Modular UML Semantics: Interpretations in Z Based on Templates and Generics

Nuno Amálio, Susan Stepney, and Fiona Polack

Dept. of Computer Science, University of York, YO10 5DD, UK
{namalio,susan,fiona}@cs.york.ac.uk

Abstract

Component systems comprise components linked in various ways. We illustrate an approach to expressing and selecting appropriate semantics for components, using as a starting point UML class diagrams. Like most diagrammatic notations, UML does not have a fixed interpretation. We present a meta-modular framework for the combined use of UML and Z, based on two levels. At the meta-level, we express modular semantic interpretation of UML diagrams by using templates and generics. At the instantiation-level, UML models are translated into Z specifications by instantiating the corresponding meta-level semantic interpretations. This allows the definition of semantic interpretations that are precise and unambiguous in a modular fashion, allowing tailoring of semantic interpretations by plugging-in or plugging-out modules representing some semantic aspect, and enhancing the readability, conciseness and abstraction of the resulting Z specification.
Keywords: Z, UML, modular semantics, translation, formal interpretation.

1 Introduction

Component systems comprise components linked in various ways. Important issues in the construction of components and component-based systems include the expression of the abstract purpose of the components and of the links among components. To explore component and linkage semantics, we choose to base our research on a well-known graphical notation for modelling components (here object-oriented (OO) classes) and links, namely UML. We illustrate an approach to expressing appropriate semantics for components by expressing different semantic interpretations of the OO components in Z, and constructing a formal model of the system by selecting the required semantic interpretation of each component and linkage.

The Unified Modelling Language (UML) [1, 2] is a set of notations for describing a system from various views. These notations are recognised as being intuitive and the result of modelling experience. However, UML, and other component description languages such as those used for architecture description (ADLs), suffers from the usual problems of diagrammatic-based notations.

- Diagrammatic notations have weak semantics. There is substantial disagreement on the semantics of some key concepts (for example, associations, aggregation and specialisation in UML [3]).
- Diagrammatic models are not amenable to formal reasoning techniques, as the notations do not have a formal semantics.
- Diagrammatic models do not have a fixed interpretation. Developers tailor the interpretation of diagrams to the problem at hand informally, tacitly and sometimes unconsciously. This constitutes a source of confusion. It is very likely that different developers interpret the same model in different ways.

There have been many attempts to overcome these limitations. One approach consists of giving a formal semantics to diagrammatic notations, where the aim is the sole use of diagrams with a sound semantics. Another approach consists of combining diagrammatic notations with formal specification languages (FSLs), where the aim is to use both notations, and to explore the formal reasoning capabilities of the FSL. This is the approach explored here.

Combinations are achieved by representing the modelling concepts of the diagrams in the formal model, thus providing a semantics for the diagram concepts. The graphical and textual representations can then be seen as alternative representations of the same thing.

There have been combinations of UML with B [4], Z [5, 6, 7], and Object-Z [8, 7]. These approaches give a unique semantics to the UML diagrams. However, we wish to accommodate semantic variability and tailoring semantics to different problem domains.

In this paper, we present a meta-modular framework for the combined use of UML and the FSL Z [9]. This framework is based on two levels: the *meta-level* and the *instantiation-level*. At the meta-level, we express semantic interpretations of UML diagrams. At the instantiation-level, UML models are translated into Z specifications by instantiating the corresponding meta-level semantic interpretations. Our approach is modular because at both levels we represent concepts as a collection of pieces, where each piece represents a particular semantic aspect. At the meta-level these pieces are Z generics and our templates. At the instantiation level these pieces are mostly Z schemas, but they may include other Z constructs. Semantic interpretations can be tailored at the meta-level by plugging-in and plugging-out pieces representing some semantic aspect.

Modularity is a key characteristic of our approach. It enhances the readability and abstraction

of the resulting Z specifications, and makes the whole framework very flexible, allowing easy tailoring of semantic definitions to accommodate different interpretations.

Templates and Z generics are parametric language constructs, that allow us to factor out commonly used structures, containing variations.

Z generics are provided by the language itself. They allow parameters to be associated with certain Z constructs, namely abbreviation definitions, axiomatic definitions and schemas. A parameter represents a set and set only.

We introduce templates to give us more flexibility than pure Z generic definitions allow. We use templates to parameterise all kinds of Z constructs with names, declarations, types and predicates. We also use templates to capture optionality. When supplied with parameter values, a template results in a well-formed Z construct.

There are advantages and disadvantages in the choice of Z to represent UML models (these are considered in more detail in [10]). In its favour, Z allows free expression of most set and predicate concepts; it has relatively mature tool support for writing and verifying descriptions; proof and refinement are well researched; Z is capable of describing most OO (Object-Oriented) concepts. The disadvantage of using Z to express OO models is that Z must explicitly capture the OO semantics as well as the modelled concepts (compared for example, to Object-Z). However, no current OO formalism provides the range and variety of tool support; verification and development approaches are not yet mature; and, the OO formalisms impose a particular semantics on the UML models, which may be appropriate for many situations, but does not give enough flexibility for expressing other interpretations.

Our OO Z structuring is based on that of Hall [11, 12]. We have introduced new features, such as the use of generics, our modular approach of organising the specification, and our encapsulation of the OO semantics.

Our approach makes use of, or elaborates, several Z patterns [13, 14]. Name predicates is the inspiration for separation of association rules into separate predicates, a source of much flexibility in the templates; name consistently is elaborated to define the naming conventions of our approach (see Appendix A). We believe that the modular structuring mechanisms presented here can be applied to other, perhaps non-OO, approaches to modelling of components and component architectures.

We have developed a core semantic interpretation of most concepts of UML class models [15]. In [15] we cover most concepts of the UML class diagram, including binary associations, association classes, and specialisation, and a UML toolkit. (Note that application-specific toolkit is one of the patterns identified in [14].) Here we focus on classes, composition relationships, and the expression of a system made up of subsystems.

In UML, as in other component models, there are various ways to relate components. Composition (and its weaker form, aggregation) model a relationship that had a semantic dependency,

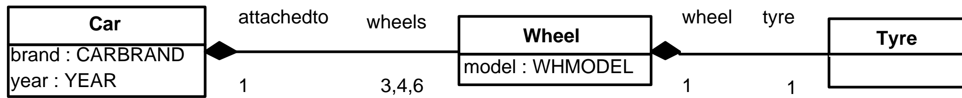

Figure 1: *Car*s are composites, where the components are a number of *Wheel*s (three, four or six). Each *Wheel* in turn has one component *Tyre*.

often referred of as a "part of" relationship. The classes, or components, are distinct, but are dependent (in the context of the model). In our illustration, figure 1, the composition relationships model the situation where instances of *Tyre* are inherently part of an instance of *Wheel*, and instances of *Wheel* are inherently part of a *Car* (the semantics are discussed further below). This example has been chosen to show our approach to multiple composition and how we explore the formal reasoning capabilities of Z.

In the following sections, we present our template notation, and illustrate aspects of our approach with the simple model of the car.

2 Template Notation

Our template notation comprises elements to be substituted, various forms of optionality, and various forms of list. These elements must be clear in the templates.

The following notational conventions are used.

- substitutable parameters are placed between the bracket symbols $\leqslant$and $\geqslant$
- optional elements are placed between $\llbracket$ and $\rrbracket$
- the symbol $[]$ is used within brackets to denote optionality: $(a \; [] \; b)$ means "either a or b"

Where a list of elements is used, we introduce a general subscripted component within optional brackets with the following decorations:

- $\llbracket \; element_i \; \rrbracket^*$ indicates a list of $element_1$ to $element_n$, where the list may be empty
- $\llbracket \; element_i \; \rrbracket^+$ indicates a list of $element_1$ to $element_n$, where the list must have at least one element

3 Classes

In the OO paradigm objects have identity. This needs to be represented in Z [10].

A class has two related but distinct meanings. *Class intension* defines the properties shared by the objects of that class (for example, class *Car* of figure 1 has properties *brand* and *year*). *Class extension* defines the class in terms of currently existing objects (for example, *Car* is $\{MyPorsche, HisFerrari, HerMini\}$). In our OO Z structuring, these two concepts are represented separately.

3.1 Object Identity

Our model of object identity is as follows. Each class has an *identity set*; each object has an identity, which is a member of its class identity set. Identity sets form a hierarchy. The root of the hierarchy is the set *IDOBJECT* (defined in the Z specification as given set). Actual class identity sets are either defined as a subset of *IDOBJECT*, or as a subset of a specific class identity, if the class being defined is a specialisation.

Here we present the template for defining class identity sets. Actual object identities are defined in the class extension.

Template and Illustration. The class identity set template (left) and the illustration of its instantiation (right) are given below. When we instantiate a template in order to translate a UML diagram we use systematic naming conventions; this can be observed in all illustrations.

$[IDOBJECT]$

$$\llbracket\ ID\langle\!\langle CL_i\rangle\!\rangle\ \rrbracket^+ \ :\ \mathbb{P}\ ID\langle\!\langle CLSup\rangle\!\rangle$$
$$\llbracket\ \text{disjoint}\langle\ \llbracket\ ID\langle\!\langle CL_i\rangle\!\rangle\ \rrbracket^+\ \rangle\ \rrbracket$$

$$IDCAR, IDWHEEL, IDTYRE : \mathbb{P}\ IDOBJECT$$
$$\text{disjoint}\langle\ IDCAR, IDWHEEL, IDTYRE\ \rangle$$

3.2 Class Intension

Class intension is represented in a Z schema. It defines attributes, association roles, and intension class invariant.

In the interpretation of associations presented here, we view associations as roles played by classes; an alternative model would be to consider an association as being a mathematical relation between classes [10]. This representation of associations involves role definition, and the definition of the properties of the association as a whole (next section).

3.2.1 Attributes

Attribute definition comprises the definition of its type (if not directly supported in Z) and the actual definition of the attribute.

[15] provides templates for defining attribute types as given sets and enumerations; here only given sets and Z's natural numbers are used. The template for attribute types as given sets, and an illustration of its instantiation, are:

$[\llbracket \lll ATTRTYPE_i \ggg \rrbracket^{+}]$ $\qquad\qquad$ $[CARBRAND, YEAR, WHMODEL]$

Attribute definitions link the attribute name with the appropriate attribute type construction. The type construction instantiates one of the attribute generics[1]. The template is:

$$\lll attrName \ggg : \lll attrGeneric \ggg [\lll ATTRTYPE \ggg] \llbracket \ (\lll n \ggg) \ \rrbracket \llbracket \ (\lll n_1 \ggg \ldots \lll n_2 \ggg) \ ) \ \rrbracket$$

In the illustration (below), *brand*, *year*, and *model* are single-valued attributes, and are declared using the $Attrib_1$ generic. Generics for multi-valued attributes are also provided; they have the same form as multi-valued roles (see below).

3.2.2 Roles

Association roles describe the participation of classes in associations. In an OO context, role definitions state the class of objects at the opposite end of the association, and the number of allowed object references that each class object may hold (multiplicity). They are essentially class attributes whose possible values are object references. Like class attributes, they are defined by instantiating appropriate generics from our toolkit.

The role definition template is:

$$\lll roleName \ggg : \lll roleGeneric \ggg [ID \lll CLEND \ggg] \llbracket \ ((\lll n \ggg) \ \rrbracket \llbracket \ (\lll n_1 \ggg \ldots \lll n_2 \ggg) \ ) \ \rrbracket$$

The instantiation is as for attributes. In the illustration (below), the roles of composition $Car \rightarrow Wheel$ are defined in the class intension schemas: *wheels* instantiates the $\mathrm{Role}_{©[\ ]..}$ generic with the proper multiplicity value; *tyre* instantiates $\mathrm{Role}_{©1}$ as multiplicity is one; *attachedTo* instantiates $\mathrm{Role}_{\circledcirc 1}$ generic. In our naming conventions, © stands for component role and $\circledcirc$ for composite role.

[1] [15] includes a toolkit containing attribute and role generics; a small subset of this toolkit is given in appendix B.

3.2.3 Intension Schema

In our naming convention, class intension schemas are preceded by $\mathbb{A}$.

Template. The class intension schema template is:

$$
\begin{array}{|l}
\hline
\mathbb{A}\!\ll\! Cl \!\gg \\
\llbracket\ \textit{attribute definition } i\ \rrbracket^* \\
\llbracket\ \textit{role definition } i\ \rrbracket^* \\
\hline
\llbracket\ \textit{intension invariant } i\ \rrbracket^* \\
\hline
\end{array}
$$

Illustration. The intension schema instantiation for classes *Car* and *Wheel* is:

$$
\begin{array}{|l}
\hline
\mathbb{A}\,Car \\
brand : \mathrm{Attrib}_1[CARBRAND] \\
year : \mathrm{Attrib}_1[YEAR] \\
wheels : \mathrm{Role}_{©[\ ]..}[IDWHEEL](\{3,4,6\}) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\mathbb{A}\,Wheel \\
model : \mathrm{Attrib}_1[WHMODEL] \\
attachedto : \mathrm{Role}_{\circledcirc 1}[IDCAR] \\
tyre : \mathrm{Role}_{©1}[IDTYRE] \\
\hline
\end{array}
$$

where the attributes and role generics used are defined in appendix B.

3.3 Class Extension

The class extension expresses the set of class objects in terms of object identities. Class extensions are defined generically. Names of class extension schemas are preceded by $\mathbb{S}$, following [12].

Generic and Template. The class extension generic and the template that describes its instantiation are:

$$
\begin{array}{|l}
\hline
\mathbb{S}\mathrm{Class}\,[IDCL, ATTRSCL] \\
objIds : \mathbb{P}\ IDCL \\
objAttrs : IDCL \rightarrow\!\!\!\!\!+ ATTRSCL \\
\hline
\mathrm{dom}\, objAttrs = objIds \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\mathbb{S}\!\ll\! Cl \!\gg \\
\mathbb{S}\mathrm{Class}[ID\!\ll\! CL \!\gg, \mathbb{A}\!\ll\! Cl \!\gg] \\
\quad [\ll\! cl \!\gg Ids/objIds, \ll\! cl \!\gg Attrs/objAttrs] \\
\hline
\llbracket\ \textit{extension invariant } i\ \rrbracket^* \\
\hline
\end{array}
$$

The generic defines a mapping from class identity set to class intension (expressed as partial function), linking objects to their state. The template instantiates the generic for a specific class, and renames the components of the generic based on the name of the class. This renaming is done to avoid name clashing, when all the extensions are put together in system schemas.

Illustration. The instantiation of the template for the class *Car* is:

$$\mathbb{S}Car == \mathbb{S}\mathrm{Class}[IDCAR, \mathbb{A}Car][carIds/objIds, carAttrs/objAttrs]$$

4 Composition

In the OO paradigm, *part-of* relationships are used when one class is part of or *subordinate to* another. There are many possible interpretations (or flavours) of this abstract OO concept; two named forms are *aggregation* and *composition*.

Composition defines a strong form of ownership, requiring that components (or parts) be associated to a composite (or whole) throughout their existence. Compositions have the following properties:

- the part must be included in at most one composite at a time (*unshared containment*);
- if a composite is destroyed the parts must also be destroyed (*deletion propagation*).

As for ordinary UML associations, representing compositions in Z involves the definition of class roles in intension schemas of participant classes and the definition of properties of the association in a *properties schema*. In addition, *deletion propagation* needs to be stated separately, as it involves the dynamic behaviour of objects.

Composition roles have been discussed in the previous section. Here we focus our attention on the properties schema and deletion propagation property.

4.1 Properties schema

Our Z representation of any kind of association uses a properties schema to enforce constraints of associations. This includes consistency of links and other association constraints (possibly stated in OCL). In our interpretation of associations, consistency of links includes *link coherence* (links do actually exist) and *mutual reference* (references to objects in the context of associations are mutual)[2]. In addition to these, compositions require the *unshared containment* constraint.

Each constraint is expressed in a separate Z schema, following the Name Predicates pattern [13]. The whole properties schema is built-up from these schemas.

[2] An interpretation without *mutual reference* is also possible.

LinkCoherence Template. This checks that object references held by objects are defined in the class extension schemas. Note the optionality between Z terms, which depends on role multiplicity.

$$
\begin{array}{l}
\textit{LinkCoherence} \ll Cl_1 \gg \ll Cl_2 \gg \\
\hline
\mathbb{S} \ll Cl_1 \gg ; \ \mathbb{S} \ll Cl_2 \gg \\
\hline
\forall \ \ll o_1 \gg \ : \ \ll cl_1 \gg Ids; \ \ll o_2 \gg : \ll cl_2 \gg Ids \ \bullet \\
\quad (\ll cl_1 \gg Attrs \ll o_1 \gg).\ll role_1 \gg (\ \subseteq \ \ [] \ \ \in \ ) \ll cl_2 \gg Ids \\
\quad \wedge \ (\ll cl_2 \gg Attrs \ll o_2 \gg).\ll role_2 \gg (\ \subseteq [] \in \ ) \ll cl_1 \gg Ids
\end{array}
$$

MutualReference Template. This checks that object references are mutual, i. e., if obj_a holds a reference to obj_b, then the inverse must also hold. The optionality is as above.

$$
\begin{array}{l}
\textit{MutualReference} \ll Cl_1 \gg \ll Cl_2 \gg \\
\hline
\mathbb{S} \ll Cl_1 \gg ; \ \mathbb{S} \ll Cl_2 \gg \\
\hline
\forall \ll o_1 \gg : \ll cl_1 \gg Ids; \ \ll o_2 \gg : \ll cl_2 \gg Ids \ \bullet \\
\quad (\ (\ \forall \ll o_2 i \gg : [] \ \mathbf{let} \ll o_2 i \gg == \ ) \ (\ll cl_1 \gg Attrs \ll o_1 \gg).\ll role_1 \gg \bullet \\
\qquad \ll o_1 \gg (\ \in [] = \ ) \ (\ll cl_2 \gg Attrs \ll o_2 i \gg).\ll role_2 \gg) \\
\quad \wedge \ (\ (\ \forall \ll o_1 i \gg : [] \ \mathbf{let} \ll o_1 i \gg == \ ) \ (\ll cl_2 \gg Attrs \ll o_2 \gg).\ll role_1 \gg \bullet \\
\qquad \ll o_2 \gg (\ \in [] = \ ) \ (\ll cl_1 \gg Attrs \ll o_1 i \gg).\ll role_1 \gg)
\end{array}
$$

LinkCoherence and MutualReference Illustration. The illustration for the $Car \rightarrow Wheel$ composition is:

$$
\begin{array}{l}
\textit{LinkCoherenceCarWheel} \\
\hline
\mathbb{S} Car; \ \mathbb{S} Wheel \\
\hline
\forall \ c : carIds; \ w : wheelIds \ \bullet \\
\quad (carAttrs \ c).wheels \subseteq wheelIds \wedge (wheelAttrs \ w).attachedto \in carIds
\end{array}
$$

$$
\begin{array}{l}
\textit{MutualReferenceCarWheel} \\
\hline
\mathbb{S} Car; \ \mathbb{S} Wheel \\
\hline
\forall \ c : carIds; \ w : wheelIds \ \bullet \\
\quad (\forall \ wi : (carAttrs \ c).wheels \bullet c = (wheelAttrs \ wi).attachedto) \\
\quad \wedge \ (\mathbf{let} \ ci == (wheelAttrs \ w).attachedto \bullet w \in (carAttrs \ ci).wheels)
\end{array}
$$

Other Constraints Template. This states other constraints of an association. (Since there are none in our example, it is not illustrated here.)

$$
\begin{array}{l}
\textit{OtherConstraints} \ll Cl_1 \gg \ll Cl_2 \gg \\
\hline
\mathbb{S} \ll Cl_1 \gg ; \ \mathbb{S} \ll Cl_2 \gg \\
\hline
[\![\ constraint \ predicate \ i \ ]\!]^{+}
\end{array}
$$

Unshared Containment Template. This states that the parts of any two distinct composite objects must be disjoint.

$$
\begin{array}{l}
\textit{UnsharedContainment}\ll ClWhole\gg\ll ClPart\gg \\
\hline
\mathbb{S}\ll ClWhole\gg \\
\hline
\forall \ll w_1\gg, \ll w_2\gg : \ll clWhole\gg Ids \mid \ll w_1\gg \neq \ll w_2\gg \bullet \\
\quad \mathrm{disjoint}\langle(\ll clWhole\gg Attrs\ \ll w_1\gg).\ll parts\gg, (\ll clWhole\gg Attrs\ \ll w_2\gg).\ll parts\gg\rangle
\end{array}
$$

Composition Properties Template. This groups all the composition properties in the predicate.

$$
\begin{array}{l}
\textit{Comp}\ll ClWhole\gg\ll ClPart\gg \\
\hline
\mathbb{S}\ll ClWhole\gg;\ \mathbb{S}\ll ClPart\gg \\
\hline
\textit{UnsharedContainment}\ll ClWhole\gg\ll ClPart\gg \\
\textit{LinkCoherence}\ll ClWhole\gg\ll ClPart\gg \\
\textit{MutualReference}\ll ClWhole\gg\ll ClPart\gg \\
[\![\ \textit{OtherConstraints}\ll ClWhole\gg\ll ClPart\gg\ ]\!]
\end{array}
$$

Illustration. The illustration of the *unshared containment* predicate and the properties schema for the $Car \rightarrow Wheel$ composition is,

$$
\begin{array}{l}
\textit{UnsharedContainmentCarWheel} \\
\hline
\mathbb{S}Car \\
\hline
\forall c_1, c_2 : carIds \mid c_1 \neq c_2 \bullet \\
\quad \mathrm{disjoint}\langle(carAttrs\ c_1).wheels, \\
\qquad (carAttrs\ c_2).wheels\rangle
\end{array}
$$

$$
\begin{array}{l}
\textit{CompCarWheel} \\
\hline
\mathbb{S}Car;\ \mathbb{S}Wheel \\
\hline
\textit{UnsharedContainmentCarWheel} \\
\textit{LinkCoherenceCarWheel} \\
\textit{MutualReferenceCarWheel}
\end{array}
$$

4.2 Deletion Propagation

Deletion propagation is stated in terms of deletion operations and assertion of the property through a conjecture.

4.2.1 Delete Operations

The *Delete* operation on composite objects, involves several operation schemas. *Delete*0 is applicable to any class, and *Delete* is only applicable to composite classes.

Delete0 Template. The $Delete0$ operation removes a set of objects from a class extension. It states that in the after-state of the operation the given objects are deleted from the class extension; it does not itself include deletion of the parts. The optional 0 is used if the class is a composite.

$$
\begin{array}{l}
\textbf{schema } Delete\ll Cl \gg [\![\ 0\ ]\!] \\
\Delta \mathbb{S}\ll Cl \gg;\ \ll cl \gg Ids? : \mathbb{P}\ ID\ll CL \gg \\
\hline
\ll cl \gg Ids? \subseteq \ll cl \gg Ids \\
\ll cl \gg Ids' = \ll cl \gg Ids \setminus \ll cl \gg Ids? \\
\ll cl \gg Attrs' = \ll cl \gg Ids? \lhd \ll cl \gg Attrs
\end{array}
$$

Delete0 Illustration. $Delete0$ is applied to the classes Car and $Tyre$.

$$
\begin{array}{l}
\textbf{schema } DeleteCar0 \\
\Delta \mathbb{S} Car;\ carIds? : \mathbb{P}\ IDCAR \\
\hline
carIds? \subseteq carIds \\
carIds' = carIds \setminus carIds? \\
carAttrs' = carIds? \lhd carAttrs
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } DeleteTyre \\
\Delta \mathbb{S} Tyre;\ tyreIds? : \mathbb{P}\ IDTYRE \\
\hline
tyreIds? \subseteq tyreIds \\
tyreIds' = tyreIds \setminus tyreIds? \\
tyreAttrs' = tyreIds? \lhd tyreAttrs
\end{array}
$$

PartsToDelete Template. If a class is a composite, its parts must also be deleted. The $PartsToDelete$ operation obtains the parts of the composite objects that are to be deleted.

$$
\begin{array}{l}
\textbf{schema } \ll ClWhole \gg PartsToDelete \\
\Xi \mathbb{S}\ll ClWhole \gg;\ \ll clWhole \gg Ids? : \mathbb{P}\ ID\ll CLWHOLE \gg \\
[\![\ \ll clPart_i \gg Ids? : \mathbb{P}\ ID\ll CLPART_i \gg\ ]\!]^{+} \\
\hline
[\![\ \ll clPart_i \gg Ids? = [\![\ \bigcup\ ]\!]\ \{\ll w \gg : \ll clWhole \gg Ids? \bullet (\ll clWhole \gg Attrs\ w).\ll parts_i \gg\}\ ]\!]^{+}
\end{array}
$$

PartsToDelete Illustration. The instantiation for Car and $Wheel$ is:

$$
\begin{array}{l}
\textbf{schema } CarPartsToDelete \\
\Xi \mathbb{S} Car;\ carIds? : \mathbb{P}\ IDCAR \\
wheelIds? : \mathbb{P}\ IDWHEEL \\
\hline
wheelIds? = \bigcup\{c : carIds? \bullet \\
\quad (carAttrs\ c).wheels\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } WheelPartsToDelete \\
\Xi \mathbb{S} Wheel;\ wheelIds? : \mathbb{P}\ IDWHEEL \\
tyreIds? : \mathbb{P}\ IDTYRE \\
\hline
tyreIds? = \{w : wheelIds? \bullet \\
\quad (wheelAttrs\ w).tyre\}
\end{array}
$$

Delete Template. The template of $Delete$ for composites is given below. In the declaration (before $\bullet$), the extensions of the part classes and the inputs of the delete operations on parts are hidden (as in promotion [13]). In the predicate (after $\bullet$), we have a conjunction of operations:

the part objects to delete are obtained (*PartsToDelete*), part objects are deleted (*Delete* on parts), and composite objects are deleted (*Delete* on composite). *Delete* on parts can be either an instantiation of *Delete*0 or *Delete* (if the part is also a composite) operation templates.

$$\begin{aligned} Delete{\ll}ClWhole{\gg} == \exists\ [\![\ \Delta{\ll}\mathbb{S}ClPart_i{\gg};\ {\ll}clPart_i{\gg}Ids? : \mathbb{P}\ ID{\ll}CLPART_i{\gg}\ ]\!]^+ \bullet \\ {\ll}ClWhole{\gg}PartsToDelete \land [\![\ Delete{\ll}ClPart_i{\gg}[\![\ 0\ ]\!]\ ]\!]^+ \\ \land\ Delete{\ll}ClWhole{\gg}0 \end{aligned}$$

Delete Illustration. The instantiation of *Delete* is illustrated for *Wheel* and *Car*.

$$\begin{aligned} DeleteWheel &== \exists\ \Delta\mathbb{S}Tyre;\ tyreIds? : \mathbb{P}\ IDTYRE \bullet \\ &\quad WheelPartsToDelete \land DeleteTyre \land DeleteWheel0 \\ DeleteCar &== \exists\ \Delta\mathbb{S}Wheel;\ wheelIds? : \mathbb{P}\ IDWHEEL \bullet \\ &\quad CarPartsToDelete \land DeleteWheel \land DeleteCar0 \end{aligned}$$

4.2.2 Delete Propagation Conjecture

As in the specification pattern express implicit properties [14], we use a conjecture to test the assertion that deletion propagates through compositions.

Conjecture Template. The required condition of the after state of any delete operation, that the input objects are deleted, is expressed in a separate schema, *CondPostDelete*.

$$\begin{array}{|l} \hline CondPostDelete{\ll}Cl{\gg} \\ \mathbb{S}{\ll}Cl{\gg}' \\ {\ll}cl{\gg}Ids? : \mathbb{P}\ ID{\ll}CL{\gg} \\ \hline {\ll}cl{\gg}Ids? \cap {\ll}cl{\gg}Ids' = \varnothing \\ {\ll}cl{\gg}Ids? \cap \operatorname{dom}{\ll}cl{\gg}Attrs' = \varnothing \\ \hline \end{array}$$

In the full conjecture, *CondPostDelete* is instantiated for each class of the composition hierarchy. The conjecture asserts that composite objects must be deleted, along with its part objects, and that deletion propagates down the hierarchy (if parts are composites, its parts must also be deleted).

The antecedent of the conjecture introduces the state and after state of the system (objects are deleted from the collection of system objects), and the delete operation on a composite.

$$\begin{aligned} &\Delta System;\ Delete{\ll}ClWhole{\gg} \\ &\vdash \\ &CondPostDelete{\ll}ClWhole{\gg} \\ &\land (\exists{\ll}ClWhole{\gg}PartsToDelete \bullet [\![\ CondPostDelete{\ll}ClPart_i{\gg}\ ]\!]^+) \\ &\qquad \text{[The deletion of sub-components]} \\ &\qquad [\![\ \land (\exists{\ll}ClPart_i{\gg}PartsToDelete \bullet [\![\ CondPostDelete{\ll}ClSubPart_i{\gg}\ ]\!]^+)\ ]\!]^* \end{aligned}$$

Conjecture Illustration. The conjectures for the two compositions of figure 1 are:

$$\Delta System;\ DeleteWheel$$
$$\vdash CondPostDeleteWheel \land (\exists\ WheelPartsToDelete \bullet CondPostDeleteTyre)$$

$$\Delta System;\ DeleteCar$$
$$\vdash CondPostDeleteCar \land (\exists\ CarPartsToDelete \bullet CondPostDeleteWheel)$$
$$\land\ (\exists\ WheelPartsToDelete \bullet CondPostDeleteTyre)$$

We are looking at proving the conjecture for the example system in a form suitable for generalisation to the template form. Given a template proof, two things follow. Firstly, if the translation templates are correctly instantiated, the instantiated template proof automatically generates the specific proof. (If *OtherConstraints* are present, they may have to be discharged explicitly.) Secondly, if there is doubt about the instantiation of the templates, reproving the conjecture increases confidence in the translation, at least for this part of the system.

5 System and Subsystems

In representing large class models it is often convenient to group classes and relationships into subsystems. The whole system is built up from all the subsystems. In our approach, we use Z schemas to represent subsystems and systems.

Templates The subsystem schema includes the extensions of subsystem classes, and relationships between those classes. The system schema template just includes the subsystem schemas.

Subsystem$\ll Subsys \gg$

$\llbracket\ \mathbb{S}\ll Cl_i \gg\ \rrbracket^{+}$

$\llbracket\ Assoc\ll Cl_{k1} \gg\ll Cl_{k2} \gg\ \rrbracket^{*}$

$\llbracket\ Generalisation\ll ClSuper_j \gg\ \rrbracket^{*}$

$\llbracket\ AssocCl\ll AssocCl_l \gg\ll Cl_{k3} \gg\ll Cl_{k4} \gg\ \rrbracket^{*}$

$\llbracket\ Comp\ll ClWhole_w \gg\ll ClPart_p \gg\ \rrbracket^{*}$

System

$\llbracket\ \ll SubSys_i \gg\ \rrbracket^{+}$

Illustration. We illustrate our modular approach by grouping the model of figure 1 in two subsystems (for this small example one would suffice): the classes of the $Car \rightarrow Wheel$ composition are grouped in *SubsystemCar*, and the remaining ones in *SubsystemWheel*. *System* includes both subsystems.

SubsystemCar

$\mathbb{S}Car;\ \mathbb{S}Wheel$

CompCarWheel

SubsystemWheel

$\mathbb{S}Wheel;\ \mathbb{S}Tyre$

CompWheelTyre

```
┌─ System ──────────────────────────────
│ SubsystemCar; SubsystemWheel
└───────────────────────────────────────
```

6 Discussion

Some earlier formalisations have addressed composition characteristics through operations [5], and in terms of conjectures [7]. What is new here is the full expression of the additional constraints that are inherent to composition, the complete, extensible composition operations, and the conjecture on deletion for the whole composition hierarchy.

The authors are not aware of any other discussion of composition that incorporates hierarchies with more than one level. Here we illustrate it for the $Car \rightarrow Wheel \rightarrow Tyre$ composition.

The composition conjecture is a good example of the exploration of the formal reasoning mechanisms of a FSL in a context of combined use with diagrammatic notations.

We would like to emphasise the modularity and abstraction of our approach. Modularity can be observed at both the meta and instantiation levels. At the instantiation level, we can observe it in our approach to build subsystems and systems. At the meta-level, we can observe the *plug and play* semantics in our approach to composition, for example, if the *MutualReference* interpretation is not desired one can create a properties schema template that excludes it; if an extra property is desired one can create a template that includes it.

Abstraction can be observed in our approach to build the composition properties schema. Each property of composition is expressed in a separate schema, which encapsulates the actual details of how the property is stated. Property schemas are then included in the whole composition properties schema, and refered to by meaningful names, abstracting away from the details involved in the statement of the property.

7 Related Work

Our approach is related to meta-modelling: the specification of the concepts used to build models. However, unlike most uses of meta-modelling (for example, the current UML definition [1]), the meta-modelling concepts are precisely and unambiguously defined[3].

D'Souza and Wills also propose an approach to define the intended meaning of UML modelling constructs through what they termed *package semantics* [16]. However, in their work, the meaning is always defined informally by resorting to notations without a formal semantics.

[3]Most uses of meta-modelling define notations by resorting to another graphical notations (class diagrams, entity-relationship diagrams, etc.) whose precise meaning is never explicitly defined.

Other work related to ours is on the development of meta-modelling frameworks, which is being done in the context of the UML 2.0 effort. These frameworks allow the definition of graphical modelling notations, following a meta-modelling approach, and improving on previous uses of meta-modelling by defining precisely the meta-modelling concepts using meta-languages with a formal semantics. This includes the development of BOOM [17] and MMF [18, 19]. This line of work differs from ours as follows.

- It aims at the sole use of graphical notations with precise semantics; we aim at a combined use of graphical notations and FSLs.
- It resorts to newly defined formalisms for defining modelling concepts; we use a well-established FSL, with a proof system and a refinement calculus, opening to us years of research in formal methods.

8 Future Work

We are looking at annotating UML models with properties currently expressed only in Z, such as invariants and conjectures. In our framework we elect Z as the language for expressing detailed properties; the use of OCL [20] would require a further translation to Z, besides OCL does not have a formal semantics and has many problems [21].

We also intend to explore the refinement calculus in this framework. We aim at obtaining a refined Z specification through calculation, and from this refined Z specification obtain a UML representation. This is possible provided the overall OO structuring is maintained in the refinement, and by exploring our annotation facility to represent extra information not expressible in UML.

Future work will also involve statecharts. If we explore a concurrent interpretation of statecharts (as most are) we may use Circus [22], as Z is limited at expressing concurrent properties.

The translation of UML models into Z (that is the instantiation of the templates) is currently done manually. This could be automated, which would require formalising (in some sense) the template concepts. Our aim is to build a tool that allows specifiers to make translations of diagrams based on a selected semantic interpretation. Such a tool would keep an association between problem domain and semantic interpretations, and would allow the addition of new problem domains and associated semantic interpretations. Each semantic interpretation is represented by a set of templates and generics.

9 Conclusions

We have presented key examples of a set of templates that can be instantiated to produce a formal (Z) specification of a component system modelled in a UML class diagram. This set of templates incorporates a particular semantic interpretation.

We have also shown the modular structure of our framework. At the instantiation level concepts are factored out in schemas, and schemas are combined to make the whole. At the meta-level these instantiation pieces are factored out as templates and generics.

We believe that the principles outlined here can be used to give formal interpretations to other kinds of diagrams, including component models in other paradigms, such as ADLs.

Our approach is akin to component-based development in its emphasis on *compositionality*, *reuse*, and *adaptability*:

- concepts are represented as pieces (or modules) that are composed with other pieces to build a whole;
- desired properties of a composition of pieces are stated as conjectures, whose satisfaction can be verified through proof;
- our templates are powerful and flexible units of reuse allowing us to factor out commonly used structures;
- new semantic interpretations are built by assembling and adapting a variety of exiting pieces;
- domain-specific interpretations are also units of reuse, encapsulating valuable modelling experience that can be reused in other problems within the same domain, or that can be adapted to meet the needs of different problem domains.

Our approach, using generics and templates,

- enhances the readability and abstraction of the Z specifications resulting from the UML translation;
- enhances the modularity and compositionality of our approach, making very clear the distinct components of the UML model,facilitating the combination of different diagrams, and allowing reuse;
- enhances the traceability of the Z specification, owing to the modularity and the naming conventions of our approach.

The modularity and abstraction mechanisms of our approach are substantial improvements over previous formalisations of UML (see [10]). Our Z meta-structuring, based on templates and generics, and our treatment of multiple level composition using template conjectures, are new. We intend to develop further our meta-structuring to handle proof and refinement.

Acknowledgements. This research was supported for Amálio by the Portuguese Foundation for Science and Technology under grant 6904/2001.

References

[1] OMG. *Unified Modeling Language Specification, version 1.5.* Object Management Group (2003). Available at http://www.omg.org/uml

[2] Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual.* Addison Wesley Longman, Reading, Mass. (1999)

[3] Kent, S., Evans, A., Rumpe, B. UML semantics FAQ. In Moreira, A., Demeyer, S., eds., *Object-Oriented Technology, ECOOP'99*, volume 1743 of *LNCS*. Springer-Verlag (1999)

[4] Nguyen, H. P. *Dérivation De Spécifications Formelles B à partir de Spécifications Semi-Formelles.* Ph.D. thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Evry, France (1998)

[5] France, R. B., Grant, E., Bruel, J.-M. UMLtranZ: An UML-based rigorous requirements modeling technique. Technical report, Colorado State University, Fort Collins, Colorado, USA (2000)

[6] Bruel, J.-M., France, R. B. Transforming UML models to formal specifications. In Bézivin, J., Muller, P.-A., eds., *UML'98: The Unified Modeling Language, Mulhouse, France*, volume 1618 of *LNCS*. Springer-Verlag (1998)

[7] Dupuy, S. *Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information.* Ph.D. thesis, Université Joseph Fourier, Grenoble I (2000)

[8] Kim, S.-K., Carrington, D. A formal mapping between UML models and Object-Z specifications. In Bowen, J., et al., eds., *ZB 2000: Formal Specification and Development in Z and B, York, UK*, volume 1878 of *LNCS*, pp. 2–21. Springer-Verlag (2000)

[9] Spivey, J. M. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition (1992)

[10] Amálio, N., Polack, F. Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In Bert et al. [23], pp. 339–358

[11] Hall, A. Using Z as a specification calculus for object-oriented systems. In Hoare, C. A. R., Bjørner, D., Langmaack, H., eds., *VDM '90 VDM and Z– Formal Methods in Software Development*, volume 428 of *LNCS*, pp. 290–318. Springer (1990)

[12] Hall, A. Specifying and interpreting class hierarchies in Z. In Bowen, J., Hall, A., eds., *Z User Workshop, Cambridge*, Workshops in Computing, pp. 120–138. Springer-Verlag (1994)

[13] Stepney, S., Polack, F., Toyn, I. Patterns to guide practical refactoring: examples targetting promotion in Z. In Bert et al. [23], pp. 20–39

[14] Stepney, S., Polack, F., Toyn, I. A Z patterns catalogue I, specification and refactoring. Technical Report YCS-2003-349, Department of Computer Science, University of York (2003)

[15] Amálio, N., Polack, F., Stepney, S. Templates and generics for translating UML class diagrams into Z. Technical Report YCS-2003, Department of Computer Science, University of York (2003). In press

[16] D'Sousa, D., Wills, A. C. *Object Components and Frameworks with UML: the Catalysis approach*. Addison-Wesley (1998)

[17] Övergaard, G. Formal specification of object-oriented meta-modelling. In Maibaum, T., ed., *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pp. 193–207. Springer (2000)

[18] Clark, T., Evans, A., Kent, S. Engineering modelling languages: A precise meta-modelling approach. In R.-D.Kutsche, Weber, H., eds., *Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *LNCS*, pp. 159–173. Springer-Verlag (2002)

[19] Clark, T., Evans, A., Kent, S. The metamodelling language calculus: foundation semantics for UML. In Hussmann, H., ed., *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *LNCS*, pp. 17–31. Springer-Verlag (2001)

[20] Warmer, J., Kleppe, A. *The Object Constraint Language: precise modeling with UML*. Addison-Wesley (1999)

[21] Vaziri, M., Jackson, D. Some shortcomings of OCL, the object constraint language of UML. In Li, Q., Firesmith, D., Riehle, R., Pour, G., Meyer, B., eds., *TOOLS: 34th International Conference*, pp. 555–560. IEEE Computer Society (2000)

[22] Woodcock, J., Cavalcanti, A. The semantics of Circus. In Bert, D., et al., eds., *ZB 2002: Formal Specification and Development in Z and B, Grenoble*, volume 2272 of *LNCS*, pp. 184–203. Springer (2002)

[23] Bert, D., Bowen, J. P., King, S., Waldén, M., eds. *ZB 2003: Formal Specification and Development in Z and B, Int. Conference, Turku, Finland*, volume 2651 of *LNCS*. Springer-Verlag (2003)

A Naming Convention Pattern

The Z pattern Name Consistently encourages the declaration and use of an explicit naming convention. The convention that we use has three elements, each presented here as a separate

pattern elaboration.

Name Consistently (elaboration): Distinguish Class Elements

Intent: The UML class concept is formalised as an intension (or type) and an extension (or set of objects). These need to be clear in the templates, and in the resultant Z.

Solution: Following Hall [11, 12], class extension schemas are preceeded by $\mathbb{S}$. We introduce the $\mathbb{A}$ to preceeded class intension schemas.

□

Name Consistently (elaboration): Distinguish Generic Schemas

Intent: distinguish specific elements of the Z language for readability

Problem: This paper defines generic and template schemas, as well as giving actual Z descriptions. The templates and the Z descriptions instantiate generics. The templates, in particular, are not easy to read. It is desirable to be able to distinguish the generic schema definitions and their instantiations.

Solution: Use a different font for the name of generic schemas, in definition and instantiation. Here, we use plain Roman font for generics, and default italic face for all other Z.

□

B Toolkit

Bounded Containers. Here we present some of the *bounded containers* generics of [15]. These are called bounded because a limit is imposed on the size of the container.

There are several kinds of bounded containers. Bounded powersets are presented below — one defines a container whose size is within a set of possible values, the other defines a container whose size is a specific value. [15] also defines containers of kind sequences, injective sequences and bags.

$$\mathrm{bset}_{..}[X] == (\lambda\, r : \mathbb{F}\,\mathbb{N} \bullet \{xs : \mathbb{F}\, X \mid \# xs \in r\})$$
$$\mathrm{bset}_{\mathrm{n}}[X] == (\lambda\, n : \mathbb{N} \bullet \mathrm{bset}_{..}[X](n \mathinner{..} n))$$

UML Multiplicity. Multiplicity constraints restrict the number of values (or references) held by roles and attributes. The constant *many* represents the UML '*'; *UMLCard* (cardinality) defines the set of multiplicity values; $UMLCard_2$ is just a subset of *UMLCard*.

$| \; \mathrm{many} : \mathbb{N}_1$

$\mathrm{UMLCard} == \{x : \mathbb{N} \mid x \leq \mathrm{many}\}$

$\mathrm{UMLCard}_2 == \mathrm{UMLCard} \setminus \{0, 1\}$

Role and Attribute Generics. [15] define several kinds of roles and attribute generics. Here we define the ones used in this paper. Attribute and role generics have very similar forms.

$\mathrm{Attrib}_1[X] == X$

$\mathrm{Role}_{\copyright 1}[X] == X$

$\mathrm{Role}_{\copyright[\;]..}[X] == \mathrm{bset}_{..}[X] \cap (\mathbb{F}\,\mathrm{UMLCard} \times \mathbb{P}(\mathbb{F}\,X))$

$\mathrm{Role}_{\odot 1}[X] == X$