual (Second Edition)*. Prentice Hall, 1992.