Chapter 5

UML+Z: Augmenting UML with Z

Nuno AMÁLIO, Fiona POLACK, and Susan STEPNEY

5.1 Overview of *UML + Z*

UML + Z is a framework for building, analysing and refining models of software systems based on the UML and the formal specification language Z. It is, in fact, an instance of an approach to build rigorous engineering frameworks for model-driven development based on templates, which we call *CiTRUS* [AMA 06]. *UML + Z* is targeted at developers who have minimal knowledge of Z, but are familiar with UML-based modeling. *UML + Z* models comprise class, state and object UML diagrams, which are represented in a common Z model (the semantic domain) in Figure 5.1; the Z model gives the precise meaning of the diagrams. The framework tries to minimise exposure to the Z model, with UML diagrams acting like a graphical interface for the formality (Z) that lies underneath, but this is not always possible and one Z expert is required in the development to describe system properties that are not expressible diagrammatically (mainly operations and constraints).

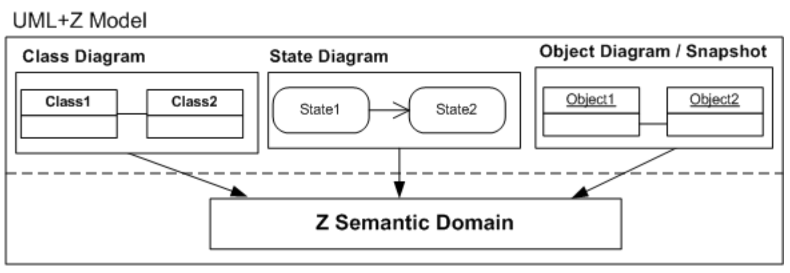

Figure 5.1: Models in the *UML + Z* framework

A crucial component of $\mathit{UML} + Z$ is a catalogue of *templates* and *meta-theorems*. Templates are generic representations of sentences of some formal language, which, when instantiated, yield actual language sentences. To express templates, we have developed the formal template language (FTL), which enables an approach to proof with template representations of Z (*meta-proof*). This enables the representation of structural Z patterns as templates (e.g. the structure of a Z operation), but also reasoning with these template representation to establish *meta-theorems* (e.g. calculate a precondition). Every sentence of the Z model in $\mathit{UML} + Z$ is generated by instantiating one of the templates, and meta-theorems can be used to simplify, and in some cases fully discharge, proofs associated with the Z model.

The modeling and analysis process with $\mathit{UML} + Z$ begins with the drawing of UML class and state diagrams. These diagrams are then used to instantiate templates from the catalogue to generate the Z model. The developer then adds operations and system constraints that are not expressed diagrammatically to the Z model. The Z specification is then checked for consistency using the meta-theorems of the $\mathit{UML} + Z$ catalogue and a Z theorem prover. Finally, there is a process of model analysis, where the developer draws snapshots to validate the model; these snapshots are represented in Z and the analysis is assisted by the Z/Eves theorem prover. Usually, the analysis phase of the process results in changes to the diagrams or the portion of the Z model not expressed diagrammatically.

Currently, the process of instantiating templates is manual. The templates from the catalogue are manually selected and instantiated with names from the diagrams. The reasoning with templates is also based on manual instantiation, but it is assisted by Z proof tools such as Z/Eves [SAA 97].

5.2 Analysis and specification of case 1

We start by modeling the static and behavioral aspects of the system with UML class and state diagrams. Then we discuss the Z model that is generated from the diagrams by instantiating templates. Finally, the $\mathit{UML} + Z$ model is consistency-checked and analysed.

The following terminology is used to refer to the Z generated from templates:

- *fully generated* – the Z is fully generated by instantiating templates with information of diagrams. If we had a tool, the Z would be automatically generated;
- *partially generated* – the instantiation depends on information that does not come from UML diagrams, but needs to be explicitly added by the developer (usually constraints not expressible diagrammatically).

5.2.1 UML class model

A UML class diagram describes *classes* and their relationships. At the abstract level, a class diagram captures the main entities (or concepts) of a system and the structural relationship that exists between them. A class denotes a set of objects (individuals) that share certain *attributes* and *operations*.

Question 1: What data does the system manage? What are the main entities of the system and what are the relationships among them?

Answer: The case study states that the system is about invoicing orders that are placed for products. We model *order* and *product* as two UML classes and relate them with an association.

Question 2: Are there limits on the number of products that an order can reference? How many orders can reference the same product?

Answer: The case study makes clear that each order references precisely one product, but a product can be referenced by many orders. We assume that there are products that are not referenced by orders. We represent this information in the multiplicity of the association between *Order* and *Product*: *Order* references *one* product, but a product may be referenced by *many* orders.

Question 3: What information should the system hold for products and orders?

Answer: The case study states that orders can be for different quantities of the same reference; we represent the ordered *quantity* as an attribute of the class *Order*. The case study also refers to quantities of products available in stock. We model the *stock* quantity as an attribute of class *Product*.

Question 4: Do we know what type of information these quantities hold? Are products whole things (eg. books or parts) or continuous amounts (eg. water or cloth)? Can there be negative stock?

Answer: The case study does not fully answer these questions. We assume that the attributes are of the same type, since the things being ordered are from the stock of a product. We know that a mathematical ordering on the type is required (since stock may be less than or equal to the ordered quantity). So, based on our intuitive notion of quantity, we assume that the attributes *quantity* and *stock* are natural numbers (we do not allow negative stock).

These features are modeled in a UML class diagram in Figure 5.2. This says that there are two classes, *Order* and *Product*, each representing a set of objects (the orders and products of the system). Each *Order* object has a *quantity* attribute, referring to the ordered quantity of a product, and each *Product* object has a *stock* attribute, recording how much of the product is available in stock. *Order* and *Product* are related through an association, which says that each order refers to exactly one product, and that each product may be referred by many orders.

5.2.2 UML state models

UML state diagrams describe the permitted state transitions of the objects of a class. Here we draw the UML state diagram of class *Order*.

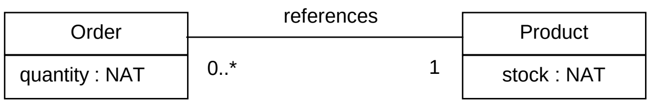

Figure 5.2: Initial version of the high-level UML class diagram

Question 5: What happens when a product is ordered? What happens to an order?

Answer: The case study says that an order can be *invoiced*, and that this changes the state of an order from *pending* to *invoiced*. Invoicing only occurs if the ordered quantity is either less than or equal to the quantity that is in stock. We assume that all new orders have the state, *pending*. When invoiced, the order changes from the state *pending* into the state *invoiced*. This is captured in the state diagram of Figure 5.3. Note that the constraint *orders are invoiced only if there is enough stock to fulfill the order* (ordered *quantity* $\leq$ *stock* quantity) is not expressed here as a guard of the state transition. This because that constraint involves the state of a *Product* object and here we are restricted to state constraints expressible in terms of the state of *Order*.

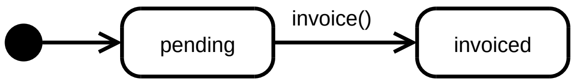

Figure 5.3: The state diagram of *Order*

5.2.3 The Z model

The Z model resulting from the UML class and state diagrams above is *fully generated*. It enables formal consistency-checking and validation of the whole *UML* + *Z* model. The Z model follows a structuring to specify object-oriented (OO) systems in Z, which uses *views* to separate the description of the different aspects of a system (see [AMA 05] for full details); there are five views, namely, structural, intensional, extensional, relational and global. In the following, the fully generated Z model is presented for each view.

The Z model follows certain naming conventions. The names of operations that perform a change of state include the symbol Δ. The names of extensional view definitions are prefixed by $\mathbb{S}$, and the ones of the relational view by $\mathbb{A}$.

5.2.3.1 Structural view

The *structural* view defines the set of all classes of a model as atoms, and assigns to each class a set of objects: the set of all possible objects of the class.

The Z toolkit of $UML + Z$ defines the set of all objects (OBJ) and asserts that this set is non-empty:

$$[OBJ] \qquad\qquad OBJ \neq \varnothing$$

The set $CLASS$ defines the set of all classes of a model and the function $\mathbb{O}$ gives all the possible objects of a class. As we do not have *subclassing* (inheritance) in our model the sets of objects of each class are mutually disjoint:

$$CLASS ::= OrderCl \mid ProductCl \qquad \begin{array}{|l} \mathbb{O} : CLASS \rightarrow \mathbb{P}_1\, OBJ \\ \hline \text{disjoint}\, CLASS \lhd \mathbb{O} \end{array}$$

5.2.3.2 Intensional view

The intensional view describes the intensional meaning of a class, that is: the properties shared by all its objects. This amounts to defining: (a) the state space of objects, which includes the definition of class attributes; (b) the initialisation of state; and (c) the operations of the objects of the class.

The class *Order* requires a type representing the set of all states of the state diagram:

$$ORDERST ::= pending \mid invoiced$$

The state space comprises *state*, which records the current state of an *Order* as defined in the state diagram, and *quantity*, which is defined in the class diagram:

$$\begin{array}{|l} \hline Order \\ state : ORDERST \\ quantity : \mathbb{N} \\ \hline \end{array} \qquad \begin{array}{|l} \hline OrderInit \\ Order' \\ quantity? : \mathbb{N} \\ \hline state' = pending \\ quantity' = quantity? \\ \hline \end{array}$$

The invoice operation captures the state transition from *pending* to *invoiced*, as described in the state diagram:

$$\begin{array}{|l} \hline Order_{\triangle}Invoice \\ \Delta Order \\ \hline state = pending \wedge state' = invoiced \wedge quantity' = quantity \\ \hline \end{array}$$

The intension of *Product* is defined as described in the class diagram in Figure 5.2:

$$\begin{array}{|l} \hline Product \\ stock : \mathbb{N} \\ \hline \end{array} \qquad \begin{array}{|l} \hline ProductInit \\ Product' \\ \hline stock' = 0 \\ \hline \end{array}$$

5.2.3.3 Extensional view

The extensional view describes the extensional meaning of classes, that is, the set of all existing objects of a class. Like intensions, class extensions comprise a state space, an initialisation and operations. A class extension is defined by using Z *promotion* [WOO 96, STE 03], a structuring technique for constructing aggregate (or whole-part) structures. Promotion is used whenever there is a component that comprises independent parts, with their own state. In this case, the state of a class extension is defined as an aggregate structure comprising all its existing objects (each an instance of a class intension): the class extension *promotes* its intension.

The state space of all class extensions is defined by instantiating the $\mathbb{S}\mathrm{Class}$ *generic* schema of the framework's Z toolkit. This generic introduces the parameters *OSET* and *OSTATE*, which are to be substituted by the set of all objects of the specific class and the intensional state space of the class (as defined above). The schema component *objs* represents the set of existing objects of the class and, *objSt* is a function that gives the current state of one object:

$$\begin{array}{|l}
\hline \mathbb{S}\mathrm{Class}\,[\mathit{OSET}, \mathit{OSTATE}] \\
\mathit{objs} : \mathbb{P}\,\mathit{OSET} \\
\mathit{objSt} : \mathit{OSET} \leftrightarrow\!\!\!\!\!\!\;\!+\; \mathit{OSTATE} \\
\hline
\mathrm{dom}\,\mathit{objSt} = \mathit{objs} \\
\hline
\end{array}$$

The extension of *Order* instantiates the class generic. The components of the generic are renamed to avoid name clashes when all the extensional schemas are put together to make the system schema (see below). The initialisation sets the schema components to the empty set: in the initial state there no orders:

$$\mathbb{S}\mathit{Order} == \mathbb{S}\mathrm{Class}[\mathbb{O}\mathit{OrderCl}, \mathit{Order}][\mathit{orders}/\mathit{objs}, \mathit{orderSt}/\mathit{objSt}]$$
$$\mathbb{S}\mathit{OrderInit} == [\ \mathbb{S}\mathit{Order}\,' \mid \mathit{orders}' = \varnothing \land \mathit{orderSt}' = \varnothing\ ]$$

Operations defined using promotion require an auxiliary schema that specify the *frame* of the operation: the aggregate components that are to change as a result of the operation and those that should remain unaltered. There are different kinds of promotion frames (see [AMA 05, WOO 96, STE 03] for details). To create a new *Order* object, we need a frame specifying the addition of a new object to the class extension (names of promotion frames are preceeded by Φ):

$$\begin{array}{|l}
\hline \Phi\mathbb{S}\mathit{OrderNew} \\
\Delta\mathbb{S}\mathit{Order}\,;\ \mathit{Order}\,' \\
\mathit{oOrder}! : \mathbb{O}\mathit{OrderCl} \\
\hline
\mathit{oOrder}! \in \mathbb{O}\mathit{OrderCl} \setminus \mathit{orders} \\
\mathit{orders}' = \mathit{orders} \cup \{\mathit{oOrder}!\} \\
\mathit{orderSt}' = \mathit{orderSt} \cup \{\mathit{oOrder}! \mapsto \theta\mathit{Order}'\} \\
\hline
\end{array}$$

The Z operator θ captures a schema binding: an assignment of values to the schema's variables. The frame says that the newly created object (*oOrder*!) will be mapped to some state, which is defined in the initialisation being promoted. The actual operation to create a new *Order* object uses the framing schema:

$$\mathbb{S}_{\triangle}\mathit{OrderNew} == \exists\ \mathit{Order}\,' \bullet \Phi\mathbb{S}\mathit{OrderNew} \wedge \mathit{OrderInit}$$

The binding ($\theta\mathit{Order}'$) of the frame is specified in *OrderInit*.

The update frame specifies a state transition of a single *Order* object:

$$\begin{array}{|l}
\multicolumn{1}{l}{\Phi\mathbb{S}\mathit{OrderUpdate}} \\
\hline
\Delta\mathbb{S}\mathit{Order}\ ;\ \Delta\mathit{Order} \\
\mathit{oOrder}? : \mathbb{O}\mathit{OrderCl} \\
\hline
\mathit{oOrder}? \in \mathit{orders} \\
\theta\mathit{Order} = \mathit{orderSt}\ \mathit{oOrder}? \\
\mathit{orders}' = \mathit{orders} \\
\mathit{orderSt}' = \mathit{orderSt} \oplus \{\mathit{oOrder}? \mapsto \theta\mathit{Order}'\} \\
\hline
\end{array}$$

The extension operation to invoice an *Order* uses the update frame:

$$\mathbb{S}_{\triangle}\mathit{OrderInvoice} == \exists\,\Delta\mathit{Order} \bullet \Phi\mathbb{S}\mathit{OrderUpdate} \wedge \mathit{Order}_{\triangle}\mathit{Invoice}$$

The extension of *Product* is similarly defined.

5.2.3.4 Relational view

The relational view defines the associations between classes. Associations are represented as a relation in Z, and denote a set of object tuples (the objects being related).

The state of the association *References* is defined as a mathematical relation between the set of all objects of the class *ORder* and that of *Product*. The initialisation sets all tuples to empty: in the initial state there are no objects and no links between them:

$$\mathbb{A}\mathit{References} == [\ \mathit{references} : \mathbb{O}\mathit{OrderCl} \leftrightarrow \mathbb{O}\mathit{ProductCl}\ ]$$
$$\mathbb{A}\mathit{ReferencesInit} == [\ \mathbb{A}\mathit{References}\,' \mid \mathit{references}' = \varnothing\ ]$$

5.2.3.5 Global view

This view looks at the system from a global viewpoint. It represents the system structure as a composition of local structures (classes and associations), and constraints that can only be expressed in the context of the system as a whole.

The multiplicity of associations is a constraint that cannot be expressed in the relational view. These constraints affect the existing objects of the associated classes, defined in the class extension. The schema *Link$\mathbb{A}$References* expresses the multiplicity constraint of the association *References* using the generic $\mathrm{Rel}_{*,1}$ from the Z toolkit of *UML* $+$ *Z*; this says that *references* (a relation) is constrained to be a total function, from the set of existing orders to the set of existing products:

$$\mathrm{Rel}_{*,1}[X, Y] == X \rightarrow Y$$

$$\begin{array}{|l}
\textit{Link}\mathbb{A}\textit{References} \\
\hline
\mathbb{S}\textit{Order}; \mathbb{S}\textit{Product}; \mathbb{A}\textit{References} \\
\hline
\textit{references} \in \mathrm{Rel}_{*,1}[\textit{orders}, \textit{products}] \\
\hline
\end{array}$$

The system schema includes all component schemas and the association constraint:

$$\begin{array}{|l}
\textit{System} \\
\hline
\mathbb{S}\textit{Order}; \mathbb{S}\textit{Product}; \mathbb{A}\textit{References} \\
\hline
\textit{Link}\mathbb{A}\textit{References} \\
\hline
\end{array}$$

The initialisation of the system is the initialisation of the system's components:

$$\textit{SysInit} == \textit{System}' \land \mathbb{S}\textit{OrderInit} \land \mathbb{S}\textit{ProductInit} \land \mathbb{A}\textit{ReferencesInit}$$

5.2.4 Checking model consistency

Question 6: Is the model internally consistent?

Answer: The Z model is type-correct (checked with Z-Eves). But a model can be type-correct and still be inconsistent. The consistency of Z models is demonstrated by proving certain conjectures. Z conjectures have the form $\vdash ?\ P$, where P is a well-formed Z predicate, which is said to be proved under the statements of the specification; when P is true, the conjecture establishes a theorem of the specification.

To demonstrate the consistency of state space descriptions, one is required to prove initialisation conjectures, to show that there is at least one valid instance of the state space description satisfying its initialisation (an existence proof). For example, the initialisation conjecture for *Order* (intensional view, above) is: $\vdash ?\ \exists \textit{OrderInit} \bullet \text{true}$.

We prove initialisation conjectures for the various system components and the whole system:

- the initialisation conjectures of class intensions are simplified by appeal to *UML*$+$*Z* meta-theorems, and they are then automatically proved in Z/Eves;
- the initialisation conjectures of class extensions, associations and system are *true by construction*, by appeal to meta-theorems of *UML* $+$ *Z*.

5.2.5 Validating the model

Question 7: How can we be confident that the model expresses the intent of our customer?

Answer: We need to check the model against the requirements of the system. A model may be consistent and still not meet the system requirements. We have developed a technique, based on Catalysis snapshots [D'S 99] and formal proof, to validate the models of our framework [AMA 04]; this validation is assisted by Z/Eves.

Catalysis snapshots are UML object diagrams. These diagram are instances of class diagrams, describing the objects of a system and the way they are linked among each other at a point in time. The use of diagrams helps to involve the customer of the system in model validation, by drawing diagrams that illustrate the system's requirements.

Question 8: Obviously, we can't validate everything, but how would we know that an order really can only reference one product? Does the system accept a situation where two products are linked to one order?

Answer: The snapshot in Figure 5.4 is used to validate this requirement. It shows a state that should not be accepted by the model of the system: an order that refers to two products.

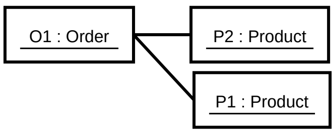

Figure 5.4: A snapshot showing an Order associated with two products

The Z representation of this snapshot is fully generated by template instantiation:

$$\begin{array}{|l}
\textit{StSnap1} \\
\hline
\textit{System} \\
\hline
\textit{orders} = \{oO1\} \wedge \textit{orderSt} = \{oO1 \mapsto O1\} \\
\textit{products} = \{oPX, oPY\} \wedge \textit{productSt} = \{oPX \mapsto PX, oPY \mapsto PY\} \\
\textit{references} = \{oO1 \mapsto oPX, oO1 \mapsto oPY\} \\
\hline
\end{array}$$

(Some definitions are omitted, such as names and states of objects.)

The state described by the snapshot should not be accepted by the system, thus we prove the conjecture (the negation of the positive case):

$$\vdash ?\ \neg\ (\exists\ \textit{StSnap1} \bullet \text{true})$$

This conjecture is provable in Z/Eves, which means that the state described by the snapshot is not valid in the model of the system.

5.3 Analysis and specification of case 2

The second case study does not change the state space, but it adds external behavior, and modifies internal behavior. Case 2 requires that the system provide the following operations:

- entries of new orders;
- cancellation of orders;
- entries of quantities in the stock.

For each of these operations in turn, we discuss its specification, then review its verification and validation. The existing generated Z model is unchanged.

5.3.1 Entries of new orders

Question 9: What must the system do when an order is received? What are the changes to the system?

Answer: ¿From the case study we deduce that the system creates the order and associates it to the ordered product. Then, if there is enough stock, the order is set to *invoiced*, otherwise it is set to *pending*. This involves the following component operations:

1. create an *Order* object (state is set to *pending*);
2. create the tuple linking the new order with the ordered product;
3. set the state of the *Order* to *invoiced* if there is enough stock.

First, we specify component operations individually, and then their composition. The first and third operations have already been specified for *Order* ($\mathbb{S}_{\vartriangle}OrderNew$ and $\mathbb{S}_{\vartriangle}OrderInvoice$, above). The second operation is defined in the relational view (*fully generated*):

$$\begin{array}{|l}
\hline \mathbb{A}_{\vartriangle}ReferencesAdd \\
\Delta\ \mathbb{A}References \\
oOrder? : \mathbb{O}OrderCl \\
oProduct? : \mathbb{O}ProductCl \\
\hline
references' = references \cup \{oOrder? \mapsto oProduct?\} \\
\hline
\end{array}$$

The system operation is defined as the composition of the component operations. This is *partially generated*. To simplify matters, the operation specification is divided in two parts: (a) create the object and tuple in the association references; (b) set the order to invoiced if there is sufficient stock. The two parts are then combined using the schema composition operator ($\fatsemi$) [WOO 96], which means that part (a) is followed by part (b).

In part (a), we start by defining the frame of the operation, which makes explicit what is to change and what is to remain unchanged. The name of system operation frames are prefixed by Ψ (by analogy to Φ promotion frames), and are formed by conjoining Δ *System* with the Ξ (nothing changes) of every system component whose state is to remain unchanged. In the system operation to create new orders, the $\mathbb{S}$*Product* component should remain unchanged (there are no changes to products):

$$\Psi \mathit{NewOrder}_a == \Delta \mathit{System} \land \Xi \mathbb{S}\mathit{Product}$$

The system operation is specified as the conjunction of the frame and the component operations; the renaming is needed so that elements of the schemas correctly communicate across the composition:

$$\begin{array}{l} \mathit{SysNewOrder}_a == \Psi \mathit{NewOrder}_a \\ \quad \land\ \mathbb{S}_{\Delta}\mathit{OrderNew} \land \mathbb{A}_{\Delta}\mathit{ReferencesAdd}[\mathit{oOrder}!/\mathit{oOrder}?] \end{array}$$

In part (b), we need a schema stating the condition on the state transition *pending* to *invoiced*, which says that an order may change to *invoiced* only if the ordered quantity is less than or equal to the stock for the ordered product:

$$\begin{array}{|l} \hline \mathit{CondStockIsAvailable} \\ \mathbb{S}\mathit{Product} \\ \mathit{oProduct}? : \mathbb{O}\mathit{ProductCl} \\ \mathit{quantity}? : \mathbb{N} \\ \hline \mathit{oProduct}? \in \mathit{products} \land \mathit{quantity}? \leq (\mathit{productSt}\ \mathit{oProduct}?).\mathit{stock} \\ \hline \end{array}$$

The frame of part (b) says that only the *Order* component may change. The actual operation says that if there is enough stock to fulfill the order then it should be invoiced, otherwise there is no change in the system:

$$\Psi \mathit{NewOrder}_b == \Delta \mathit{System} \land \Xi \mathbb{S}\mathit{Product} \land \Xi \mathbb{A}\mathit{References}$$

$$\begin{array}{l} \mathit{SysNewOrder}_b == \Psi \mathit{NewOrder}_b \\ \quad \land\ \mathit{CondStockIsAvailable} \land \mathbb{S}_{\Delta}\mathit{OrderInvoice}[\mathit{oOrder}!/\mathit{oOrder}?] \\ \quad \lor\ \neg\ \mathit{CondStockIsAvailable} \land \Xi \mathit{System} \end{array}$$

Finally, the two parts are put together to make the system operation:

$$\mathit{SysNewOrder} == \mathit{SysNewOrder}_a \fatsemi \mathit{SysNewOrder}_b$$

Question 10: Is the operation consistent? What is its precondition?

Answer: The precondition of a Z operation describes the sets of states for which the outcome of the operation is properly defined. An operation is consistent (or satisfiable) if its precondition is not *false*. The precondition of the new operation is:

$$\mathbb{O}OrderCl \setminus orders \neq \varnothing \wedge oProduct? \in products$$

This precondition is not *false*. It states that the system has capacity for another *Order* object and that the ordered product is an existing product. This is exactly what we expect the precondition to be.

Question 11: Can we examine the model with a snapshot for the case that there is enough stock to fulfill an order?

Answer: We can write snapshots that describe state transitions. These snapshots are divided in two parts: the system state before an operation, and the state of the system after the operation.

Figure 5.5 shows a before state at the top (there is one product *PX*); the values of all the attributes are shown in the object boxes. The required after state of the first running of the operation is in the middle – there is now one order for *PX*. Order *O1* is for a *quantity* less than product *PX*'s quantity of *stock*, so the *state* of *O1* is *invoiced*. This is also the before state for another running of the operation, which adds *O2*, another order whose *quantity* is less than product *PX*'s quantity of *stock*, so its *state* is also *invoiced*.

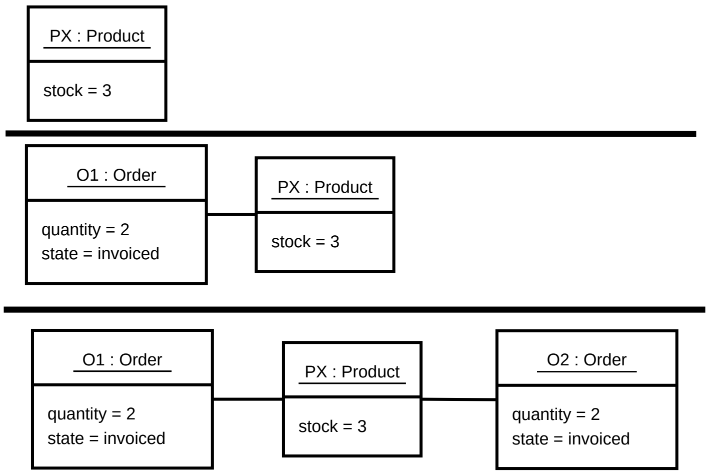

Figure 5.5: *SysNewOrder* snapshot: request for a product with enough stock

To check the validity of the operation snapshot, we represent the states in Z and we prove conjectures that perform three types of checks. The first checks that (a) the before-state is a valid system state and (b) it satisfies the operation precondition, an existence proof. The second checks that the after-state is a valid system state. The third checks that the operation satisfies the constraints described by the snapshot. These conjecture are captured by templates (see [AMA 06, AMA 04] for full details).

The conjectures for the snapshot in Figure 5.5 are all provable by Z/Eves.

Question 12: But how can we keep placing orders against product *PX*? Why is the stock not running out?

Answer: Here, the visualisation highlights a problem with the model. The case study does not say anything about what happens to *stock* when an order is performed. We assume that the stock is subtracted the ordered *quantity* each time.

Question 13: What do we need to do to the models?

Answer: First, we change the snapshot to describe what we want the system to do. Figure 5.6 shows the snapshot for the corrected second running of the operation: now the ordered amount is subtracted from the stock.

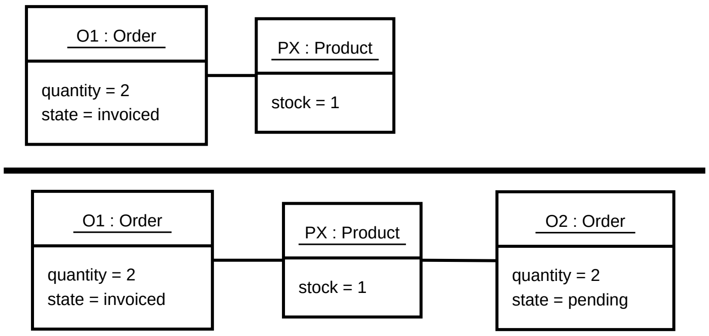

Figure 5.6: Revised *SysNewOrder* snapshot with deletion of quantity from stock

As expected, this snapshot fails to validate. The first two checks (above) are still true, since the before and after states are valid states of the system, and the before state is a valid precondition state for the operation. The third conjecture, which looks at whether the operation actually changes the before state into the after state, is false (its negation is *true*) – the operation does not perform the required transition. The Z operation specification needs to be corrected so that stock subtraction is performed.

We specify an operation in the intension of *Product* to subtract an input quantity (*quantity*?) from the product *stock*:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Product}_{\triangle}\textit{StockSubtract}} \\
\hline
\Delta \mathit{Product} \\
\mathit{quantity}? : \mathbb{N} \\
\hline
\mathit{stock}' = \mathit{stock} - \mathit{quantity}? \\
\hline
\end{array}$$

As before, the operation is promoted in the extensional view (*fully generated*):

$$\begin{array}{l}
\mathbb{S}_{\triangle}\mathit{ProductStockSubtract} == \exists\ \Delta \mathit{Product} \bullet \\
\qquad \Phi\mathbb{S}\mathit{ProductUpdate} \wedge \mathit{Product}_{\triangle}\mathit{StockSubtract}
\end{array}$$

The original *SysNewOrder* has two components: the creation, then the invoicing. The stock change only applies to orders that can be invoiced. The calculated precondition of the new operation now captures the precondition previously expressed in the schema *CondStockIsAvailable*, so this is no longer required. The new version is:

$$\begin{array}{l}
\Psi \mathit{NewOrder}_b == \Delta \mathit{System} \wedge \Xi\mathbb{A}\mathit{References} \\
\mathit{SysNewOrder}_b == \Psi \mathit{NewOrder}_b \\
\qquad \wedge\ \mathbb{S}_{\triangle}\mathit{OrderInvoice}[\mathit{oOrder}!/\mathit{oOrder}?] \wedge \mathbb{S}_{\triangle}\mathit{ProductStockSubtract} \\
\qquad \vee\ \neg\ \mathrm{pre}\ \mathbb{S}_{\triangle}\mathit{ProductStockSubtract} \wedge \Xi \mathit{System} \\
\mathit{SysNewOrder} == \mathit{SysNewOrder}_a \fatsemi \mathit{SysNewOrder}_b
\end{array}$$

Question 14: Is the new version of the operation consistent? What is its precondition? And is the operation snapshot valid now?

Answer: The precondition of the operation remains the same, which is what we want. The validation has to be repeated to ensure that the snapshots are still valid. In fact all the necessary proofs are now true; the system behaves as the customer wishes.

5.3.2 Cancellation of orders

Question 15: What happens when an order is cancelled? Is the data relating to the cancelled order retained in the system?

Answer: The case study just says that orders may be cancelled. We assume that cancelled orders are deleted from the system. (The alternative, recording cancelled orders, would require the addition of a *cancelled* state to the state diagram of *Order*.)

Question 16: Can an invoiced order be cancelled?

Answer: The case study is not clear on restrictions on cancellation. Here, we assume that only *pending* orders can be cancelled. (The cancelling of *invoiced* orders would require that the ordered quantity be placed back into the stock.)

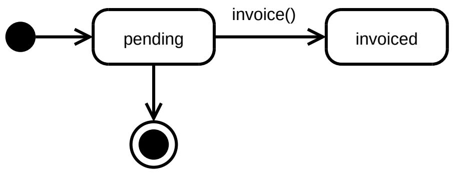

Figure 5.7: The updated state diagram of *Order*

To express cancellation we need to change the state diagram of *Order*, by adding an arrow from the state *pending* to the terminal state (Figure 5.7). This means that the object is deleted only when in the *pending* state.

This new arrow requires that more Z is *fully generated* from templates. In the intension of *Order*, finalisation captures the fact that *Order* objects may be deleted only if in the *pending* state:

$$OrderFin == [\, Order \mid state = pending \,]$$

The finalisation is then promoted in the extensional view. The delete promotion frame for *Order* removes an object from the class extension:

$$\begin{array}{|l}
\hline
\Phi\mathbb{S}OrderDelete \\
\hline
\Delta\mathbb{S}Order \\
Order \\
oOrder? : \mathbb{O}OrderCl \\
\hline
oOrder? \in orders \\
\theta Order = orderSt\ oOrder? \\
orders' = orders \setminus \{oOrder?\} \\
orderSt' = \{oOrder?\} \ntriangleleft orderSt \\
\hline
\end{array}$$

The operation that deletes *Order* objects uses the frame to promote the finalisation:

$$\mathbb{S}_{\Delta}OrderDelete == \exists\, Order \bullet \Phi\mathbb{S}OrderDelete \wedge OrderDelCond$$

Question 17: What must the system do when an order is received? What are the changes to the system?

Answer: ¿From the case study and our assumptions, we deduce that the system must delete the *Order* object and its tuple in the association *References*. This involves two component operations: deletion of the order object (above) and deletion of tuple; both are *fully generated*.

The association operation deletes the tuple from the association, given an *Order* object as input:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{A}_{\vartriangle}\mathit{ReferencesDelOrder}} \\
\Delta\ \mathbb{A}\mathit{References} \\
\mathit{oOrder}? : \mathbb{O}\mathit{OrderCl} \\
\hline
\mathit{references}' = \{\mathit{oOrder}?\} \ndres \mathit{references}
\end{array}$$

The system operation to cancel orders conjoins its frame with the component operations:

$$\begin{aligned}
&\Psi \mathit{CancelOrder} == \Delta \mathit{System} \land \Xi \mathbb{S}\mathit{Product} \\
&\mathit{SysCancelOrder} == \Psi \mathit{CancelOrder} \land \mathbb{S}_{\vartriangle}\mathit{OrderDelete} \land \\
&\qquad\qquad\qquad\qquad \mathbb{A}_{\vartriangle}\mathit{ReferencesDelOrder}
\end{aligned}$$

Question 18: Is the operation consistent? What is its precondition?

Answer: The precondition predicate (calculated with Z/Eves) is:

$$\mathit{oOrder}? \in \mathit{orders} \land (\mathit{orderSt}\ \mathit{oOrder}?).\mathit{state} = \mathit{pending}$$

The operation is satisfiable, its precondition requires that the order to be cancelled is an existing order and that its *state* is *pending*, as expected.

Question 19: Does the order cancellation operation do what we expect?

Answer: Yes. The operation has been validated. We do not show the validation snapshots here, but we tested the case where the order to delete is *pending* (conjectures were all true) and the case where the order to delete is *invoiced* (one conjecture was false, as expected).

5.3.3 Entries of quantities into stock

Question 20: What is an "entry of quantities in the stock"? Does the required operation have to (a) just add stock, as in stock delivery or return; (b) also subtract stock, as in stock decay or wastage; or (c) accommodate arbitrary re-setting of the stock, as in stock-taking?

Answer: The case study does not elaborate on the meaning of stock entry. We assume that it adds an input quantity to the *stock* attribute.

The modeler needs to specify an operation to add stock, on the intension of *Product*. The operation receives a quantity of stock as input and adds this quantity to the existing stock:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Product}_{\vartriangle}\mathit{StockAdd}} \\
\Delta \mathit{Product} \\
\mathit{nStock}? : \mathbb{N} \\
\hline
\mathit{stock}' = \mathit{stock} + \mathit{nStock}?
\end{array}$$

This operation is promoted in the extension of *Product* (fully generated):

$$\mathbb{S}_{\Delta}ProductStockAdd == \exists\ \Delta Product \bullet$$
$$\quad \Phi\mathbb{S}ProductUpdate \land Product_{\Delta}StockAdd$$

The system operation simply puts the promoted operation into the context of the system:

$$\Psi AddStock == \Delta System \land \Xi\mathbb{S}Order \land \Xi\mathbb{A}References$$
$$SysAddStock == \Psi AddStock \land \mathbb{S}_{\Delta}ProductStockAdd$$

Question 21: Is the operation consistent? What is its precondition?

Answer: The precondition (calculated in Z/Eves) is:

$$oProduct? \in products$$

The operation is consistent, its precondition requires that the product to which stock is to be added is an existing product.

Question 22: Can we validate this operation with a snapshot?

Answer: We draw a snapshot illustrating the addition of stock to a product (Figure 5.8). A quantity of eight is added to the *stock* of *PY*, but there is an order *pending* with a *quantity* of three. This snapshot is valid in our model (conjectures proved in Z/Eves), but this poses a question.

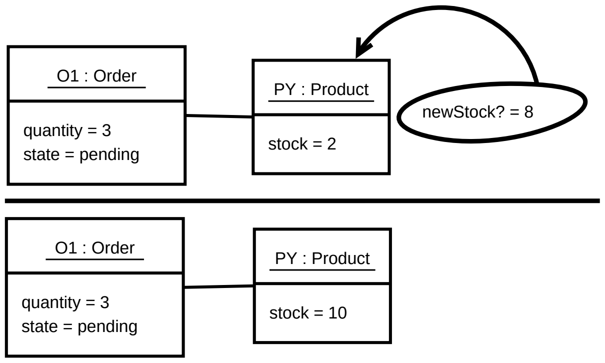

Figure 5.8: Snapshot for *Add Stock*: *Order* remains *pending*

Question 23: What happens when stock is added to products with pending orders? For example, in the snapshot, after the addition, there is enough stock to fulfill the order $O1$? Should $O1$ be changed to *invoiced* and stock subtracted the ordered amount?

Answer: The case study does not say what happens here. We assume that the pending order should be set to invoiced in this case. The snapshot should be corrected to reflect this: the after state of $O1$ would be *invoiced*, and *stock* value would be adjusted by $8 - 3$. This snapshot is valid in our model (proved in Z/Eves).

Question 24: But what if there were more than one order *pending* on *PY*?

Answer: Again, the case study is omissive. We assume that any order or orders are invoiced, until there is insufficient stock. We also decide not to impose any *ordering* to fulfill orders that are pending. This is is illustrated in Figures 5.9 and 5.10 – either of these diagrams can represent the effect of adding eight elements to the stock of *PY*. We model the fulfillment of orders after the addition of stock non-deterministically, which allows the developer to refine the model to enforce any ordering later in the development.

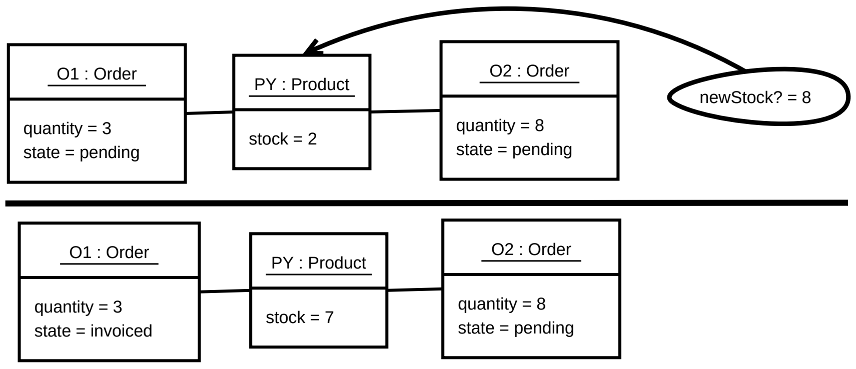

Figure 5.9: Snapshot for *Add Stock*: stock is increased and only one of the pending orders is invoiced (option 1)

Currently, the operation *SysAddStock* specified above does not change the state of *Order* objects, it just adds to the stock. We need to follow the same pattern used for the *New Order* operation: the specification is divided in two parts that are put together using sequential composition. These two parts are: (a) add the new quantity to the product's stock and (b) update the orders that are pending on the product until there is insufficient stock remaining for any more, and reduce the stock by the sum of the quantities of all orders that have been invoiced.

Above, we have specified part (a), so we just need to add part (b). First, we specify the component operation that updates a set of orders in the extension of *Order*. This needs to invoice a set of *Order* objects, which is defined using *multi-promotion* [STE 03][1]. $Order_{\triangle}Invoice$ is promoted to be executed on a set of *Order* objects (fully generated):

[1]Multi-promotion promotes operations on a set of objects, rather than a single object.

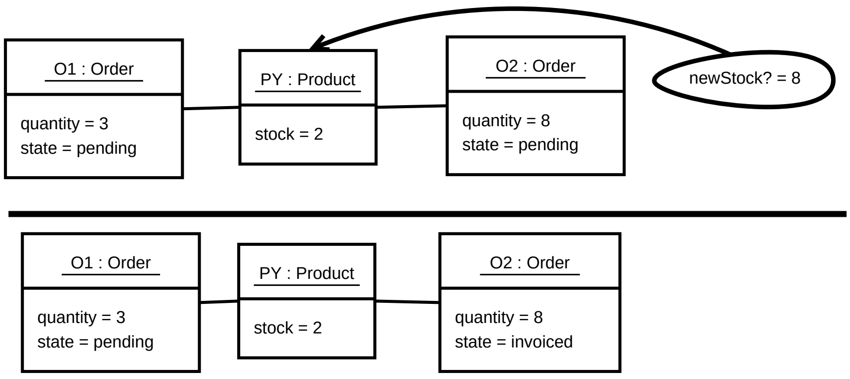

Figure 5.10: Snapshot for *Add Stock*: stock is increased and only one of the pending orders is invoiced (option 2)

$$\begin{array}{|l}
\mathbb{S}_{\triangle}\mathit{OrderInvoiceSet} \\
\hline
\Delta\mathbb{S}\mathit{Order} \\
\mathit{osOrder}? : \mathbb{P}(\mathbb{O}\mathit{OrderCl}) \\
\hline
\mathit{osOrder}? \subseteq \mathit{orders} \\
\mathit{orders} = \mathit{orders}' \\
\mathit{osOrder}? \lhd \mathit{orderSt} = \mathit{osOrder}? \lhd \mathit{orderSt}' \\
\forall o : \mathit{osOrder}? \bullet \exists \Delta\mathit{Order} \bullet \\
\quad \mathit{orderSt}\ o = \theta\mathit{Order} \land \mathit{orderSt}'\ o = \theta\mathit{Order}' \land \mathit{Order}_{\triangle}\mathit{Invoice} \\
\hline
\end{array}$$

The part (b) system operation is defined by composing component operations. First, we need a schema expressing the precondition for the operation; invoicing starts if there is at least one pending order on the product that can be fulfilled:

$$\begin{array}{|l}
\mathit{PreCondAddOrders} \\
\hline
\mathbb{S}\mathit{Order};\mathbb{S}\mathit{Product};\mathbb{A}\mathit{References} \\
\mathit{oProduct}? : \mathbb{O}\mathit{ProductCl} \\
\hline
\exists o : \mathit{references}^{\sim}(\!|\ \{\mathit{oProduct}?\}\ |\!) \bullet \\
\quad (\mathit{orderSt}\ o).\mathit{state} = \mathit{pending} \\
\quad \land (\mathit{orderSt}\ o).\mathit{quantity} \leq (\mathit{productSt}\ \mathit{oProduct}?).\mathit{stock} \\
\hline
\end{array}$$

We have chosen to specify this operation non-deterministically, so we express the desired postcondition of the operation: after the operation is executed, there are no pending orders on the product that could be fulfilled:

$$\begin{array}{|l}
\textit{PostCondAddOrders} \\
\hline
\mathbb{S}\mathit{Order}';\mathbb{S}\mathit{Product}';\mathbb{A}\mathit{References}' \\
\mathit{oProduct}? : \mathbb{O}\mathit{ProductCl} \\
\hline
\forall\, o : \mathit{references}'^{\sim}(\!|\ \{\mathit{oProduct}?\}\ |\!) \mid (\mathit{orderSt}'\ o).\mathit{state} = \mathit{pending} \bullet \\
\qquad (\mathit{orderSt}'\ o).\mathit{quantity} > (\mathit{productSt}'\ \mathit{oProduct}?).\mathit{stock} \\
\hline
\end{array}$$

Next, we define a connector for use in the composition. This says that the set of *Order*s to invoice (*osOrder*?) is a subset of all the orders that are pending on the updated product and that can be fulfilled (it is non-deterministic), and that the *quantity*? to subtract from stock is the sum of the quantities of all orders that are to be invoiced:

$$\begin{array}{|l}
\textit{Connector} \\
\hline
\mathbb{S}\mathit{Order};\ \mathbb{A}\mathit{References};\ \mathbb{S}\mathit{Product} \\
\mathit{oProduct}? : \mathbb{O}\mathit{ProductCl} \\
\mathit{quantity}? : \mathbb{N} \\
\mathit{osOrder}? : \mathbb{P}(\mathbb{O}\mathit{OrderCl}) \\
\hline
\mathit{osOrder}? \subseteq \{\ o : \mathit{references}^{\sim}(\!|\ \{\mathit{oProduct}?\}\ |\!) \mid \\
\qquad\qquad (\mathit{orderSt}\ o).\mathit{state} = \mathit{pending} \\
\qquad\qquad \wedge\ (\mathit{orderSt}\ o).\mathit{quantity} \leq (\mathit{productSt}\ \mathit{oProduct}?).\mathit{stock}\ \} \\
\mathit{quantity}? = \Sigma\ \{\ o : \mathit{osOrder}? \bullet (o \mapsto (\mathit{orderSt}\ o).\mathit{quantity})\ \} \\
\hline
\end{array}$$

This uses the generic operator Σ from the Z toolkit of $\mathit{UML} + Z$:

$$\begin{array}{|l}
[L] \\
\hline
\Sigma : (L \mathrel{+\!\!\!\!\to} \mathbb{Z}) \to \mathbb{Z} \\
\hline
\Sigma\ \varnothing = 0 \\
\forall\, l : L;\ n : \mathbb{Z};\ S : L \mathrel{+\!\!\!\!\to} \mathbb{Z} \mid l \notin \operatorname{dom} S \bullet \Sigma\ (\{l \mapsto n\} \cup S) = n + \Sigma\ S \\
\hline
\end{array}$$

The part (b) operation $\mathit{SysAddStock}_b$ is the composition of all these definitions: the operation says that if the precondition schema (*PreCondAddOrders*) is *true*, then orders are invoiced, otherwise nothing changes. The system operation, *SysAddStock*, composes the part (a) and part (b) specifications using sequential composition:

$$\begin{array}{l}
\Psi\mathit{AddStock}_b == \Delta\mathit{System} \wedge \Xi\mathbb{A}\mathit{References} \\
\mathit{SysAddStock}_b == (\Psi\mathit{AddStock}_b\ \wedge \\
\qquad \mathit{PreCondAddOrders} \wedge \mathbb{S}_{\Delta}\mathit{OrderInvoiceSet} \\
\qquad\qquad \wedge\ \mathit{Connector} \wedge \mathbb{S}_{\Delta}\mathit{ProductStockSubtract} \wedge \mathit{PostCondAddOrders} \\
\qquad \vee\ \neg\ \mathit{PreCondAddOrders} \wedge \Xi\mathit{System}) \setminus (\mathit{osOrder}?, \mathit{quantity}?) \\
\mathit{SysAddStock} == \mathit{SysAddStock}_a \mathbin{⨾} \mathit{SysAddStock}_b
\end{array}$$

Question 25: Is the new version of the operation consistent? What is its precondition? Are the snapshots valid in this new version?

Answer: The precondition remains the same, as expected. The validation proofs for the snapshots are provable in Z/Eves.

5.4 Natural language description of the specification

5.4.1 Case 1

An order references one product and a product may be referenced by zero or more orders. Orders may be in one of two states, *pending* or *invoiced*, depending on whether they are waiting to be fulfilled or if they have been fulfilled and invoiced; orders have a quantity, which is a positive whole number. An *Order* has the operation *invoice*, which changes the state of the *Order* from *pending* into *invoiced*. In the initial state of the system there are no orders and no products.

5.4.2 Case 2

Case 2 extends case 1 with three system operations to enter an order, to cancel an order, and to add stock.

The order entry operation stores the order as a new object, links it to the ordered product, and checks whether the order can be invoiced (in the same way as for case 1). If an order can be invoiced, then the order *quantity* is subtracted from the product *stock*, and the order *state* is set to *invoiced*. If not, then the product is unchanged, and the order *state* is set to *pending*.

Order cancellation applies only to orders in the *pending* state. The operation removes the order and the link between the cancelled order and the ordered product.

Entries of stock are explicitly additions to the product *stock*. When stock is received for a known product, it is added to the existing stock. The *pending* orders linked to that product are checked, and they are invoiced until no more orders can be met. There is no imposed ordering on this process, and no requirement to invoice as many orders as possible, but the non-deterministic ordering would allow such conditions to be added by refinement if required.

5.5 Conclusion

Our approach has produced a combined object-oriented and formal model of the system. Some aspects of the case studies were clarified simply by expressing the requirements in UML diagrams; the underlying Z representation of the UML diagrams forced us to be clear and precise in drawing UML diagrams. Many requirements were clarified through snapshot-based validation.

The Z model gives the precise meaning of the UML diagrams and enables formal verification and validation of UML-based models. We would like make the Z hidden to the user as much as possible, but this could not be fully achieved. At least one expert is required to write Z operation specifications and invariants that are not expressible diagrammatically. Nevertheless, the $UML + Z$ framework offers the benefits of formal development, whilst allowing people who are not Z experts to engage in the modeling and analysis effort, by drawing class, state and object diagram.

An important feature of our approach is templates. The templates of the $UML + Z$ catalogue are expressed in FTL [AMA 06]. All Z in this chapter is generated by instantiating templates from the catalogue. In most cases, the instantiation was fully generated from the diagrams; a few times, the developer needed to add extra information. The meta-theorems of the catalogue reduce the proof effort associated with checking the consistency of the $UML + Z$ model; sometimes no proof at all was required (consistency conjectures were *true by construction*); at other times the meta-theorems simplified the proof to a point where it could be easily discharged in Z/Eves.

Our illustration of $UML + Z$ validation has shown that errors in models can be found: (a) because the model does not capture what the developer intended; (b) because the model is inconsistent; (c) because the developer's intention is consistently expressed in the model but is not what the client wanted. The use of both diagrammatic and formalised snapshots is shown to be necessary to extract the various failings of the models.

Acknowledgements

Nuno Amálio is funded by the Portuguese Foundation for Science and Technology under grant 6904/2001.

Bibliography

[AMA 04] AMÁLIO N., STEPNEY S., POLACK F., "Formal Proof From UML Models", in DAVIES J. *et al.*, Eds., *ICFEM 2004*, vol. 3308 of *LNCS*, Springer, p. 418–433, 2004.

[AMA 05] AMÁLIO N., POLACK F., STEPNEY S., "An Object-Oriented Structuring for Z based on Views", in TREHARNE H. *et al.*, Eds., *ZB 2005: International Conference of B and Z Users*, vol. 3455 of *LNCS*, Springer, p. 262–278, 2005.

[AMA 06] AMÁLIO N., Frameworks based on templates for rigorous model-driven development, PhD thesis, Department of Computer Science, University of York, 2006.

[D'S 99] D'SOUZA D. F., WILLS A. C., *Objects, Components and Frameworks in UML: The Catalysis Approach*, Addison-Wesley, 1999.

[SAA 97] SAALTINK M., "The Z/EVES system", in *ZUM'97, Reading, UK*, vol. 1212 of *LNCS*, Springer, 1997.

[STE 03] STEPNEY S., POLACK F., TOYN I., "Patterns to Guide Practical Refactoring: Examples Targeting Promotion in Z", in BERT D. *et al.*, Eds., *ZB 2003, Turku, Finland*, vol. 2651 of *LNCS*, Springer, p. 20–39, 2003.

[WOO 96] WOODCOCK J., DAVIES J., *Using Z: Specification, Refinement, and Proof*, Prentice-Hall, 1996.