Expressing dynamic properties of static diagrams in Z

Yves Ledru
Laboratoire LSR/IMAG
BP 72
F-38402 Saint-Martin-d'Hères cedex
Yves.Ledru@imag.fr

Sophie Dupuy=Chessa
Laboratoire CLIPS/IMAG
BP 53
38041 Grenoble cedex 9
Sophie.Dupuy@imag.fr

Abstract

Au cours des dernières années, plusieurs équipes de recherche se sont intéressées à la traduction de langages graphiques tels que UML vers des méthodes formelles. Les langages à base de modèles comme Z et B sont particulièrement bien adaptés à la traduction des diagrammes statiques d'UML. Néanmoins, certaines constructions de ces diagrammes mettent en oeuvre des propriétés dynamiques. Par exemple, la composition en UML impose une dépendance existentielle. De telles propriétés ne peuvent pas être traduites par des invariants. Cet article étudie la traduction de ces propriétés en contraintes sur une paire d'états. En Z, de telles contraintes peuvent être intégrées à la spécification des opérations, ou exprimées par des obligations de preuve.

In the past years, several groups have addressed the translation of graphical formalisms such as UML into formal methods. Model-based approaches such as Z or B are particularly well-suited to translate static diagrams of UML into formal specifications. Still some constructs of these diagrams involve dynamic properties, e.g. composition in UML requires existential dependency. These properties cannot be translated into invariant properties. This paper explores the translation of these properties into constraints on a pair of states. In Z, such constraints can either be included into the specification of operations, or correspond to specific proof obligations for these operations.

1 Introduction

In the past years, several groups [FBLPS97, LB98, KC99, MS99] have addressed the translation of graphical formalisms such as UML [BRJ98] into formal methods. This approach brings several benefits:

- From the formal methods point of view, it enables the integration of these techniques into the actual practice of software engineers and facilitates the production of formal specifications.
- From the graphical notations point of view, it helps understanding the semantics of the notation. In particular, it points out those constructs that are ambiguous.

In a longer term, both communities may expect mutual benefits from the integration of formal methods tools into these graphical approaches.

In the recent years, our group has worked on the development of the RoZ tool[DLCP00a]. RoZ produces a Z specification from a UML class diagram annotated in Z. It also enables to synthesize the specification of basic operations associated to the diagram. Finally, it generates proof obligations

for each operation based on the computation of its precondition. These proof obligations can then be discharged using a standard Z prover (e.g. Z-Eves[Saa97]). The RoZ approach had the following interesting results.

- A specification expressed in RoZ has a precise semantics, which corresponds to the generated Z specification. One could probably argue that other semantics than ours could be associated to the diagram. Still, we believe that having a precise semantics is better than many or no semantics.
- RoZ enables to associate formal annotations as a complement to the UML specification. They roughly play the same role as OCL annotations and improve the expressiveness of the specification language. We chose Z instead of OCL [WK99] because a larger palette of tools is available for this language. This may evolve in the future and may lead to drop Z and replace it by OCL in our work.
- The design of RoZ relies on a careful study of the kinds of annotations that can be associated to a UML diagram. It led us propose a classification of these annotations [DFLCP00]. This classification can be used independently of the RoZ tool by a UML analyst. It helps identify and locate the relevant constraints for a given class diagram.
- Of course, the tool also generates a formal Z specification, and this specification can be used as input by several formal method tools. We have experienced the use of a theorem prover in conjunction with RoZ [Led98]. Current experiments explore the use of Jaza [Utt02] as an animation tool. Further work could experiment with test synthesis tools based on Z like [LPU02].

Model-based approaches such as Z or B are particularly well-suited to translate the static diagrams of UML into formal specifications. In RoZ, the translation of the class diagram produces a collection of variables and types associated to invariant properties. Still some constructs of these diagrams involve dynamic properties, e.g. composition in UML requires existential dependency. These properties can hardly be translated into invariant properties.

This paper addresses the translation of these properties in Z. Section 2 will first give some examples of dynamic properties expressed in static diagrams. Then section 3 will discuss the translation of these properties in Z. Finally, section 4 will discuss the potential integration of these techniques into the RoZ tool and draw the conclusions of this work.

2 Dynamic constructs in static diagrams

This section will introduce two constructs of static diagrams which involve dynamic properties:

- composition in UML,
- temporality in ZSP.

2.1 Composition in UML

In UML, the composition construct expresses that a class is “part-of” another class. It is a variant of the agregation construct. Fig. 1 gives the example of a conference center. The conference center is composed of several conference rooms. Unlike the agregation construct, which does not impose arity constraints between the whole and its parts, the composition relation imposes that the parts belong to

a single whole. Here each conference room belongs to one and only one conference center. The composition construct also imposes existential dependency between the whole and its parts: if the whole is deleted, its parts must be deleted. In the example of the conference center, if the conference center is suppressed from the information system, then its rooms must also be suppressed. The interested reader will find in [HSB99] a more detailed discussion of variants of the aggregation construct.

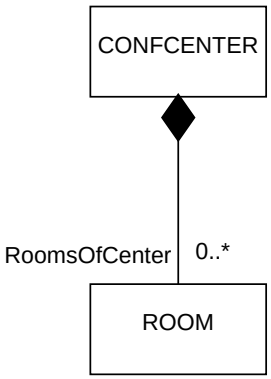

Figure 1: The UML specification of a conference center

This existential dependency is an example of dynamic property that cannot be expressed as invariant property. In the composition relation, the invariant property states that each room belongs to a single conference center. Unfortunately this is not sufficient to express the existential dependency. Fig. 2 illustrates this fact. A conference center is deleted but one of its conference rooms is not deleted and reaffected to another conference center. This figure, which is unlikely to happen in most information systems, complies to the arity constraints of the composition relation. Still, it does not correspond to the intended semantics of the composition construct. This demonstrates that a constraint on the arity of relations is not sufficient to capture the full semantics of composition.

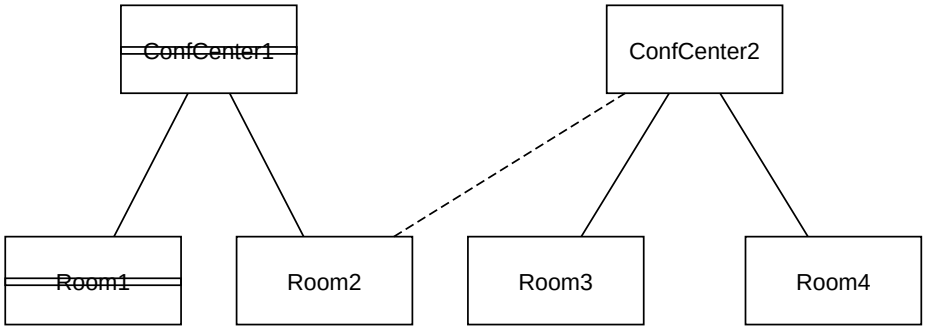

Figure 2: The deletion of a conference center

2.2 Temporalities in ZSP

ZSP [Abr77, CL94] is a graphical formalism based on an entity-relationship approach. J.R. Abrial initiated ZSP which corresponds to early work on Z performed before he went to Oxford. ZSP has mainly been designed to specify databases and information systems. It also includes a dynamic construct in its static diagram: the "temporality" construct. For clarity reasons, instead of using the ZSP syntax, we will use the classical UML notation, augmented with this temporality construct. A "[T]" sign will denote attributes and roles of relations subject to a temporality constraint.

"Temporality" expresses an eventual property: a function that is initially not defined for a given element eventually becomes defined. This property may apply to both attributes and relations in a class diagram. For example, let us consider the class of persons (Fig. 3). A person has several attributes, including two dates: the date of birth and the date of death. The date of death is initially undefined, but eventually it will get a value.

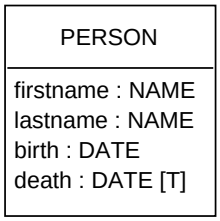

Figure 3: The class of PERSONs

If we model the date of death as a function between persons and dates, the temporality of this function should not be mistaken with a partial function that eventually becomes total. If this should be the case, then it would mean that, at some point in the future, all persons would be dead. Actually, the function remains partial (there are always some living persons) but the cardinality of its domain is monotonically increasing.

Temporalities also exist for relations. Fig. 4 shows the relation between two classes in the information system of an airline company:

- A FLIGHT is characterized by a flight number, its scheduled departure and arrival times, and the airports it links (origin, destination).
- An INSTANCE corresponds to a given flight at a particular date. It is characterized by the date of the flight.

A temporality exists on this relation: flight instances appear after the flight has been defined. Otherwise, it would require to define the flight and its first instance simultaneously. In database systems, creation or deletion of objects and relations often corresponds to transactions, and it is not recommended that transactions become too complex. The temporality expressed here means that the transaction which creates the flight will be performed before the transaction which creates the instance and links it to the flight. The temporality may also be the result of the way the company defines its commercial strategy: new flights could be planned in the information system several days before the first instance is decided.

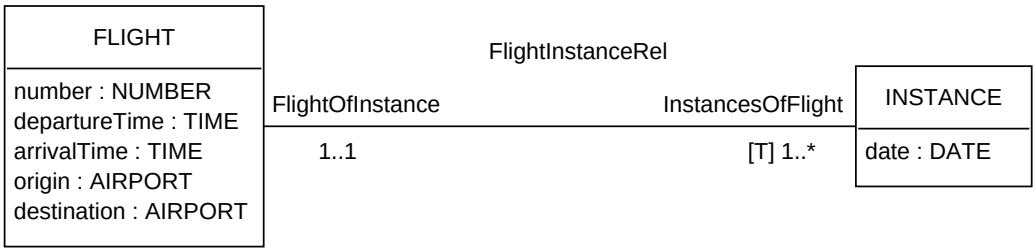

Figure 4: Flights and instances

In summary, temporalities are mainly motivated by two reasons:

- the decomposition of the transaction mechanism which initially breaks some invariant laws;
- the modelisation of laws of the specified system which require that some properties eventually hold.

Such temporalities cannot be expressed as invariant properties, because they express a constraint on the evolution of variables. They correspond in some way to "stable" properties in Unity logic [CM88], i.e. properties that remain invariant once established.

This section has shown that static diagrams may also include some properties which refer to the dynamic behaviour of objects. In notations such as UML, dynamic properties are usually expressed in dynamic diagrams, i.e. Statecharts. This corresponds to a separation of concerns, and it is wise not to overload static diagrams with non-static aspects. Still, UML itself includes dynamic constraints in the semantics of composition. The temporalities of ZSP could also be easily added to the UML class diagram, either as stereotypes on the relations, or as additional annotations to the elements of the diagram. The major reason why it is interesting to denote such properties on the static diagram is that it is much easier to add a simple element to the class diagram than to start drawing a dynamic diagram. Also, state diagrams are unable to express the eventual character of these properties (see Sect. 3).

From a more general point of view, many information systems have only a few invariant properties. Often the information system follows a life-cycle and properties remain invariant only during a portion of this life-cycle [PBV95]. It is therefore necessary to be able to identify the major phases of the life-cycle and for each phase the corresponding invariant properties. Here temporalities correspond to properties that are verified after an initialisation phase, while the existential dependency of the UML composition may correspond to properties of a closing phase.

3 Translating dynamic properties in Z

The dynamic properties such as temporality and existential dependency must be taken into account in the translation into formal specifications in order to propose a precise semantics of graphical notations. This work completes the translation of static properties supported by tools like RoZ.

3.1 Elements of the RoZ translation

The goal of this section is to summarize some principles of the RoZ tool. The starting point of RoZ is a UML class diagram annotated with Z constraints (Fig. 5). From this information, it produces a Z specification whose structure is provided by the graphical constructs. This structure is completed with type definitions and the Z annotations. RoZ also enables the synthesis of the specification of standard operations associated to the class diagram, and the generation of proof obligations based on pre-condition computation. The interested reader will find a more complete introduction to RoZ in [Dup99].

The RoZ tool translates a static diagram into a Z specification according to rules described in [DLCP00b, Dup00]. For example, class PERSON of Fig. 3 is translated into a pair of Z schemas: PERSON and PersonExt (see later).

$$
\begin{array}{|l}
\textit{PERSON} \\
\hline
\textit{firstname} : \textit{NAME} \\
\textit{lastname} : \textit{NAME} \\
\textit{birth} : \textit{DATE} \\
\textit{death} : \textit{DATE} \\
\hline
\end{array}
$$

PERSON is a schema type that defines the attributes of the class. The UML diagram must be associated to a definition of types NAME and DATE. For example, *NAME* can be introduced as a given set and *DATE* as a set of integers which count the number of days since a reference date.

$$[\textit{NAME}]$$

$$\textit{DATE} == \mathbb{Z}$$

The temporality on *death* requires that it may also be undefined. There are several ways to express undefinedness for attributes in RoZ. Let us use a simple and classical means: we choose a constant date in the future that is greater than all dates used in the system[1].

$$
\begin{array}{|l}
\textit{undefinedDate} : \mathbb{Z} \\
\hline
\textit{undefinedDate} = 9999999999 \\
\end{array}
$$

The RoZ tool also allows designers to associate constraints to attributes of the class, inside the standard UML environment [2] (Fig. 5). For example, the following constraint can be stated: death always happens after birth.

$$\textit{death} \geq \textit{birth}$$

This constraint also permits *death* to be undefined, because we chose *undefinedDate* to be large enough. It must also be noted that the birth date may never be undefined, which is expressed by the following constraint:

$$\textit{birth} \neq \textit{undefinedDate}$$

[1] Many programmers used to consider 9/9/99 as such a value for undefined dates. Let us use something like 9/9/999999 instead!

[2] Here, Rational Rose™

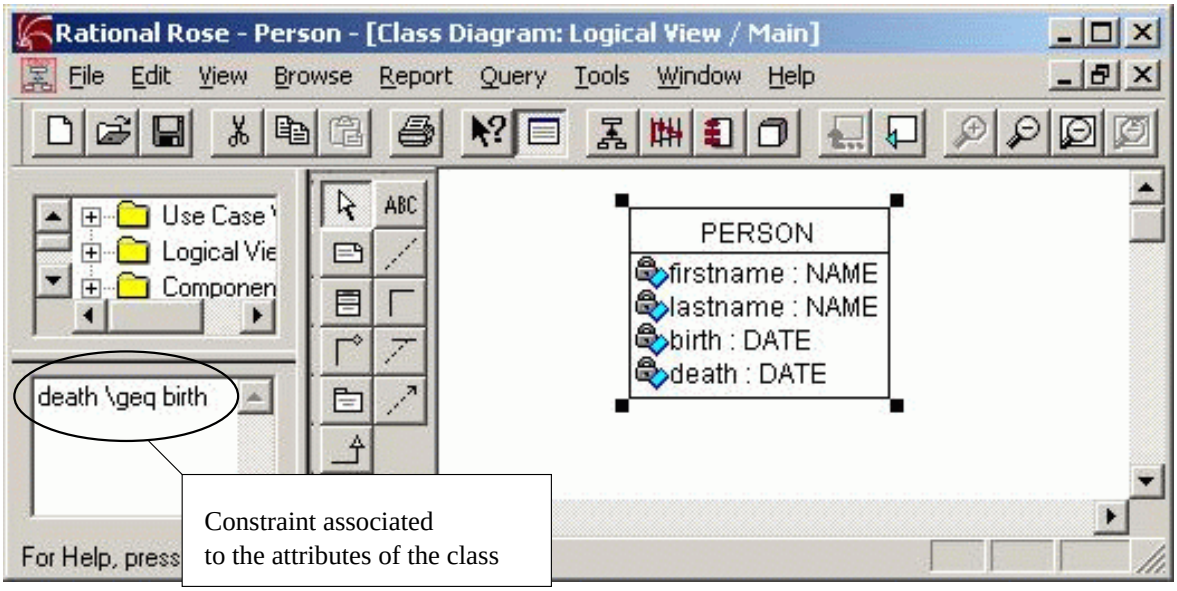

Figure 5: A Z annotation is associated to the attributes of the class

In RoZ, each class is translated into a pair of schemas. The first one is a schema type which gives the type of all elements of the class, the second one includes a variable which records the finite set of all intances of the class.

$$\begin{array}{|l} \textit{PersonExt} \\ \hline \textit{Person} : \mathbb{F}\, \textit{PERSON} \\ \hline \end{array}$$

With this translation, two living persons born on the same date and with the same first and last name are considered as the same element of the set. This probably suggests that we should further refine our model and have an additional attribute like the social security number.

Finally, let us have a look at the tranlation of relations. The relation between flights and instances of Fig. 4 is translated into:

$$\begin{array}{|l} \textit{FlightInstanceRel} \\ \hline \textit{InstanceExt};\ \textit{FlightExt} \\ \textit{InstancesOfFlight} : \textit{FLIGHT} \pfun \mathbb{F}\, \textit{INSTANCE} \\ \textit{FlightOfInstance} : \textit{INSTANCE} \pfun \textit{FLIGHT} \\ \hline \mathrm{dom}\, \textit{FlightOfInstance} = \textit{Instance} \\ \mathrm{ran}\, \textit{FlightOfInstance} \subseteq \textit{Flight} \\ \textit{InstancesOfFlight} = \{x : \mathrm{ran}\, \textit{FlightOfInstance} \bullet \\ \quad x \mapsto \{y : \mathrm{dom}\, \textit{FlightOfInstance} \mid \textit{FlightOfInstance}(y) = x \bullet y\}\} \\ \hline \end{array}$$

This Z schema declares a pair of functions, one for each role. *InstancesOfFlight* associates a set of instances to a flight, while *FlightOfInstance* links a single flight to each instance. The constraints

of the schema state that each instance has a flight, and that some flights may (temporarily) have no instance. These constraints involve the extensions of both classes and the schemas corresponding to these extensions (*InstanceExt* and *FlightExt*) are included in the schema. The final constraint expresses that both functions are linked and include the same information.

Composition is treated in RoZ as a special case for relations. So the relation between a conference center and conference rooms would be translated as:

ConfCenterComposition

$$
\begin{array}{l}
ConfcenterExt;\ RoomExt \\
RoomsOfCenter : CONFCENTER \pfun \mathbb{F}\, ROOM \\
CenterOfRooms : ROOM \pfun CONFCENTER \\
\hline
\mathrm{dom}\, CenterOfRooms = Room \\
\mathrm{ran}\, CenterOfRooms \subseteq Confcenter \\
RoomsOfCenter = \{x : \mathrm{ran}\, CenterOfRooms \bullet \\
\quad x \mapsto \{y : \mathrm{dom}\, CenterOfRooms \mid CenterOfRooms(y) = x \bullet y\}\}
\end{array}
$$

This schema follows the same translation rules as *FlightInstanceRel*. Actually, UML composition is not simply a UML relation with existential dependency, it also involves other properties like non-reflexivity (an element may not be a part of itself), anti-symetry (the whole is not part of its parts) and transitivity (subparts of an element are parts of this element). A more complete treatment of these properties can be found in [DLCP00b, Dup00].

At this stage, temporalities and composition have been translated into Z data structures associated to invariant properties. Temporalities require attributes like *death* to be potentially undefined, or functions like *InstancesOfFlight* to be partial, i.e. its arity is "0..*". This allows the corresponding data structure to be temporarily not defined. Composition requires *CenterOfRooms* to have a "1..1" arity, because a part must correspond to one and only one whole. But, as already stated in section 2, these static properties do not translate the full semantics of the constructs.

3.2 Translation of temporality on attributes

Temporality on attributes can be expressed in temporal logic as:

$$\Diamond(death \neq undefinedDate)$$

This formula means that eventually ($\Diamond$) a property will hold: *death* will be defined. Temporality also means that this property is stable:

$$death \neq undefinedDate \Rightarrow \Box(death \neq undefinedDate)$$

Once death is defined, it is always ($\Box$) defined.

The $\Diamond$ and $\Box$ operators of linear temporal logic state properties about infinite sequences of states. Z does not permit to state such properties about the future[3]. Z enables to express properties that take into account information about a single state. Such invariant properties are expected to hold in any possible state of the system. They thus constrain its possible evolutions. Still the dynamic properties

[3] Early versions of Object-Z [DKRS91] included the possibility to specify behaviours of classes as temporal logic formulae. Unfortunately, this feature has disappeared in subsequent versions of Object-Z.

that we want to express involve more than one state. There are two ways to express such dynamic properties in Z:

- The only way to translate these into invariant properties is to record information about the past of the system in some state variable and to constraint the evolution of this variable.
- Z enables to define properties on a pair of states in the specification of operations. This technique has already been used in the past. For example, in [Mor93] Morgan uses it to express the behaviour of a telephone network.

In this paper, we will mainly exploit this second possibility to express dynamic properties in UML or ZSP diagrams. Actually, the behaviour of *death* can be specified by the state machine of Fig. 6. This figure states that *death* is initially undefined, and stays undefined for a while. A transition may lead it to a defined state from where it will never exit. Actually, this state machine specifies that definedness is a stable property; it does not require *death* to reach this stable state. So the state machine does not express the full semantics of temporality. It must be noted that the other approach, which records information about the past, does not allow to express this property about the future too.

Since it is impossible to express this eventual character in Z, we will restrict our translation to constraints on the dynamic evolution from initial to final state.

Figure 6: State machine specification of *death*

Z allows us to state a property about initial and final states of transitions on this diagram:

$$death \neq undefinedDate \Rightarrow death' \neq undefinedDate$$

Actually, this property states that there does not exist a transition that goes from *defined* to *undefined*. The three transitions of Fig. 6 satisfy this property.

Also, we could state another dynamic property about *death*:

$$death \neq undefinedDate \Rightarrow death' = death$$

This property states that, once defined, the value of *death* is fixed. It expresses the fact that persons only die once.

3.3 Exploiting dynamic properties in Z specifications

In the previous section, we have seen that Z allows us to express dynamic properties on a pair of states. We propose two ways to exploit these properties. The first one is to include the property in any operation that accesses *PERSON* in read/write mode (Δ)[4]. For example, the following operation modifies the *death* attribute.

ModifyDeath

$$\Delta PERSON$$
$$d? : DATE$$

$$death' = d?$$
$$firstname = firstname' \land lastname = lastname' \land birth = birth'$$

We can define an additional schema:

DeathDynConstraints

$$\Delta PERSON$$

$$death \neq undefinedDate \Rightarrow death' \neq undefinedDate$$
$$death \neq undefinedDate \Rightarrow death' = death$$

and define a new version of *ModifyDeath*

$$ConstrainedModifyDeath \mathrel{\widehat=} ModifyDeath \land DeathDynConstraints$$

Of course, including this constraint in all schemas that access *death* requires a systematic procedure. Only a tool can guarantee that no operation has been forgotten.

Another way to take these constraints into account is to state a proof obligation which applies to all operations that access *death*.

$$\forall PERSON; PERSON'; d? : DATE \mid ModifyDeath \bullet DeathDynConstraints$$

Here, a slightly modified version of the operation satisfies the proof obligation:

*ModifyDeath*2

$$ModifyDeath$$

$$death = undefinedDate$$

In this modified version, a precondition has been added which guarantees that the operation is only executed when *death* is undefined. The above mentioned proof obligation has actually been discharged easily by the Z-Eves prover for *ModifyDeath*2.

In summary, this section shows that dynamic properties expressed on a pair of states can be exploited in the development of a Z specification. It requires the development of adequate tools which either complete existing specifications with these additional constraints or generate proof obligations.

[4] In recent evolutions of Z, this can be restricted to the operations that access the *death* variable in Δ mode.

In a context where implementations are developed by refinements of the specification, these two approaches (additional constraints and proof obligations) are somehow equivalent. If the addititional constraints approach is chosen, it boils down to consider more complex proofs during refinements. In a context where the specification only serves as a reference document and where no consistency proof is performed on the specification, the additional constraint approach would probably be adopted because it does not require a proof activity.

3.4 Temporalities on relations

Temporalities on relations can be treated in a similar way. Roughly speaking, the temporality on *InstancesOfFlight* requires that once a flight has corresponding instances, it keeps at least one instance. This can be expressed by the following property:

$$\mathrm{dom}\ \mathit{InstancesOfFlight} \subseteq \mathrm{dom}\ \mathit{InstancesOfFlight}'$$

The property expresses that the domain of the function may only increase. As soon as the function is defined for a given flight, it will remain defined. Moreover, the constraints of the *FlightInstanceRel* schema make it impossible for a flight to be associated to an empty set of instances, because the domain of *InstancesOfFlight* is the range of *FlightOfInstance*.

Actually, this property may be too strong because it does not allow to delete a flight from the information system. It can be weakened into:

$$(\mathrm{dom}\ \mathit{InstancesOfFlight} \cap \mathit{Flight}') \subseteq (\mathrm{dom}\ \mathit{InstancesOfFlight}' \cap \mathit{Flight}')$$

In this constraint, we only take into account the flights that remain after the operation. These flights are collected into the set *Flight′* which is the extension of flights. Since dom *InstancesOfFlight′* is necessarily included in *Flight′*, this can be simplified into:

$$(\mathrm{dom}\ \mathit{InstancesOfFlight} \cap \mathit{Flight}') \subseteq \mathrm{dom}\ \mathit{InstancesOfFlight}'$$

3.5 Translation of existential dependency

A similar approach can be adopted for the expression of existential dependency. Let us start with a simplified version of existential dependency where the association between a room and a center may never change: the room is defined and definitely associated to a center. If we don't allow the deletion of rooms or centers, we have the following property:

$$\mathit{CenterOfRooms} \subseteq \mathit{CenterOfRooms}'$$

This property expresses that the only evolution of the function is to add new associations between rooms and centers. Let us now express that some deletions may occur:

$$(\mathrm{dom}\ \mathit{CenterOfRooms}') \lhd \mathit{CenterOfRooms} \subseteq \mathit{CenterOfRooms}'$$

Here, $(\mathrm{dom}\ \mathit{CenterOfRooms}') \lhd \mathit{CenterOfRooms}$ is the initial function whose domain is restricted to the final domain, i.e. the remaining rooms in the final state. The property requires inclusion of the initial and final functions for the final domain. It allows to delete rooms, but also conference centers. If a conference center is deleted, then the corresponding rooms may not remain in the domain of the function, because the constraint would require these rooms to be associated to the previous conference center.

Still, this constraint does never allow to modify the assignment of rooms to conference centers. The semantics of composition allow such a change of parts when the whole is not deleted. The following property is weaker and expresses existential dependency:

$$\mathrm{dom}(\mathit{CenterOfRooms} \rhd \mathrm{ran}\ \mathit{CenterOfRooms}') \cap \mathrm{dom}\ \mathit{CenterOfRooms}' = \emptyset$$

Here, $\mathit{CenterOfRooms} \rhd \mathrm{ran}\ \mathit{CenterOfRooms}'$ is the initial function where the associations to remaining conference centers have disappeared. Taking the domain of this function returns the set of rooms whose conference center has been deleted. The property expresses that these rooms no longer appear in the domain of the final function. In other words, all rooms associated to deleted conference centers have been suppressed.

Once again, we see that it is possible to express interesting dynamic properties by constraining a pair of states. These constraints can now be included in the specification of all operations that modify the composition relation, or can be used in corresponding proof obligations.

4 Conclusion

This paper has shown that the semantics of static diagrams, such as the UML class diagram, may involve some dynamic properties. Languages like Z which are mainly aimed at the expression of invariant properties still enable to translate part of the semantics of these constructs. The basic idea is to capture dynamic behaviours by constraints on pairs of states.

These dynamic constraints are associated to variables of the specification (attributes, functions,...). They can be included in the Z specification either as additional predicates in the definition of operations or as proof obligations on the operations. A similar approach has been adopted by Habrias and Griech who translate dynamic constraints in B as a combination of constraints and proof obligations [HG97].

In order to ensure that the constraints are taken into account by all operations, it is necessary to be very systematic. The systematic inclusion of constraints in operation specifications or the generation of the proof obligations should better be automatized by a tool. RoZ can be extended to become one of these tools.

This paper has made the deliberate choice to stick with Z as a target language because it provides satisfactory answers to a majority of concerns of the RoZ project. One may also reconsider the choice of the target language and chose a language which better integrates dynamic properties of operations with the description of complex data structures, like the combination of TLA with Z [Lam94b, Lam94a], or the evolution of the B method to dynamic properties [AM98].

Eventualities The translation of temporalities into constraints on pairs of states does not carry the full semantics of the construct. Fig. 6 has shown that these constraints correspond to the specification of a state machine but do not express the eventual character associated to the temporality contruct. Actually Z does not permit to specify eventualities, because it can not express properties on the future. Still, based on the temporalities expressed in the diagram, it may be wise to generate the specification of some observation operations. For example, the following operation returns the set of flights which have not yet been associated to a first instance.

SearchFlightsWithoutInstances

Ξ*FlightInstanceRel*

$s? : \mathbb{F}\, FLIGHT$

$s? = Flight \setminus \mathrm{dom}\, InstancesOfFlight$

The operation simply returns the elements of the extension of flights which are not in the domain of the function. Such an operation may be very useful for the administrator of the information system, because it points out the set of flights which don't satisfy the dynamic constraint yet. The administrator can then take the necessary actions, if needed.

Variants of dynamic properties Temporalities and existential dependency are two examples of dynamic constraints. In ZSP and UML, they correspond to the semantics of graphical elements of the static diagrams. We have also seen that several dynamic properties could be considered. Existential dependency can be translated in two different ways. The weakest definition simply requires that parts of a deleted whole are deleted also; the strongest definition further constrains parts to be associated to a unique whole during their lifetime. Choosing the right property helps identifying the precise nature of the composition association in the context of a given modelisation. Such an integration of formal methods into graphical formalisms improves the semantical awareness of the analyst.

One may also wish to generalize this notion and permit to express dynamic properties which do not necessarily correspond to graphical elements. This would lead to associate arbitrary dynamic properties to variables or schemas of the specification.

Still, one must take care that the properties expressed on a pair of initial and final states are not necessarily preserved by consecutive applications of operations. If a constraint holds between instants 1 and 2, and between instants 2 and 3, it does not necessarily hold between 1 and 3. This is the case for existential dependency. Let us consider the example of Fig. 2. A first operation can delete *ConfCenter*1, *Room*1, and *Room*2, satisfying the existential dependency constraint. Then a second operation can recreate *Room*2 and link it to *ConfCenter*2. Individual operations satisfy the constraint, but not their composition. Other constraints stated in this paper feature this transitivity property. For example, the dynamic constraints expressed on the *death* attribute are preserved by consecutive applications of operations. For each dynamic constraint *D* in a given specification, the analyst should check whether it satisfies the transitivity property or not. Once again, he should take benefit of tools that would systematically generate a proof obligation to show this transitivity, and possibly submit it to a theorem prover.

In summary, dynamic constraints offer an interesting specification construct in the context of model-based methods such as Z or B. But they require the help of tools to help understand them precisely and to support their systematic and rigorous use.

Acknowledgments We want to thank Dominique Rieu and Jean-Pierre Giraudin for useful discussions about the dynamic constructs of static diagrams, and Monique Chabre-Peccoud for her collaboration in the RoZ effort and her support on information systems matters.

References

[Abr77] J.R. Abrial. Manuel du langage Z (Z/13). Technical report, Electricité De France, 1977.

[AM98] J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 : Recent Advances in the Development and Use of the B method*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[BRJ98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison Wesley, 1998.

[CL94] Y. Chiaramella and Y. Ledru. Modélisation ZSP (in French). Technical report, Université Joseph Fourier - UFR Informatique et Mathématiques Appliquées, 1994.

[CM88] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[DFLCP00] S. Dupuy, J. Celsio Freire, Y. Ledru, and M. Chabre-Peccoud. Formal and informal specifications: a proposal for a coupling. In *13th International Conference Software & Systems Engineering and their Applications-ICSSEA'2000*, Paris, France, 2000.

[DKRS91] 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 Queensland, Australia, Software Verification Research Centre, April 1991.

[DLCP00a] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ : a Tool for Integrating UML and Z Specifications. In *12th Conference on Advanced information Systems Engineering-CAiSE'2000*, volume 1789 of *Lecture Notes in Computer Science*, Stockholm, Suède, 2000. Springer-Verlag.

[DLCP00b] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z. *L'Objet, numéro thématique Approches formelles à objets*, 6(1), 2000.

[Dup99] S. Dupuy. RoZ version 0.3 : an environment for the integration of UML and Z. http://www-lsr.imag.fr/Les.Groupes/PFL/RoZ/index.html, 1999.

[Dup00] S. Dupuy. *Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information*. PhD thesis, Université Joseph Fourier, 2000.

[FBLPS97] 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.

[HG97] H. Habrias and B. Griech. Formal Specification of Dynamic Constraints with the B method. In Michael G. Hinchey and Shaoying Liu, editors, *Proc. of the 1st IEEE Int. Conf. on Formal Engineering Methods*, Hiroshima, Japan, 1997. IEEE Computer Society Press.

[HSB99] B. Henderson-Sellers and F. Barbier. A survey of the UML's aggregation and composition relationships. *L'Objet*, 5(3-4):339–366, 1999.

[KC99] S.-K. Kim and D. Carrington. Formalizing the UML Class Diagram Using Object-Z. In *2nd International Conference on the Unified Modeling Language - "UML"'99*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, USA, Octobre 1999. Springer.

[Lam94a] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam94b] L. Lamport. TLZ. In *Proceedings of the 8th Z Users Meeting*. Springer-Verlag, 1994.

[LB98] K. Lano and J. Bicarregui. Semantics and Transformations for UML Models. In J. Bezivin and P-.A. Muller, editors, *UML'98 - Beyond the Notation*, Lecture Notes in Computer Science, Mulhouse, France, Juin 1998. Springer.

[Led98] Y. Ledru. Identifying pre-conditions with the z/eves theorem prover. In *Proceedings of the 13th International Conference on Automated Software Engineering*. IEEE Computer Society Press, october 1998.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, July 22-24 2002.

[Mor93] C. Morgan. *Telephone network.* In *Specification Case Studies*, chapter 3. Prentice Hall, 1993.

[MS99] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems - FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 875–896, Toulouse, France, 1999. Springer-Verlag.

[PBV95] H. Habrias P. Bernard and A. Vailly. Diachronie et synchronie en spécification, une illustration en Z. In *Actes du XIIIe congrès INFORSID*, 1995.

[Saa97] 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, April 1997. Springer-Verlag, Berlin.

[Utt02] M. Utting. The Jaza animator. http://www.cs.waikato.ac.nz/ marku/jaza/, 2002.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, Reading, Mass., 1999.