Invoicing Orders

Achim D. Brucker

October 14, 2008

We present a reference example for data refinement using UML/OCL. The refinement related proof obligations are proven using HOL-OCL [1, 2].

1 The Invoice System

We present a well-known case study for comparing specification formalisms, e. g., Frappier and Habrias [4] give an overview of several formalization of this case study using different formalisms.

1.1 Informal Description

Frappier and Habrias [4] describe the InvoicingOrders system informally as follows (see also `http://www.dmi.usherb.ca/~spec/texte-cas-invoicing.htm`):

1. The subject is to InvoicingOrders orders.
2. To InvoicingOrders is to change the state of an order (to change it from the state "pending" to "invoiced").
3. On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders.
4. The same reference can be ordered on several different orders.
5. The state of the order will be changed into "invoiced" if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product.
6. You have to consider the following two cases:
   a) Case 1:
      All the ordered references are references in the stock. The stock or the set of orders may vary:
      - due to the entry of new orders or canceled orders;

- due to having a new entry of quantities of products in stock at the warehouse.

But, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

b) Case 2:
You do have to take into account the entries of

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

1.2 Formal Specification

In this section, we present a formalization of the Invoice case-study using UML/OCL. Dupuy et al. [3] already present a UML specification for the InvoicingOrders system. But this specifications lacks any usage of OCL. Moreover, the use of UML is quite informal, e. g., their specification is untyped. Our work is inspired by the formalization of Dupuy et al. [3], in fact, we restrict ourselves to making their specification more rigid. For example, we provide full type annotation and complete the diagrammatic part of UML with a detailed OCL specification.

1.2.1 Case 1: Products and Orders

Figure 2 shows the data model of our case study is quite simple. For realizing item 6a we only need the classes `Product` and `Order`. For realizing the item 6b, we also model a class `Warehouse`. **??** presents the UML data model of the Invoice system. Table 1 presents the OCL specification for constraining the state part of the system, i. e., constraining the datatypes. For example, UML/OCL does not provide a datatype for natural numbers, therefore we use the datatype `Integer` and constrain the corresponding attributes to positive values. **??** describes the behavior of the Invoice case-study, i. e., the precondition and postconditions of the operations. Overall, this completes the OCL specification.

Finally, **??** presents the complete theory file containing the proofs explained in **??**.

theory *InvoicingOrders*
imports
  OCL
begin

1.2.2 Importing The Model

Importing the Model is easy, just use

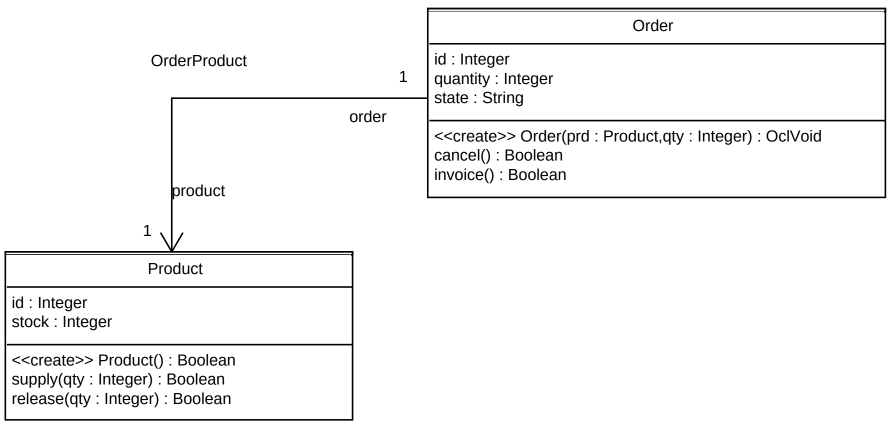
Order
id : Integer
quantity : Integer
state : String
<<create>> Order(prd : Product,qty : Integer) : OclVoid
cancel() : Boolean
invoice() : Boolean
OrderProduct
1
order
product
1
Product
id : Integer
stock : Integer
<<create>> Product() : Boolean
supply(qty : Integer) : Boolean
release(qty : Integer) : Boolean

Figure 1: Case 1: The Invoice Case-study models a simple system for invoicing orders; thus we need to model at least products, orders, and a warehouse managing the orders and products.

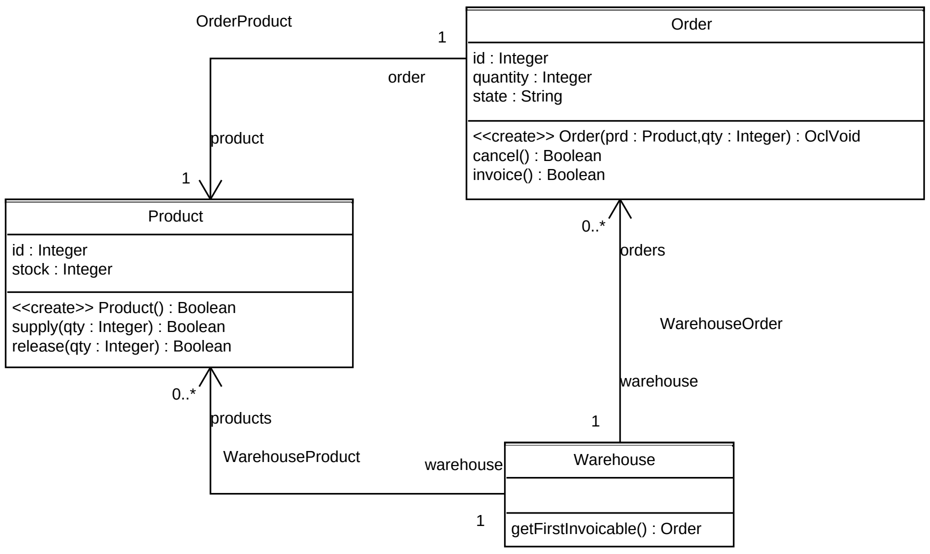

Figure 2: Case 2: The Invoice Case-study models a simple system for invoicing orders; thus we need to model at least products, orders, and a warehouse managing the orders and products.

```
------------------------------------------
-- Case 1
------------------------------------------
package InvoicingOrders
  --
  -- 1) Constraining the Data Model
  -- ==============================

  -- The stock of a Product is always a natural number, i.e., it is a
  -- positive Integer. This also ensures the definedness of the stock.
  context Product
    inv isNat: self.stock >= 0

  -- The Product id is unique.
   context Product
     inv idUnique: Product::allInstances()
                      ->forAll(p1:Product, p2:Product | p1.id <> p2.id)

  -- The quantity of an Order is always a natural number, i.e., it is
  -- a positive Integer. This also ensures the definedness of the
  -- quantity.
  context Order
    inv isNat: self.quantity >= 0

  -- The state of an Order should either be ‘pending’  or ‘invoiced’.
  -- As a direct support for enumeration is not well developed in most
  -- CASE tools, we use a String and constrain it to the two
  -- alternatives using an invariant.
  context Order
    inv stateRange:    (self.state = 'pending')
                    or (self.state = 'invoiced')

  -- The Order id is unique.
   context Order
     inv idUnique: Order::allInstances()
                      ->forAll(o1:Order, o2:Order | o1.id <> o2.id)



  --
  -- 2) Constraining the Dynamic Description
  -- =======================================

  -- Initialize the state of an Order
  context Order::state : String
    init: 'pending'

  -- Create a new Order
  context Order::Order(prd:Product, qty:Integer):OclVoid
    pre: qty > 0
    pre: self.warehouse.products->exists(x:Product | x = prd)
    pre: not prd.oclIsUndefined()
    post: self.oclIsNew() and  self.quantity = qty
       --   and self.orderedProduct = prd

  -- The state of the order will be changed into "invoiced" if the ordered quantity
  -- is either less or equal to the quantity which is in stock according to the
  -- reference of the ordered product.
   context Order::invoice() : Boolean
     pre: self.state = 'pending'
--          and self.quantity <= self.orderedProduct.stock
     post: self.state = 'invoiced' and self.quantity = self.quantity@pre
--           and self.orderedProduct = self.orderedProduct@pre
--           and self.orderedProduct.stock = self.orderedProduct@pre.stock - self.qu

  -- Cancel order as an opposite operation to invoice order
  context Order::cancel() : Boolean
    pre:  self.state = 'invoiced'
    post: self.state = 'pending'
```

import_model *InvoicingOrders.zargo InvoicingOrders.ocl*

which loads the data model and the OCL specification of our case study. The import of our model takes about 10 seconds and already generates . . . of conservative definitions and proven theorems.

1.2.3 Proving simple properties

First we show, that a very simple property of OCL: an positive Integer is allways defined:

```
lemma foo: τ ⊨ Product.inv.isNat self ⟹ τ ⊨ ∂ Product.stock self
  apply(frule isDefined_if_valid')
  apply(simp add:  Product.inv.isNat_def)
  done

lemma foo [simp]: τ ⊨ ∂ (0::('a Integer))
apply(auto)
done


lemma [simp]: τ ⊨ 0≤ (0::('a Integer))
apply(simp add:OCL_Integer.OclLe_def OclLocalValid_def)
apply(simp add: lift2_def strictify_def)
apply(auto)
prefer 2
apply(simp add: OclTrue_def lift0_def)
apply(unfold Zero_ocl_int_def)
apply(auto simp: OclTrue_def lift0_def)
done


lemmas Product_weakinv =
weakinvariant1.init_stock_def
Product.weakinvariant1.isNat_def
Product.weakinvariant1.init_id_def
weakinvariant1_def

lemmas Product_post =
  Product_Boolean.post1_def
  Product_Boolean.post1.post_0_def
  Product_Boolean.post1.post_0_def


lemmas supply_post =
supply_Integer_Boolean.post1_def
supply_Integer_Boolean.post1.post_0_def
  supply_Integer_Boolean.post1.post_0_def

lemmas release_pre =
release_Integer_Boolean.pre1_def
```

release_Integer_Boolean.pre1.pre_0_def
release_Integer_Boolean.pre1.pre_0_def

lemma *implies_false_asumption*: $\tau \models \neg\ \varphi \Longrightarrow \tau \models \varphi \ \longrightarrow\ \psi$
 by(*ocl_simp*, *simp*)

lemma *implies_prem*: $\tau \models \varphi \Longrightarrow \tau \models \psi \longrightarrow\ \varphi$
  by(*ocl_simp*, *simp*)

lemma *implis_prem*: $\tau \models \varphi \Longrightarrow \tau \models \psi \longrightarrow\ \varphi$
  apply(*unfold OclImplies_def*)
  apply(*rule or_TRUE_I_core*)
  apply(*rule disjI2*)
  by(*assumption*)

end

References

[1] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. Ph.d. thesis, ETH Zurich, Mar. 2007. URL `http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007`. ETH Dissertation No. 17097.

[2] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006`.

[3] S. Dupuy, A. Front-Conte, and C. Saint-Marcel. Using UML with a behaviour-driven method. In Frappier and Habrias [4], chapter 6. ISBN 1-85233-353-7.

[4] M. Frappier and H. Habrias, editors. *Software Specification Methods: An Overview Using a Case Study*. Formal Approaches to Computing and Information Technology. Springer-Verlag, London, 2000. ISBN 1-85233-353-7.