Applying Fusion/UML to the Invoice Problem

R. B. France
Department of Computer Science & Engineering
Florida Atlantic University, Boca Raton, Florida, USA

J-M. Bruel
University of Pau, France

Abstract

In this paper, we use the Fusion Object-Oriented (OO) analysis method to create graphical OO models of the Invoicing Case Study proposed by Henri Habrias. The OO models are expressed in the Unified Modeling Language (UML). The UML models we produced were analyzed by transforming them to Z specifications.

1 Introduction

In this paper we use an integrated informal Object-Oriented (OO) and formal modeling technique to model and analyze the Invoicing case study proposed for the Invoice'98 Workshop. The technique is a result of research carried out by the *Methods Integration Research Group* (MIRG) at Florida Atalnatic University.

Figure 1 outlines the process we followed in developing the OO models. The *Informal OO Modeling* activity involved developing graphical OO models according to the Fusion OO analysis process [2]. Rather than use the (out-dated) notation presented in [2], we used the newer Unified Modeling Language (UML) [8] where possible.

The *Formalization* activity involved producing Z characterizations of the semantics captured by the OO models. We used the latest variant of the Fusion-to-Z transformation rules we developed as part of research on integrated techniques. Specifically, the rules used are a modification of the rules presented in [4]. The formalization was done by hand, though some aspects are mechanizable. An early version of the transformation rules were implemented in a tool called FuZE. Unfortunately, FuZE has not yet been

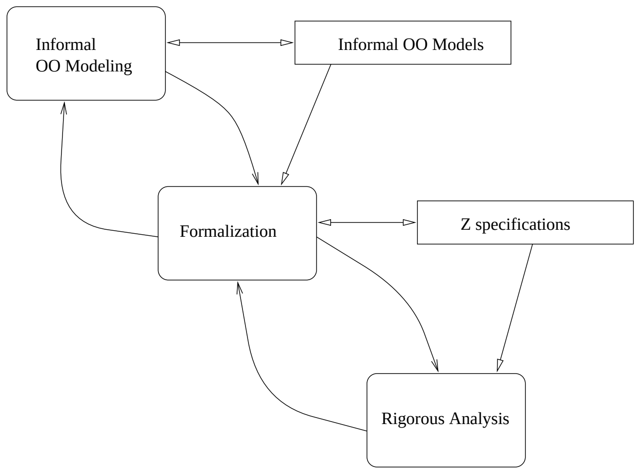

Figure 1: Model Development Process

updated with the new rules so we could not use the tool. On the other hand, this exercise provided insight into the effectiveness of the rules that can only be gained through manual application. Typically, formalization can uncover problems related to missing information and ambiguous structures. Addressing these problems may involve changing the graphical OO models, as reflected by the arrow going from the formalization activity to the informal modeling activity.

Analysis involved discharging the typical proof obligations associated with Z specifications. One can also use Z/EVES and the Z animator ZANS in this phase.

This paper describes our experiences with applying the above process to the invoicing problem. Section 2 gives an overview of the notations and techniques used. In Section 3 we describe our experiences with applying the integrated technique to Case 1 and Case 2 of the invoicing problem. We conclude in Section 4 with an overview of our experiences.

2 Background

2.1 The Fusion/UML Method

Fusion is an object-oriented software development methodology that combines and extends existing techniques, e.g., Rumbaugh's Objet Model Technique (OMT) [11], Booch's technique [1], Wirfs-Brock's Class Responsibility

Collaborator [14] (CRC) technique, and Jacobson's Objectory [10]. In this section we present an overview of the Fusion analysis process and products.

In Fusion's analysis phase the required behavior of the system is described by:

- a *Class Diagram* that defines the static structure of the information manipulated by the application in terms of classes and the relationships among them; and
- an *Interface Model* that defines the externally observable behavior of the application. The Interface Model, in turn, consists of two models:
  - An *Operation Model* that characterizes the effect of application services in terms of the observable state changes they make and the output events they send to the environment. The services identified in the interface model are not associated with classes at the analysis level.
  - A *Life-Cycle Model* that characterizes the allowable sequences of service invocations for the application.

A Class Diagram identifies the classes that are pertinent to the problem. The properties of each class are expressed in terms of data attributes and associations (references to other classes). The result is a graph in which classes are nodes and associations are edges. We use the UML notation to express Class Diagrams in this case study.

The Fusion Operation Model consists of a set of schemas, each describing the effects of a system-level operation. Each operation schema consists of the following parts:

Description: an informal and concise description of the operation.

Reads: a list of the items that the operation reads.

Changes: a list of the items that the operation may change.

Sends: a list of the events that the operation may send to the environment.

Assumes: a condition that describes what is assumed true at the start of the operation.

Result: describes what is true after the operation has completed its e xecution.

The lifecycle models for the case studies are not interesting so they are not presented in this paper.

2.2 Integrated Fusion/Z technique

Our work on integrating FSTs with less formal graphical modeling techniques (e.g., see [6, 7]) is based on our observation that formal and informal models can play complementary roles in software development. Formal specification techniques (FSTs) can enhance the application of less formal OO modeling techniques (OOMTs). specifically, FSTs can be used to reinforce and/or instill confidence in, and to uncover problems with the less formal OO models.

Informal OO models can, in turn, enhance the application of FSTs. Models produced by FSTs, while abstract, precise and analyzable, may not be as easy to understand as informal models. In this respect, informal models can be used to depict the behavior and structure expressed in formal models in a more understandable manner. Also, graphical OOMTs provide a rich set of structuring mechanisms that can be used as the basis for creating well-structured formal models.

One approach to making OOMTs more precise and amenable to rigorous analysis is to integrate them with suitable formal specification notations that are supported by analysis mechanisms (e.g., see [3]). The intent is to obtain formal specifications from OO models that are analyzable. Semantic analysis then takes the form of rigorous analysis of the formal specifications.

Our early work in this area resulted in a set of rules for translating Fusion analysis models [2] to Z specifications. These rules are implemented in a prototype tool called FuZE (Fusion/Z Environment) [3]. FuZE is an extension of a commercial OO CASE tool that provides support for the automated generation of Z specifications from Fusion analysis models. In this paper we use the latest version of the rules (not yet implemented in FuZE). The rules are outlined in the Appendix.

2.3 The Invoicing Problem

The following is a distillation of the Invoicing requirements statement.

- An order consists of an order for one and only one product. The same product and quantity can appear on more than one order.

- An order is said to be invoiced if the quantity ordered is less than or equal to the quantity stocked for ordered product, an order is pending if it is not cancelled or invoiced.

- CASE 1: The ordered product is in stock. In this case the invoicing operation assumes that the stock and the set of orders is up-to-date.

Adding new products and orders, and cancelling orders is outside the scope of the system.

- CASE 2: Adding new products and orders, and cancelling orders are within the scope of the system.

3 Formalized Fusion/UML Models

In this section we present our Fusion/UML requirements models for Case 1 and Case 2 of the Invoicing problem, and outline the rationale underlying the models. We also present the questions raised during the modeling activity.

3.1 Case 1

Our analysis of the requirements statement resulted in the identification of the following classes of objects (see Figure 2):

Order: The properties of an order that seemed relevant are invoicing status and the order item. We modeled the invoicing status as a data attribute that can have two values: *pending* (not invoiced) and *invoiced*. The order item property consists of two subproperties, a reference to a product and the quantity ordered. Initially we modeled the order item property as two attributes, one representing the product reference and the other representing the quantity ordered. While this approach works, it makes it difficult to incorporate changes that involve increasing the number of order items per order beyond one. As it is good practice to create requirements models that facilitate change, we later decided to model the order item as a separate object that is an integral part of an order. The result is that an order is modeled as a composite that consists of one part, an *order_item*. The quantity property of an order_item is modeled as a data attribute of order_item, and the product reference property is modeled as an association, called *orders*, between the order_item and a product record (see Product_Info below). Alternatively, the product reference could be modeled as a data attribute, but this is a less direct way of stating that the order_item is related to a product. An instance of an order_item must belong to a unique order; order_items cannot be shared. Note that this does not mean that two orders cannot order the same quantity of a product. The quantity and product references are properties that can be varied, thus two distinct order_items can have the same property values.

Product_Info: In this system, information about products is manipulated, rather than the actual (physical) products. An instance of class *Product_Info* contains information about products of a particular type. We abstract over such information by modeling it as a single data attribute called *prod_desc* (for product description).

Stock: We represented the stock as an instance of the class *Stock*. This raised the question as to whether there should one or more stocks. In 'real-world' terms one can think of a stock instance as a wharehouse, and it is possible that the system will need to check stocks at multiple wharehouses. The requirements statement seems to imply that there is only one stock (reference to 'the' stock). It was not clear to us what was really needed, so we decided to model a more general case, of which the single stock system would simply be an instantiation. In our model there are multiple stocks, but an order can only be invoiced against one stock (i.e., it is not possible in our system to draw products for a single order from different stocks). A stock instance has the following properties:

- For each product type held in stock, information about the quantity stocked. This property is modeled as an integral component (i.e., a component that is not shared with other stocks, and lives and dies with the stock) called *Stocked_Product_Info* that has a single attribute *qty* and a reference (the association *stocked*) to the product type.
- References to the orders that have been invoiced against the stock modeled by an association called *invoiced_by*.

The Class Diagram reflecting the above decisions is given in Figure 2.

Invoicing an order is modeled as a system operation in our Fusion model. A question that arose as we specified the operation is whether the stocked quantity is updated to reflect the fact that an order has been invoiced against it. We chose to reflect the change in quantity in our model. Alternatively, one can define a separate operation to do so, and require that this operation be carried out between order invoicing. The Operation Schema for the invoicing system operation is given below:

OPERATION : Invoice_Order

DESCRIPTION : Checks whether an order can be satisfied or not and changes it's status accordingly.

READS :

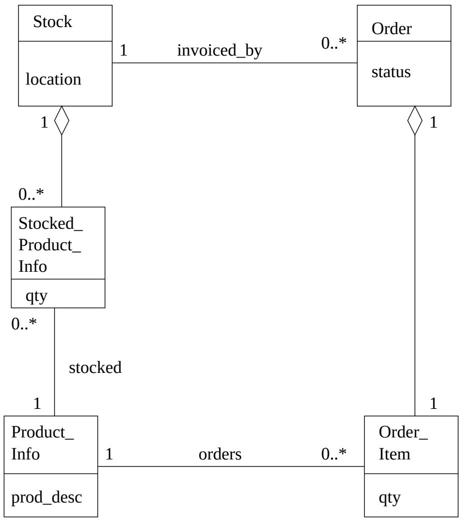

Figure 2: Invoicing System: Case 1

CHANGES : supplied $order : Order$
s : $Stock$ where s is the stock against which order is invoiced.

SENDS :

ASSUMES : $order.status = pending$ and
there exists s : $Stock$; spi : $Stocked\_Product\_Info$ such that
spi is in $s.stocked\_product\_info$ and
$spi.product\_info = order.order\_item.product\_info$

RESULT : if $spi.qty \geq order.order\_item.qty$
then
$order.status = invoiced$ and
$spi.qty = spi.qty - order.order\_item.qty$ and
$(s, order) \in invoiced\_by$
else
order.status = pending

in the above we use the Object Constraint Language [8] to express object properties. For example, the expression $s.stocked\_product\_info$ is the set of $Stocked\_Product\_Info$ objects associated with the stock s, and $order.order\_item.product\_info$

is the $Product\_Info$ object associated with the order item of $order$.

The above operation schema is stated informally and is thus not amenable to rigorous analysis. In order to analyze the models and to obtain a more precise specification we transformed the Fusion models to Z using the rules outlined in an earlier section. The resulting Z specifications are given in the Appendix. With the Z specification we were able to check that the invariants in the Class Diagram were consistent with each other and that the operation preserved the system invariants as expressed in the Class Diagram. In this case, the formalization served to enhance our confidence in the above models.

3.2 Case 2

In Case 2, adding new orders and products, and cancelling orders are within the scope of the system. The only change we made to the Class Diagram was a change in the definition of *status* attribute of order. The status of an order can be pending, invoiced, and cancelled.

We added the following three system operations to the model:

Add_Order : Takes a reference to a product in the system and a quantity and creates an order in the system. The referenced product must have a corresponding $Stocked\_Product\_Info$.

Cancel_Order : If the order status is pending it changes the status to cancelled. If the order is invoiced it is not clear in the requirements statement what should happen. We chose to allow the cancellation. In this case the stock quantity is updated (recall that invoicing an order causes the stock quantity to be reduced) by adding the ordered quantity to its current value, the *invoiced_by* reference is moved, and the status is changed to cancelled.

Add_Stocked_Product : We identified three situations under which products can be added (1) the product has a stock record (i.e., there is a corresponding $Stocked\_Product\_Info$ object), (2) there is no stock record for the product, but there is a corresponding $Product\_Info$ object, and (3) there is no corresponding $Product\_Info$ object for the product. Situations (2) and (3) require creation of the appropriate objects. One can define a single system operation that will handle all the above cases. In this paper we model the minimal case (situation (1)) only. If the other two situations are needed they can be added as extra operations.

The Operation Schemas for the above three operations are given below:

OPERATION : Add_Order

DESCRIPTION : Creates an order in the system.

READS : supplied $qty : Nat$
supplied $desc : PROD\_DESC$

CHANGES : new $order : Order$

SENDS :

ASSUMES : there exists $pi : Product\_Info$ such that
$pi.prod\_desc = desc$ and
$count(pi.stocked\_product\_info) > 0$

RESULT : $order.status = pending$ and
$order.order\_item.qty = qty$ and
$(order.order\_item, pi) \in orders$

OPERATION : Cancel_Order

DESCRIPTION : Cancels an order in the system.

READS :

CHANGES : supplied $order : Order$

SENDS :

ASSUMES :

RESULT : if $order.status = invoiced$ then
$order.stock.stocked\_prod\_info.qty = order.stock.stocked\_prod\_info.qty +$
$order.order\_item.qty$ and
$order.status = cancelled$ else
$order.status = cancelled$

OPERATION : Add_Stocked_Product

DESCRIPTION : Adds to a product stock.

READS : supplied $desc : PROD\_DESC$
supplied $qty : Nat$
supplied $locn : LOCATION$

CHANGES : $s : Stock$ where $s.location = locn$

SENDS :

ASSUMES : there exists $s : Stock$; $spi : Stocked\_Product\_Info$
such that
$s.location = locn$ and
$spi \in s.stocked\_product\_info$ and
$spi.product\_info.prod\_desc = desc$

RESULT : spi.qty = spi.qty + qty

The Z formalizations of the preceding schemas are given in the Appendix, and they were analyzed as done in Case 1.

4 Conclusion

In this paper we give Fusion/UML models of the Invoicing problem, and analyze them by producing Z specifications from them and discharging general Z proof obligations. At the time this paper is written, this was the only analysis carried out. We plan to analyze the Z specifications by animating and checking domain-specific properties using the Z/EVES and ZANS tools.

The formalization of the OO models have not raised any major problems with our models to date. Most of the issues and questions were raised during the informal modeling stage. In this particular case study the formalization served to enhance our confidence in the models. With the Z animator we can also 'test' the operations using specific values (to do this here requires restating those parts that use the theta notation). This we plan to do before the workshop.

The models produced in this case study are relatively small, thus it should not be surprising that issues were uncovered during the informal modeling phase. For more complex problems the formalization activity is more likely to uncover significant inconsistencies, and missing details simply because the complexity of the problem does not allow one to unover these faults using only informal modeling techniques.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Menlo Park, CA, Second edition, 1994.

[2] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, Englewood Cliffs, NJ, Object-Oriented Series edition, 1994.

[3] Robert B. France, Jean-Michel Bruel, and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *To appear in the Journal of Object-Oriented Programming (JOOP)*, 1997.

[4] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Emanuel Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In *Proceedings of the 6th International AMAST Conference*, 1997.

[5] Robert B. France, Jean-Michel Bruel, Monika Saksena, Emanuel Grant, and Maria M. Larrondo-Petrie. Towards a rigorous object-oriented analysis and design method. In IEEE Computer Society Press, editor, *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods*, 1997.

[6] Robert B. France and Maria M. Larrondo-Petrie. A Two-Dimensional View of Integrated Formal and Informal Specification Techniques. In Jonathan P. Bowen and Michael G. Hinchey, editors, *Proceedings ZUM'95: The Z Formal Specification Notation – 9^th International Conference of Z Users,* Limerick, Ireland, volume 967 of *Lecture Notes in Computer Science*, pages 434–448. Springer-Verlag, New York, September 1995.

[7] Robert B. France, Jie. Wu, Maria M. Larrondo-Petrie, and Jean-Michel Bruel. A Tale of Two Case Studies: Using Integrated Methods to Support Rigorous Requirements Specification. In *Proceedings of the BCS FACS Methods Integration Workshop,* Leeds, UK, 25–26 March 1996. To appear in EWIC Series, Springer-Verlag, New York.

[8] The UML Group. Unified Modeling Language. Version 1.0, Rational Software Corporation, Santa Clara, CA-95051, USA, January 1997.

[9] Ian J. Hayes, editor. *Specification Case Studies.* Prentice Hall, Second edition, 1993.

[10] I. Jacobson. *Object oriented software engineering.* Addison-Wesley, 1992.

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

[12] Malcolm Shroff and Robert B. France. Towards a Formalization of UML Class Structures in Z. In *to appear in Proceedings of COMPSAC'97, Aug. 13-15, 1997*, August 1997.

[13] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.

[14] R. Wirfs-Brock and B. Wilkerson. *Designing object oriented software.* Prentice-Hall, 1990.

APPENDIX: The Z Formalizations of the Fusion/UML Models

Z: An Overview

Z [9, 13] is a general purpose formal specification notation that has a strong mathematical basis in predicate logic and set theory. Static aspects of applications are represented by sets and dynamic aspects by operations on sets. In this section we introduce only the Z notations necessary to understand the specifications given in this paper (see [13] for more details).

The primary structuring construct in Z is the *schema*. A schema has two parts: a declaration and a predicate part. The *declaration* part consists of variable declarations of the form $w : Type$, where w is a variable name and $Type$ is a type name. Intuitively, the preceding declaration means that the value of w is a member of the set named by $Type$ (types are sets in Z). The *predicate* part consists of a predicate that expresses the relationships among the declared variables.

Schema
Declaration
Predicate

Types in Z can be either *basic* or *composite*. Elements of basic types (or given sets) are used as the basic building blocks for more complex elements (elements with composite types). A basic type is declared as follows:

$$[BASIC\_TYPE\_NAME]$$

The internal structure of the elements of basic types are abstracted out (i.e., they are not of interest). There are three kinds of composite types in Z: set types, cartesian product types, and schema types. Using these types one can also define functions and relations (which are essentially sets of pairs, in which the first element is from the domain and the second element is from the range of the function/relation).

Schemas are used to model both the static and dynamic aspects of a system. A schema that captures the static aspect of a system will be referred to as a *state schema*, and a schema that captures the dynamic aspect will be referred to as an *operation schema*. In a state schema the components of a system's state are declared in the declaration section and the constraints on the state are given in the predicate part.

An operation schema defines the relationship between the state at the start and at the end of an operation's execution. The declaration part of an operation schema declares variables representing the before and after states, inputs, outputs, and any other variables needed to define the relationship. The predicate part of the

schema defines the relationship between the before and after states. The following conventions are used for variable names in operation schemas: unprimed variable (e.g., w) - value of variable before operation execution;
primed variable (e.g., w') - value of variable after operation execution;
variable ending in '?' - an input to the operation;
variable ending in '!' - an output from the operation;
ΔS denotes a possible change in state S.
ΞS denotes that the state S does not change.

Transforming Fusion Class Diagrams to Z specifications

Here we give only the transformation rules used in the case study.

In our formalization, a Class Diagram is a characterization of the valid states of the system, and the Operation Model specifies operations in terms of their observable effects on valid states. Here, a valid state of a system is a linked set of objects that exhibit the properties expressed in the Class Diagram (i.e., the structure is consistent with cardinality and other constraints, including those expressed as annotations in the Class Diagram). A valid state is called a *configuration* in this paper.

Formalizing Class Diagrams

Transforming Basic Classes. A basic class is a class that is not an aggregate or a subclass in a specialization hierarchy. For a basic class, $Class$, with attributes, $attr1, \ldots, attrn$, a basic type $CLASS$, denoting unique instances of the class, is created, and a *class schema* that relates elements of $CLASS$ to attribute values is created.

$$[CLASS, ATTR1, \ldots]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Class}} \\
\hline
classes : \mathbb{P} CLASS \\
cl\_attr1 : CLASS \pfun ATTR1 \\
\ldots \\
cl\_attrn : CLASS \pfun ATTRN \\
\hline
\mathrm{dom}\ cl\_attr1 = classes \\
\ldots \\
\hline
\end{array}
$$

In the above schema, $classes$ is the set of $Class$ instances that exist at the time, and $cl\_attr1, \ldots, cl\_attrn$ are functions relating instances to their attribute values. The constraint restricts the domain of these functions to existing instances.

Transforming Compositions. The components of a composition are formalized as described above (if they are basic classes; if they are compositions this rule is applied recursively). The whole part of a composition, *Comp*, with components $Part1, \ldots, Partn$ is formalized as follows:

$[COMP, PART1, \ldots, PARTN, \ldots]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Comp}} \\
\hline
Part1;\ \ldots;\ PartN \\
comps : \mathbb{P}\,COMP \\
comp\_attr1 : COMP \nrightarrow ATTR1 \\
\ldots \\
comp\_part1 : COMP \nrightarrow PART1 \\
\hline
\mathrm{dom}\ comp\_attr1 = comps \\
\mathrm{dom}\ comp\_part1 = comps \\
\ldots \\
\mathrm{ran}\ comp\_part1 = part1s \\
\ldots \\
\hline
\end{array}
$$

The set $part1s$ is the set of existing instances defined in the component schema $Part1$.

Transforming Associations. A one-to-many association between $ClassA$ and $ClassB$ (where every existing instance of $ClassA$ is associated with exactly one instance of $ClassB$, and one instance of $ClassB$ can be associated with zero or more instances of $ClassA$ is formalized as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rel}} \\
\hline
ClassA;\ ClassB \\
rel : CLASSA \nrightarrow CLASSB \\
\hline
\mathrm{dom}\ rel = classas \\
\mathrm{ran}\ rel \subseteq classbs \\
\hline
\end{array}
$$

In the above, $classas, classbs$ are sets of existing instances of $ClassA$ and $ClassB$ respectively, defined respectively in the schemas $ClassA$ and $ClassB$.

See [4, 12, 5] for rules that handle more general relationships.

.1 Formalizing Operation Schemas

Fusion's operation schemas are expressed mostly in natural language. It is not likely that a tool that automatically produces a formal specification using only the information in the operation schema can be built currently. A tool that produces a partial specification is a more realistic possibilit y. We use the following guidelines to obtain a partial Z operation schema from a Fusion operation schema.

1. The operation name is used as the name of the Z operation schema.

2. If the **Changes** section is non-empty, then the declaration $\Delta SystemConfig$, where $SystemConfig$ is the configuration schema for the Object Model, is made in the Z operation schema. This declaration indicates that a change in the configuration is possible. More precisely, it introduces unprimed variables representing configuration elements before the operation's execution, and primed variables representin g configuration elements after execution, and a predicate expressing invariant properties on before and after configurations. If the **Change s** section is empty, then the declaration $\Xi SystemConfig$, that introduces a predicate equating the before and after configurations, is made.

3. In the **Reads** section, **supplied** elements are inputs to the operation, and the other elements are elements in the configuration. For each supplied element an input variable (with a name ending in '?') is declared in the Z schema. For each of the other elements in this section a primed and unprimed local variable (representing the before and after states of the element), is declared.

4. In the **Changes** section, **new** elements are instances created by the operation. A local variable (preferably primed) is declared for each of these elements. Other elements in this section are represented by unprimed and primed variables in the Z operation schema.

5. The data sent in the **Sends** section are declared as output variables in the Z schema.

6. Information in the **Assumes** and **Result** sections are used to construct the predicate part of the Z schema. Formalization of these sections is done manually.

Case 1 Z Specifications

$$[PRODTYPE, ORDER\_ITEM, ORDER]$$
$$STATUS ::= pending \mid invoiced$$

Order_Item

$$orderitems : \mathbb{P}\, ORDER\_ITEM$$
$$orderitems\_qty : ORDER\_ITEM \rightarrow\!\!\!\!\!\!+ \mathbb{N}_1$$

$$\mathrm{dom}\ orderitems\_qty = orderitems$$

$Order$

$Order\_Item$
$orders : \mathbb{P} ORDER$
$order\_status : ORDER \nrightarrow STATUS$
$order\_OI : ORDER \rightarrowtail ORDER\_ITEM$

$\mathrm{dom}\ order\_status = \mathrm{dom}\ order\_OI = orders$
$\mathrm{ran}\ order\_OI = orderitems$

$[STOCK\_PRODUCT\_INFO, LOCATION]$
$[PRODUCT, STOCK, DESCR]$

$Product\_Info$

$productinfos : \mathbb{P} PRODUCT\_INFO$
$productinfo\_descr : PRODUCT\_INFO \nrightarrow DESCR$

$\mathrm{dom}\ productinfo\_descr = productinfos$

$Stock\_Product\_Info$

$stproductinfos : \mathbb{P} STOCK\_PRODUCT\_INFO$
$stproductinfo\_qty : STOCK\_PRODUCT\_INFO \nrightarrow \mathbb{N}_1$

$\mathrm{dom}\ stproductinfo\_qty = stproductinfos$

$Stocked$

$Stock\_Product\_Info$
$Product\_Info$
$stocked : STOCK\_PRODUCT\_INFO \nrightarrow PRODUCT\_INFO$

$\mathrm{dom}\ stocked = stproductinfos$
$\mathrm{ran}\ stocked \subseteq productinfos$

$Orders$

$Product\_Info$
$Order\_Item$
$orders\_rel : ORDER\_ITEM \nrightarrow PRODUCT\_INFO$

$\mathrm{dom}\ orders\_rel = orderitems$
$\mathrm{ran}\ orders\_rel \subseteq productinfos$

$Stock$

$$
\begin{array}{l}
Stock\_Product\_Info \\
stocks : \mathbb{P}\, STOCK \\
stock\_locn : STOCK \rightarrowtail\!\!\!\!\!\!+\; LOCATION \\
stock\_SPI : STOCK \leftrightarrow STOCK\_PRODUCT\_INFO \\
\hline
\mathrm{dom}\ stock\_locn = \mathrm{dom}\ stock\_SPI = stocks \\
\mathrm{ran}\ stock\_SPI = stproductinfos \\
\forall s1, s2 : stocks \bullet stock\_SPI(\!|\ \{s1\}\ |\!) \cap stock\_SPI(\!|\ \{s2\}\ |\!) = \emptyset \\
\forall spi : \mathrm{ran}\ stock\_SPI \bullet \#(stock\_SPI^{\sim}(\!|\ \{spi\}\ |\!)) = 1
\end{array}
$$

$Handled\_By$

$$
\begin{array}{l}
Stock \\
Order \\
handled\_by : ORDER \rightarrowtail\!\!\!\!\!\!+\; STOCK \\
\hline
\mathrm{dom}\ handled\_by \subseteq orders \\
\mathrm{ran}\ handled\_by \subseteq stocks \\
\mathrm{dom}\ handled\_by = \{o : orders \mid order\_status(o) = invoiced\}
\end{array}
$$

$Order\_Sys$

$$
\begin{array}{l}
Stocked \\
Orders \\
Handled\_By
\end{array}
$$

$Process\_Order$

$\Delta Order\_Sys$
$order? : ORDER$
$spi : STOCK\_PRODUCT\_INFO$

$order? \in orders$
$order\_status(order?) = pending$
$spi \in (stocked^{\sim} (\| \{orders\_rel(order\_OI(order?))\} \|))$
$stproductinfo\_qty(spi) \geq orderitems\_qty(order\_OI(order?))$
$stproductinfo\_qty' = stproductinfo\_qty \oplus$
$\{(spi, stproductinfo\_qty(spi) - orderitems\_qty(order\_OI(order?)))\}$
$order\_status' = order\_status \oplus \{(order?, invoiced)\}$
$stproductinfos' = stproductinfos$
$\exists s : stocks \mid s \in stock\_SPI^{\sim} (\| \{spi\} \|) \bullet$
$handled\_by' = handled\_by \cup \{(order?, s)\}$
$orders' = orders$
$order\_OI' = order\_OI$
$stocked' = stocked$
$\theta Orders' = \theta Orders$
$stock\_locn' = stock\_locn$
$stock\_SPI' = stock\_SPI$