International Journal of Software Engineering and Knowledge Engineering

Research Notes: Formal Approach to Assertion Based Code Generation

PENGYI LI, JING SUN
Department of Electrical and Computer Engineering
Department of Computer Science
The University of Auckland, Private Bag 92019
Auckland 1142, New Zealand
pli552@aucklanduni.ac.nz, jing.sun@auckland.ac.nz

HAI WANG
School of Engineering and Applied Science
Aston University, Birmingham B47ET
United Kingdom
h.wang10@aston.ac.uk

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

With the growing in size and complexity of modern computer systems, the need for improving the quality at all stages of software development has become a critical issue. The current software production has been largely dependent on manual code development. Despite the slow development process, the errors introduced by the programmers contribute to a substantial portion of defects in the final software product. This paper investigates the synergy of generating code and assertion constraints from formal design models and use them to verify the implementation. We translate Z formal models into their OCL counter-parts and Java assertions. With the help of existing tools, we demonstrate various checking at different levels to enhance correctness.

Keywords: Formal Specification; Verification and Validation; Simulation; Java Assertions.

1. INTRODUCTION

With the growing in size and complexity of modern computer systems, the need for improving the quality at all stages of software development has become a critical issue [1, 2]. The focus lies in how to effectively develop high quality software. Looking at a topical software development life cycle, there are two main types of potential errors to a software product, i.e., design errors and programming errors [3, 4]. The former refers to the defects introduced at the design stage where the proposed product failed to capture the correct functionalities of the system under construction. The latter refers to the defects introduced at the implementation stage where the programmers failed follow the correct design in producing the program (code).

The key issue is how to effectively and efficiently develop code from verified design models [7]. Automation reflects an essential motive and aspect of future software engineering practice. The era of computing has revolutionised the mathematical computation and data processing from human-power into machine-power. However, the current software production is still largely dependent on manual code development, i.e., humans write programs manually from the design solutions. Despite the slow development process, the errors introduced by the programmers contribute to a substantial portion of defects in the final software product. Ideally, the executable program (code) should be automatically generated from the design model [13, 14], which has been known in other engineering disciplines, e.g., mechanical and electrical, as the production automation. This will not only dramatically increase the productivity, but also overcome the human errors at the implementation phase, hence improve the overall quality of modern software development.

Currently there are few approaches being developed to implement the automatic code generation[17, 4]. Most related works are based on diagrammatic notations and generating partial code framework, such as UML to Java [2, 3]. Despite the informal description of the design notations, the generated programs are skeleton code only, which are not executable. Refinement [16] is another related field that aims at deriving less abstract models from formal specifications. However, there has been no work in refining a formal model directly into the executable program. In this paper, we explore the synergy of generating code and assertion constraints from formal design models and use them to verify the implementation. We translate Z formal models into their OCL [8] counter-parts and Java assertions. With the help of existing tools, we demonstrate various checking at different levels to enhance correctness.

The rest of the paper is structured as follows. Section 2 introduces the model transformation from Z to UML/OCL. Section 3 presents the translation from UML/OCL to Java code. Section 4 describes the benefit of translations in terms of verification, simulation, code generation and validation. Section 5 discusses the related work in the field. Section 6 concludes the paper and outlooks the future work.

2. Background

2.1. *Z Formal Language*

Z [1] is a state-based formal language based on ZF set theory and first-order predicate logic. Z/EVES [9] is an interactive system for composing, checking, and analysing Z specifications. In particular, it supports general theorem proving of Z specifications. Z/EVES organises Z specifications in the form of sections to improve structure and reuse. The built-in section toolkit defines basic constants and operators. Specifications are built hierarchically by including existing sections as their parents.

In Z/EVES, theorems can be developed to describe properties about a model.

They appear in the form of axioms, rewrite rules or assumption rules. An axiom is treated as a fact, which is not invoked during reduction. When Z/EVES encounters a rewrite rule, it will rewrite its left-hand side to its right-hand side. An assumption rule is assumed to be true when Z/EVES performs simplification. Theorems can be marked as disabled so that they are not automatically invoked during reduction or rewriting.

2.2. *UML and OCL*

The Unified Modeling Language (UML) [11] is a diagrammatic notation for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. The Object Constraint Language (OCL) is as an annotation language of UML to overcome the limitations of preciseness in description, which is proposed by the Object Management Group (OMG). It can be used to express preconditions, post-conditions, invariants, guard conditions, and results of method calls in an UML design model.

The EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. The core EMF framework includes a meta-model (Ecore) for modeling. Papyrus provides a language to edit any kind of EMF model and support UML and related modeling language like SysML and MARTE. It uses diagram editors to create UML model and supports OCL language on the model.

UML-based Specification Environment (USE) is a simulation environment that supports a subset of the OCL definitions. An USE specification contains a textual description of a design model. Expressions written in the OCL are used to specify additional integrity constraints on the model. A model can be simulated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during the validation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about the system.

2.3. *Java Assertion and JUnit Test*

An assertion is a statement in the Java programming language that enables the testing of assumptions in the program. Java Assertions are used to verify the correctness of invariants in the code. Each assertion contains a Boolean expression. If the Boolean expression is evaluated to be false, the system will throw an exception. In Java, assertions use keyword “assert” to express the boolean expression.

JUnit [6] aims at unit testing in Java programs, which refers to checking the smaller units of your application, such as classes and methods. Unit tests are typically automated, meaning that once they are implemented, it can be executed repeatedly. Writing tests before developing the code could force the developers to

consider the functions and logic in the implementation more deeply. JUnit has a set of common plug-ins for the Eclipse development platforms.

3. Assertion Based Model Transformation

3.1. *The Overall Approach*

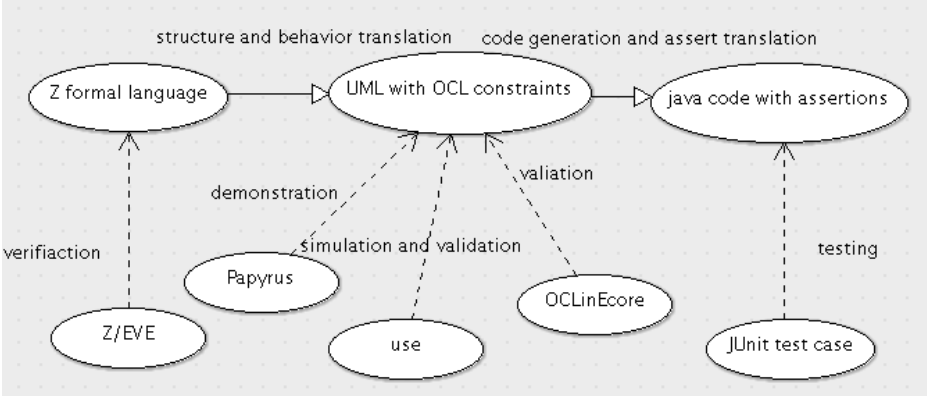

Fig. 1. Overview of our approach

Figure 1 illustrates a summary of our approach. Firstly, the design models are documented using the Z formal specification language [12]. Z is a first order logic and set theory based notation, which has a number of tools in supporting the verification and validation of the design, such as Z/EVES, CZT, ProB, etc. Due to the high level abstraction of the Z language and its lacking of object oriented design, an intermediate process is necessary for code generation. On the other hand, UML, a semi-formal diagrammatic notation, is a widely used design language by the industry and could well serve the purpose. In our project, Java is selected as the target implementation, and there are a number of tools which support the generations of skeleton code from UML into Java.

Secondly, to overcome the informal descriptions of UML, the formal specification from the Z model can be translated into OCL annotations. According to the steps shown, the approach translates Z formal models to UML with OCL constraints and then generates code with embedded assertions from UML to Java. Thirdly, to further guarantee correctness, OCL constraints are translated to Java assertions to validate the implementation. Furthermore, OCL constraints can be used in JUnit [6] test cases, which is easier for the developers to debug their code.

Finally, in addition to code generation, there are tools that could be used to support the rigorous verification at different stages of the development. For example, Z/EVE provides theorem to verify the desired properties at the specification level.

At the UML/OCL phase, Papyrus can interpret UML models with OCL constraints and generate Java code. OCLinEcore and USE both provide simulation facilities for creating object models and validating OCL constraints. At the code level, Java assertions can be used for validating the implementation. For example, assertions can be used together with JUnit for testing.

3.2. *The Online Shopping System*

A online shopping system is used as a case study and running example for presenting the translation rules and the verification. The requirements of the system aim at providing customers with abilities to perform online purchases on products on sale. The system consists of a set of products and shopping carts for users to add or remove items. A sale items has its own product description, such as the barcode, unit price, quantity available, etc. Users are able to add/delete products to /from their shopping carts. User can decide whether he/she want to continue shopping or not. During the process of products insertion, system needs to update the corresponding quantity of the product. After completing a checkout, the system should generate a bill, list the bought items and the amount to be paid for the purchases.

Consider the requirements, the Z specification of the main state schema is shown in Figure 2, which provides a static view of the system. In the declaration part, products denote a power set of SaleItem and the cart is an instance of the state schema shopping Cart. In the predicate part, it states an invariant to ensure the product code is unique. The state schema Cart has a function items mapping between a SaleItem and a natural number represents the amount bought. To specify how the

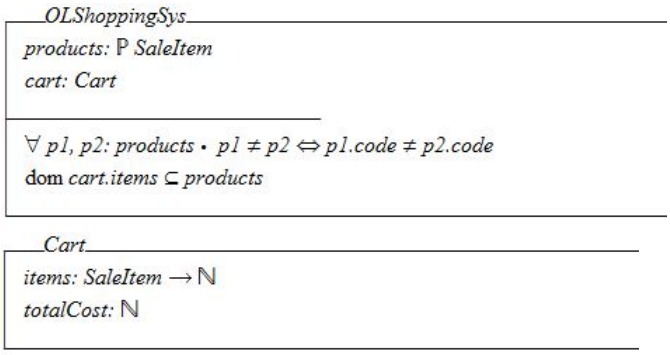

Fig. 2. The Shopping System and the Cart state schemas

system states undergo changes, we define the operation schemas. We express predicates as the relationship between the instances of the states before and after the operation.

The AddItem operation schema is defined in Figure 3. In the declaration, an

operation that changes the state schema is indicated using the delta convention. We have one product of the SaleItem type as inputs. If the selected product is on sale, the add operation can proceed. It is also necessary that in a real situation the quantity that the customer wants to add to shopping cart should be checked. If the product has already in the shopping cart, the input product's quantity in cart will increase by one. Otherwise, the shopping cart will be inserted by items with key-value format as "selected product $\mapsto$ 1" in a pair. Finally, the operation schema updates the input product's own quantity and changes the product's stock status. That is, if the AddItem operation is successful, the quantity of the selected product should be decreased by one. And if the product's quantity becomes zero after the update, the stock status of the product should be modified to 'OutofStock'.

AddItem

$\Delta OLShoppingSys$
$p?: SaleItem$

$p? \in products$
$p?.stock = InStock$
$p?.quantity > 0$
if $p? \notin \text{dom } cart.items$
then $cart'.items = cart.items \cup \{(p? \mapsto 1)\}$
else $cart'.items = cart.items \oplus \{(p? \mapsto cart.items\ p? + 1)\}$
$cart'.totalCost = cart.totalCost + p?.price$
$\exists p1: products'$
• $products \setminus \{p1\} = products \setminus \{p?\}$
$\wedge\ p1.code = p?.code$
$\wedge\ p1.description = p?.description$
$\wedge\ p1.price = p?.price$
$\wedge$ (**if** $p?.quantity > 1$
then $(p1.quantity = p?.quantity - 1 \wedge p1.stock = InStock)$
else $(p1.quantity = 0 \wedge p1.stock = OutofStock)$)

Fig. 3. AddItem operation schema

The DeleteItem operation schema is defined in Figure 4. In the declaration, p? denotes an input item for deleting. If the selected product is in the shopping cart, it can be deleted. The schema checks whether the quantity value of the selected product in shopping cart is one or not. If the quantity is one, the product should be removed from the cart. Otherwise, the system will decrease the quantity by one. If the DeleteItem operation is successful, the quantity of the product should be increased by one.

The DeleteAll operation schema is defined in Figure 5. After the operation, the

DeleteItem

$\Delta OLShoppingSys$

$p?: SaleItem$

$p? \in \text{dom } cart.items$

if $cart.items\ p? = 1$

then $cart'.items = \{p?\} \lhd cart.items$

else $cart'.items = cart.items \oplus \{(p? \mapsto cart.items\ p? - 1)\}$

$cart'.totalCost = cart.totalCost - p?.price$

$\exists p1: products'$

- $products' \setminus \{p1\} = products \setminus \{p?\}$
  - $\wedge\ p1.code = p?.code$
  - $\wedge\ p1.description = p?.description$
  - $\wedge\ p1.price = p?.price$
  - $\wedge\ p1.quantity = p?.quantity + 1$
  - $\wedge\ p1.stock = InStock$

Fig. 4. DeleteItem operation schema

shopping cart should be cleared and the system will update each product's quantity value accordingly. Other operations such as the CheckOut, RemoveItem, etc., can be defined similarly. Due to space limit, we will not present all of them here.

DeleteAll

$\Delta OLShoppingSys$

---

if $\text{dom } cart.items = \emptyset$

then $products' = products \wedge cart' = cart$

else $cart'.items = \emptyset$

- $\wedge\ cart'.totalCost = 0$
- $\wedge\ (\forall p: \text{dom } cart.items$
  - $(\exists p1: products'$
    - $(p1.code = p.code$
      - $\wedge\ p1.description = p.description$
      - $\wedge\ p1.price = p.price$
      - $\wedge\ p1.stock = InStock$
      - $\wedge\ p1.quantity = p.quantity + cart.items\ p)))$

Fig. 5. DeleteAll operation schema

3.3. *Translating Z into UML with OCL Constraints*

This section describes the translation from Z formal models into UML/OCL specifications. The mapping consists of two parts, i.e., the structure translation and the behaviour translation.

3.3.1. *Structure Translation*

Z Schema A Z formal specification contains state schemas and operation schemas. The state schemas can be translated to classes in UML/OCL. The operation schemas can be translated into methods that belong to the specific classes. Operation schemas that change the value of state variables are indicated using the "Δ" convention; operation schemas that do not change state variables are indicated with the "Ξ". The state schema name that after the "Δ"or "Ξ" symbol can be used to identify the class that a method is belonging to.

In the example system, there is a state schema named "OLShoppingSys". The state schema should be translated to a class named "OLShoppingSys". There are six operation schemas, four of them have "ΔOLShoppingSys". We translate these operation schemas to methods of the"OLShoppingSys" class. The main method is a schema calculus, which connects the operations GetChoice and AddItem or DeleteItem or DeleteAll or CheckOut.

Z Data Types A Z schema consists of two parts, i.e., declarations and predicates. The declaration part of structure translation can be summarised into data type mappings to their counterparts in UML/OCL.

Given, basic and free types Z uses [] to express a new given type. For example, [MSG] in Z can be translated to a class MSG with no attributes. Z only supports two primitive data types, i.e., natural numbers and integers, which can be mapped to integers in OCL with additional constraint, such as 'n>0'. Free types in Z can be mapped into enumeration types with values in OCL. For example, 'Z: StockStatus::=InStock | OutofStock' would be mapped into 'enum StockStatus {InStock,OutofStock}' in UML/OCL.

Sets are expressed as {x:X | P(x)} in Z, which can be directly mapped into the Set type in OCL. In Z formal language, S$\subseteq$T means that S is a subset of T, which is equivalent to T$\rightarrow$includesAll(S); For example, 'products:P SaleItem' in Z can be translated into 'products: Set(SaleItem)' in UML/OCL. Cardinality #X in Z is a natural number denoting the size of a set. In UML/OCL, there is a predefined operation $\rightarrow$size() that has same meaning with "#".

Cartesian Product, Relations and Functions : A$\times$B (A cross B) in Z can be translated in to an association class between two participating classes in UML/OCL with multiplicity constraint as many to many. A relation R from A to B in Z repre-

sents a subset of the cartesian product with special restrictions. This can be translated into invariant definitions on the associate class to restrict the mapping. A function R: A→B in Z differs from a relation in the number of valid mappings through the relationship, i.e., for each a ∈ A there is at most one b ∈ B. In UML/OCL functions be denoted as an association class that connects class A with class B, and this association has multiplicity of many to one.

Sequences in Z represents a sequence of elements from a set A, denoted as 's:seq A'. UML/OCL also has a concept 'Seq' to express a sequence of instances. Therefore, a direct mapping from 'seq' in Z to the 'Seq' type in UML/OCL.

3.3.2. *Behavioural Translation*

The syntax of standard OCL language uses contextual keyword to introduce the scope of an OCL expression, followed by the name of the type. The keyword *inv*, *pre* and *post* denotes the invariants, pre-conditions and post-conditions of the constraints. After the structure translation, the design specification needs to be transformed by the behavior translation, which consists of the following two parts:

- Translations from state schema's predicates into OCL invariants that is denoted by the prefix 'inv'.
- Translations from operation schema's predicates to OCL pre/post-conditions, which is denoted by the prefix 'pre' and 'post'.

Translation of predicates in the state schema is rather straightforward mappings into class invariants. However, the mappings of predicates in the operation schema requires the identifications of pre- and post- conditions. When system gets a predicate from the operation schema, the first thing is to confirm whether this predicate is a pre-condition or post-condition. In a post-condition, the expression can refer to two sets of values for a state variable, i.e., the value before and after the operation. If the predicate contains variable followed by the Z prime symbol 'x', it will be a post-condition. In UML/OCL, pre-state of a variable is expressed with suffix @pre, and the post-state of a variable is expressed as itself. The "@pre" postfix is allowed only in OCL expression that is a part of the post-condition.

Operation schemas may have input and output variables, input parameters are denoted with the postfix symbol of '?'; and output parameters are denoted with the postfix symbol of '!' in Z. These input and output variables can be translated to input parameters and return type of the operation schema (method) in UML/OCL. Finally, in order to translate the entire predicate into its OCL counterpart, we need to provide the mappings for various Z operators.

Logic and relational operators suc as 'Not (¬),And (⋀),Or (⋁),Implies (⇒) and Equivalence (⇔)' can be translated into 'not, and, or, implies and two direction implies in UML/OCL. When we translate A ⇔ B, we should make sure A implies

B and B implies A. Relational operators such as '$\leq, \geq, =, \neq$, and if then else' can be directly mapped to '<=,>=, <>, and if-then-else-endif' in UML/OCL.

Set operators could be mapped into operations in the UML/OCL set type, e.g., 's ∈ S', can be represented as S→includes(s) in UML/OCL. And 's∉S' means S→excludes(s). Set Union (∪), Set Intersection (∩), Set Difference (\) maps into →union(), →intersection(), and →reject() separately in UML/OCL. For example, 'products\{p}=products'\{p1}' could be translated into 'self.products@pre→reject(code=p.code) = self.products→reject(code=p1.code)'.

Quantifiers Universal Quantifier (∀) can be translated into the →forAll(...) operation in OCL; and the Existential Quantifier (∃) can be translated into the →exist(...) operation in OCL. For example, '∀p1, p2: products •p1 ≠p2 ⇔ p1.code≠ p2.code' can be translated into UML/OCL as "self.products→forAll(p1,p2|(p1<>p2 implies p1.code<>p2.code) and (p1.code<>p2.code implies p1<>p2))". Because the predicate is in the state schema OLShoppingSys, it should be translated into invariants in OCL with context of the class OLShoppingSys. In UML/OCL, self.products and products both present the products in Z, self refers to the context of OLShoppingSys. "∀p1, p2:products equals self.products→forAll(p1,p2|...)" in UML/OCL."⇔" maps into two implies, i.e., "p1<>p2 implies p1.code<>p2.code and p1.code<>p2.code implies p1<>p2". Another more complete example in Z is shown in Figure 6.

$$
\begin{aligned}
&\exists p1 : products' \\
&\quad \bullet\ products \setminus \{p1\} = products \setminus \{p?\} \\
&\quad \wedge p1.code = p?.code \\
&\quad \wedge p1.description = p?.description \\
&\quad \wedge p1.price = p?.price \\
&\quad \wedge (\textbf{if}\ p?.quantity > 1 \\
&\qquad \textbf{then}\ (p1.quantity = p?.quantity - 1 \wedge p1.stock = InStock) \\
&\qquad \textbf{else}\ (p1.quantity = 0 \wedge p1.stock = OutofStock))
\end{aligned}
$$

Fig. 6. Example for quantifies

Firstly, '∃p1:products' maps to 'products→exist(p1|...)', all the other lines after the first line of predicate is a set of boolean expression for p1, which are translated into 'exist()'. The predicate 'products\{p1}=products\{p?}' is translated into'products→exist(p1|self.products@pre→reject(x|x.code= p.code)=self.products→reject(x|x.code=p1.code))', as shown in Figure 7.

There are two things to be kept in mind, i.e., UML/OCL has a 'endif' after one 'if-else-then' expression finished and enumerations literals can not be invoked directly. It will be write as "Enumerations::literal".

```
post AddItempost3:self.products->exists(p1| p1.code=p.code
    and self.products@pre->reject(x|x.code=p.code)
        =self.products->reject(x|x.code=p1.code)
    and p1.description=p.description
    and p1.price=p.price
    and (if  p.quantity >1
         then
         (p1.quantity=p.quantity-1 and p1.stock=StockStatus::InStock)
         else
         (p1.quantity=0 and p1.stock=StockStatus::OutofStock)
         endif))
```

Fig. 7. Translation result for quantifies example

Relation operators Domain and range: dom R is a limitation for domain(A) so in UML/OCL, just limits to A class; ranR is a limitation for range(B) so in UML/OCL just limits B class. Domain and range restriction: R is a relation for A↔B and S ⊆ A and T ⊆ B; then S◁R is the set {(a,b):R|a∈S} R▷T is the set {(a,b):R|b∈T}. In UML/OCL, domain and range restrictions can be translated into invariant constraints imposed onto the association class. Relational composition: R : A ↔ B and S : B ↔ C R ; S ={(a,c):A×C|∃b:B•a R b ∧ b S c}, according to the translation, A, B and C should be three classes and R ; S just means creating a new association class from A to C.

Function overriding f,g:A→B;then f ⊕g is the function (dom g ⩤f)⋃g. As the semantics show that restrictions on the association classes f and g have to be combined. That is, the class A's value must be dom g→union(dom f), and class B's value must be ran f minus the part that dom g points to then →union(ran g).

Sequence operators such as concatenation "˜": <a, b> ˜<b, a, c> = <a, b, b, a, c> in Z means <a,b>→append(<b,a,c>) in UML/OCL. In Z, sequence also has two important functions, i.e., 'Head' and 'Tail'. Head is to obtain the first element of a sequence return as an element. Tail is to have the rest of sequence's elements except the head. In UML/OCL, for a sequence Q, 'Q→first()' equals to 'Head' and 'Q→excluding(Q→last())' equivalents to 'Tail'.

3.3.3. *Translation Results*

Based on the above mapping rules, we can translate a Z formal specification into its corresponding UML/OCL model. The result of structure translation of the online shopping system is shown in Figure 8.

For a formal Z model, we locate all state schemas, enumerations and new data types then translate them to classes. The attributes or enumeration literals should be placed in relevant classes. In the online shopping system, there are three state schemas, one enumeration and two new data types. As mentioned previously, "[String,Real]" are new data types in Z but they are primitive types in UML/OCL. For the enumeration 'StockStatus', it translates into enumerations with "InStock"

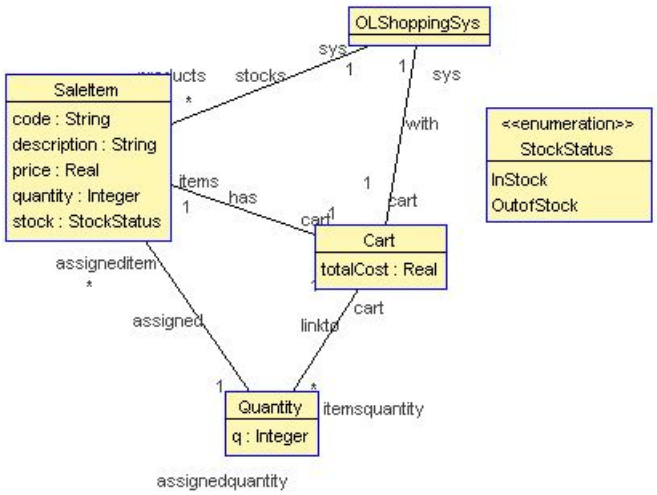

Fig. 8. Overview of class diagram

and "OutofStock". The three state schemas Cart, SaleItem and OLShoppingSys translate to class Cart, SaleItem and OLShoppingSys separately.

For the SaleItem state schema, in the declaration there are five basic attributes, type String/N/Real translate to String/Integer/double in UML/OCL. For Cart state schema, in the declaration, there are twovariables, one is items: SaleItem→N, items is a relation from SaleItem to natural number. In our example, we create a class Quantity that has an attribute q which type is Integer. The two class SaleItem and Quantity can be used to create the relation. The relation in UML/OCL is presented as three associations. Two of the associations are from Cart to SaleItem and to Quantity, both are many to many. Another association is from SaleItem to Quantity, which is one to one.

For the OLShoppingSys schema, there are two variables. For example, the products:P SaleItem can be translated to products: Set(SaleItem) as an attribute and create an association from Cart to SaleItem, the role name at SaleItem side must be products. The association has one to many multiplicity. The second varibale cart:Cart can be translated into an attribute cart:Cart and create an association from OLShoppingSys to Cart, the association is one to one. The next step for structure translation is to map the operation schemas to methods. In UML/OCL, method belongs to class. In the operation schema's declaration part, it states the 'Δ' or 'Ξ' as a prefix with the state schema names. Therefore the operation schema translates to the class the state schema related to. In the operation schema's declaration part, there may have input and output. All the inputs is translated to this method's input parameters, the type of output should be translated to this method's return type.

After structure translation, we perform the behaviour translation. In the online shopping system exanple, the main state schema has one predicate. It should be

translated to invariants in the relevant class. For example, the translation result for the " ∀p1, p2: products •p1 ≠p2 ⇔ p1.code≠ p2.code" and " dom cart.items⊆ products" is shown in Figure 9.

```
constraints
context OLShoppingSys
inv NoDuplicateProducts:
self.products->forAll(p1,p2|(p1<>p2 implies p1.code<>p2.code) and
                             (p1.code<>p2.code implies p1<>p2))

context OLShoppingSys
inv restrictsforitmes:
self.products->includesAll(cart.items)
```

Fig. 9. Translation result for invariants

After invariants translation, the next step is to translate operation schema's predicates into pre-/post-conditions in UML/OCL methods. For the AddItem operation schema, there are three pre-conditions and post-conditions. For example, the "p?∈products' maps to self.products->includes(p)", "p?.stock=InStock" maps to p.stock=StockStatus.InStock, and "p?.quantity>0" maps to "p.quantity>0". For the three post-conditions, in "cart'.items = cart.items ⊕{(p?↦ cart.items p? + 1)}", 'cart" is post-state of cart variable in Z, which maps to 'cart', and the cart is pre-state of cart variable in Z, which maps to 'cart@pre'.

Items is a relation in Z, which maps to items(SaleItem) and itemsquantity(Quantity) in UML/OCL. This post-condition can be divided into two parts, i.e., one for items and one for items quantity. The "cart.items p?' is aiming to find out related quantity with p?, and this quantity must be one instance of the cart.itemsquantity set in UML/OCL. We use "→any()" to obtain the quantity. It can be translated to "cart.itemsquantity→any(q|q.assigneditem=p).q= cart.itemsquantity→ any(q|q.assigneditem=p).q@pre+1". The second and third post-conditions can be mapped similarly.

For the DeleteItem operation, there are three post-conditions and one pre-condition. The result for DeleteItem operation schema is shown in Figure 10. For the pre-condition "p?∈dom cart.items" it should be translated to "cart.items→includes(p)". The "⩤" is translated to "→excludes()", "cart'.items={p?}⩤cart.items" maps to "cart.items→excludes(p)", then this post-condition should be translated to "DeleteItempost1". Other post-conditions are similar with the AddItem method's post-conditions.

For DeleteAll operation, there is only one complex post-condition. The translation result of this predicate is shown in Figure 11. Other operation schemas' predicates can be translated similarly according to the translation rules. Due to space limit, we will not list all of them here. After obtain the UML/OCL model, in the following sections, we will demonstrate how this model can be verified, simulated and validated against design model by using UML/OCL supporting tools.

```
context OLShoppingSys::deleteItem(p : SaleItem)
pre DeleteItempre1 : cart.items->includes(p)
post DeleteItempost1: if cart.itemsquantity->any(q|q.assigneditem=p).q@pre=1
                      then cart.items->excludes(p)
                      else cart.itemsquantity->any(q|q.assigneditem=p).q=
                           cart.itemsquantity->any(q|q.assigneditem=p).q@pre-1
                      endif
post DeleteItempost2: cart.totalCost=cart.totalCost@pre-p.price
post DeleteItempost3:self.products->exists(p1| p1.code=p.code and
                     self.products@pre->reject(x|x.code=p.code)=
                     self.products->reject(x|x.code=p1.code)
                     and self.products->excludes(p1)implies
                         self.products@pre->excludes(p)
                     and p1.description=p.description
                     and p1.price=p.price
                     and p1.quantity=p.quantity+1
                     and p1.stock=StockStatus::InStock)
```

Fig. 10. Translation result for DeleteItem

```
context OLShoppingSys::deleteAll()
post DeleteAllpost:
if cart.items@pre=Set{}
then
   (self.products=self.products@pre and self.cart=self.cart@pre)
else
   (self.cart.items->forAll(p|self.products->exists(p1|
    p1.code=p.code
    and p1.description=p.description
    and p1.price=p.price
    and p1.quantity=p.quantity+cart.itemsquantity
        ->any(q|q.assigneditem=p).q@pre
    and p1.stock=StockStatus::InStock))
    and cart.items=Set{}
    and cart.totalCost=0
)endif
```

Fig. 11. Translation result for DeleteAll post-condition

3.4. *Translating UML/OCL into Code with Java Assertions*

The next step is to validate the code from UML/OCL model. In order to do so we first generate code from the UML model and then insert the assertions for validation purposes. There are many different tools provide code generation from UML model. In this project, we just explored two candidates: Papyrus and OCLinEcore. They both support OCL and code generation. Papyrus can create UML models and OCL constraints graphically, which makes the UML/OCL model visible and easier to understand. It can also generate Java skeleton code from models and create an basic type of UML model file that can be reused in other tools such as OCLinEcore.

The reason for adding Java assertions and JUnit test cases to code is that to provide extra checking for at the implementation level. Whatever the implementation is, the assertions and test cases can always verify the code correctness. This is a novel contribution that is missing from other code generation approaches.

3.4.1. *Java Assertion*

After generating Java skeleton code using Papyrus, we translate OCL constraints to assertions to ensure program's correctness. A Java assert statement has two expressions: assert <boolean expression> and assert <boolean expression>:<error information>, if boolean expression is true then program can be continue else program will throw java.lang.AssertionError.

All of OCL constraints are boolean expressions, however, some of them can not be represented in Java assertions directly. For example, for a collection such as 'Set' in OCL, code generation tools usually translate it into a list in Java code. Thus methods in set can be expressed by list's operations, then use assertions to check this expressions (with keyword "assert"). The following provides translation guidelines from OCL to Java assertions.

Normal boolean expressions Normal boolean expressions that consist of relational/logical operators and operations on collection types can be mapped into their Java assertions counterparts.

- operators such as: <=,>=, <>,+ and -, are directly supported by Java.
- "=", "not": In OCL constraints "=" translates to "==" in Java and "not" translates to "!" in Java.
- →size(): is to calculate the number of elements in a collection. We use Java List to represent a collection type in OCL. List has a method "size()"to returns the number of elements in the list.
- →includes(object o): checks whether an object is in the collection or not. The "contains(Object o)" in List returns true if this list contains the specified element.
- →excludes(object o): based on the previous rule, "! contains(Object o)" means if the list does not contain the specified element it returns true, which is same with "→excludes(object o)".
- →includesAll(c2:Collection): is to check whether this collection contains all the elements in another collection c2, which is similar to "containsAll(Collection c2)"in Java.
- →excludesAll(c2:Collection): translates to "!containsAll(Collection c2)" in Java.
- →isEmpty(): presents as "isEmpty()" in Java, which will returns true if this list contains no elements.
- →union(c2:Collection): means the result will be a union of the referred collection with c2. In Java, "addAll(Collection c2)" has same meaning as the union.
- →intersection(c2:Collection): is to find out a set of all elements that are in both referred collection and c2. The "retainAll(Collection c2)" does the same in Java.
- - (c2:Collection): equals to the "removeAll(Collection c2)", which takes away

the elements in the referred collection, which are in c2.

Loops Apart from normal boolean expressions, there are quantifier related operators which may not have direct mappings into Java assertions expressions, such as:

- →exists(expr:OclExpression)
- →forAll(expr:OclExpression)
- →any(expr:OclExpression)
- →reject(expr:OclExpression)

These constructs above are working on collections that require to examine each individual object against the OCL expression. That is, the operations require to check every object in collection, so a loop structure would be a natural mapping for them. The actual OCL expressions are embedded inside the loop for checking individual values one by one.

For the online shopping system UML/OCL example, every methods has several pre/post-conditions, these pre/post-conditions all will be assert to check the implement. The invariants in class are also Java assertion. In online shopping system, there is two invariants in class OLShoppingSys("self.products→forAll(p1,p2|(p1<>p2 implies p1.code<>p2.code) and (p1.code<>p2.code implies p1<>p2)" and "self.products->includesAll(cart.items)"). The translation result from invariants to JUnit test case is in Figure 12. One invariant is to restrict every product's code and make them unique. Another is to restrict the cart items. These Java assertions have same meanings with the OCL constraints. For pre/post-conditions, pre-conditions can be translated to an assertion before this method is called and post-conditions can be translated to an assertion after the method is called.

```
public OLShoppingSys(){
    cart=new Cart();
    products=new ArrayList<SaleItem>();
    if(products.size()!=1){
        for(int i=0;i<products.size()-1;i++){
            for(int j=i+1;j<products.size();j++){
                assert !products.get(i).getCode().equals(products.get(j).getCode());
                }
            }
        }
    assert products.containsAll(cart.getItems());
}
```

Fig. 12. Invariants to assertions

We use the method AddItem to explain how to map OCL constraints into Java assertions. The "addItem" method has three pre-conditions and three post-conditions, as shown in Figure 13. These conditions can be translated to assertions

in Java code. Pre-conditions describe the value of a property at the start of the operation or the method. So the Java assertions translated from pre-conditions should be written in the front of the implementation of the operation or the method. Post-conditions denote the value of a property upon completion of the operation or the method. So the Java assertions translated from pre-conditions should be written behind the implementation of the operation or the method.

```
context OLShoppingSys::addItem(p : SaleItem)
pre AddItempre1 : self.products->includes(p)
pre AddItempre2: p.stock=StockStatus::InStock
pre AddItempre3: p.quantity>0
post AddItempost1: self.cart.totalCost=self.cart.totalCost@pre+p.price
post AddItempost2: if self.cart.items@pre->excludes(p)
                   then
                    (self.cart.items->includes(p) and
                     self.cart.itemsquantity.q->includes(1) and
                     self.cart.itemsquantity->any(q|q.assigneditem=p).q=1)
                   else
                    (self.cart.itemsquantity->any(q|q.assigneditem=p).q=
                     self.cart.itemsquantity->any(q|q.assigneditem=p).q@pre+1
                     and self.cart.itemsquantity->includes(
                         self.cart.itemsquantity->any(q|q.assigneditem=p))
                    )endif
post AddItempost3:self.products->exists(p1| p1.code=p.code
    and self.products@pre->reject(x|x.code=p.code)
        =self.products->reject(x|x.code=p1.code)
    and p1.description=p.description
    and p1.price=p.price
    and (if  p.quantity >1
         then
         (p1.quantity=p.quantity-1 and p1.stock=StockStatus::InStock)
         else
         (p1.quantity=0 and p1.stock=StockStatus::OutofStock)
         endif))
```

Fig. 13. Pre-/post-conditions for AddItem method

The three pre-conditions check the input variable 'p'. After the code generation, the products is a list of SaleItem. List has method contains() that has similar meaning with "→includes()", so we translate this constraint to assert products.contain(p). Other two pre-conditions are easy to map to 'assert p.getStock()==StockStatus.InStock' and 'assert p.getQuantity()>=0'. For post-conditions, 'cart.totalCost" maps to 'cart.getTotalCost()'. The first post-condition can be translated to 'assert cart.getTotalCost()==totalCostpre+p.getPrice()'. The second post-condition is an if-else-then-endif expression, Java assert does not have a function to express any loop. We use the if expression, and then check the expression in the loop. For the last post-condition, it has exist() which corresponds to a loop structure that aims to find out whether 'p1' exists. We constructed an if expression, such as 'if (products.get(i).getCode().equals(p.getCode()))'. The rest of the predicates is to map to other lines inside the if expression.

3.4.2. *JUnit Test*

OCL not only can be translated to Java assertion in Java code, but also can generate Junit test case. Junit test can test every method and class separately. The built-in assertion mechanism of JUnit is performed by the class 'org.junit.Assert'. It provides additional static methods than Java assertion. The main method in JUnit test is the "Assert.assertThat([reason,]T actual, Matcher<super T> matcher)", "reason" is a output when this assert fails; "actual" is the real result in the assertion; matcher is an matcher for the assertion. Its logic determines the actual objects whether satisfies assertion or not. For the invariant of class OLShoppingSys, we assume a situation after load all the products, every product's code should be unique, the test case is shown in Figure 14.

```
@Test
public void testOLShoppingSys() {
    if(products.size()!=1){
        for(int i=0;i<products.size()-1;i++){
            for(int j=i+1;j<products.size();j++){
                Assert.assertThat(products.get(i).getCode(),
                        not(products.get(j).getCode()));
                }
            }
        }
    Assert.assertTrue(products.containsAll(cart.getItems()));
    }
}
```

Fig. 14. Invariants test case

The 'self.products→forAll(p1,p2|(p1<>p2 implies p1.code<>p2.code)' and '(p1.code<>p2.code implies p1<>p2))' are two invariants. We create two for loops to compare all the values in the products' list. For every individual values of products list, the code should not be the same. The "Assert.assertThat(products.get(i).getCode(), not(products.get(j).getCode()))" tests if two object references do not point to the same object. The "assertTrue" has same meaning with keyword "assert", so this invariant can easily be translated from Java assertions in code to test cases.

In summary, by mapping OCL annotations on the UML model into corresponding Java assertions during the code generation process, we are able to provide extra quality assurance/checking on the derived program, as well as for the guided unit testing.

4. Formal Verification, Simulation, Validation and Checking

After the translations, we have two forms of the design that was derived from the Z formal model. In addition to the verification on the Z formal specification itself,

e.g., theorem proving with Z-EVES or model checking with ProB, we can further simulate and validate the transformed UML/OCL model with various tools. For the implementation level checking, we can perform validation and debugging with assertions. It is worth noting that the theorem proving and model checking at the Z specification level are complimentary to the simulation and validation at the UML/OCL level. In addition, the assertions at the implementation level further guarantee the code correctness.

4.1. *Theorem Proving and Model Checking with Z*

Z/EVE is a prover tool to demonstrating and verifying Z formal model. For verifying a Z model, system requirements can be described as theorems in Z/EVE, and Z/EVE can prove these theorems from Z model then to verify the design is correct. If the all theorems of system requirements are true, that means the model is correct. Z/EVE can not only automatic convert design model to theorem, but also can it prove the theorems written by customers.

theorem *AddItempre*
$\forall$ *OLShoppingSys; p?: SaleItem*
| $p? \in products \land p?.stock = InStock \land p?.quantity > 0 \bullet$ pre *AddItem*

Fig. 15. Theorem for pre-AddItem

For example, before online shopping system add one product into the cart, the product must be in stock. This theorem can be written as shown in Figure 15. The proof can be invoked by selecting the theorem in the specification window, right clicking, and selecting the proof window. If it results in the predicate true, returning to the Specification window, the status bar will be updated to show that the proof is complete. In addition to theorem proving [18], the Z specification can be exported as Latex form and loaded into ProB tool for simulation and model checking.

4.2. *Simulation and Validation with UML/OCL*

Now we have a UML model with OCL constraints, we can simulate it and dynamic validate the design by creating object instants and checking the static properties, invariants and pre/post-conditions.

The UML Specification Environment (USE) [10] is a simulation and validation tool for OCL specifications. After USE load the OCL model, it can display a view of the class diagram of the system. USE provides simulation facilities by creating objects and association instances of the model, as shown in Figure 16. The process of creating objects and simulating the model reflects a design validation. For the online

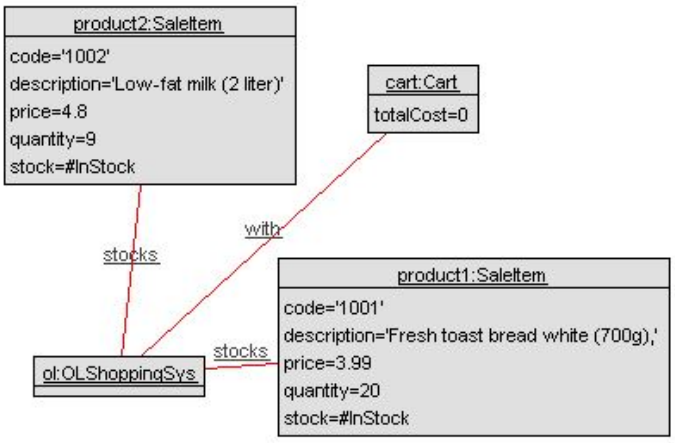

Fig. 16. Objects diagram

shopping system, we create one object named 'ol' for OLShoppingSys class and two objects named 'product1' and 'product2' for class SaleItem and one object 'cart' for class Cart, then set attributes to these objects. Object 'ol' has two attributes: cart and products, we link 'cart' to cart and '{product1,product2}' to products.

After creating the system states, USE can check the OCL invariants by class invariant view. In the example below, when we set 'products1' and 'product2' codes both to '1001', the result of invariants checking is false due to the unique code restriction. After we change 'product2' code to '1002', the checking result is true, as shown in Figure 17.

Fig. 17. Invariants checking result in USE

Pre/post-conditions can be validated in USE. It simulates the process of invoking the operation for validating the pre/post-conditions. Here is a example for the success and failure of pre-conditions of the 'DeleteItem' operation. As shown in Figure 18, if user want to delete an item from the shopping cart, the item must be in the cart. For instance, after adding 'product1' to cart, the user can not delete 'product2' from cart, the pre-condition will be false, due to the 'product2' has not been added to cart. However, the user can delete 'product1' from cart.

USE also has a functionality to record a sequence of operations by the sequence diagram view. This function serves as a simulation traces for the real user opera-

```
use> !openter ol deleteItem(product2)
precondition `DeleteItempre1' is false
Error: precondition false in operation call `OLShoppingSys::deleteItem(self:@ol,
 p:@product2)'.
use> !openter ol deleteItem(product1)
precondition `DeleteItempre1' is true
```

Fig. 18. Success and failure example for deleteItem

tions. As shown in Figure 19, the left diagram is a sequence of operations and the right diagram is a command record for these operation. We simulate a situation of customer shopping, firstly, this customer added one product1 to his/her cart, then this customer added this sale item again. He/she want to delete some items from cart, but he/she can not delete product2 because this item is not in the shopping cart. He/she changed his/her mind to purchase product1, so the customer delete all items in cart. He/she added another product2 to cart. Finally, this customer finished shopping and check out.

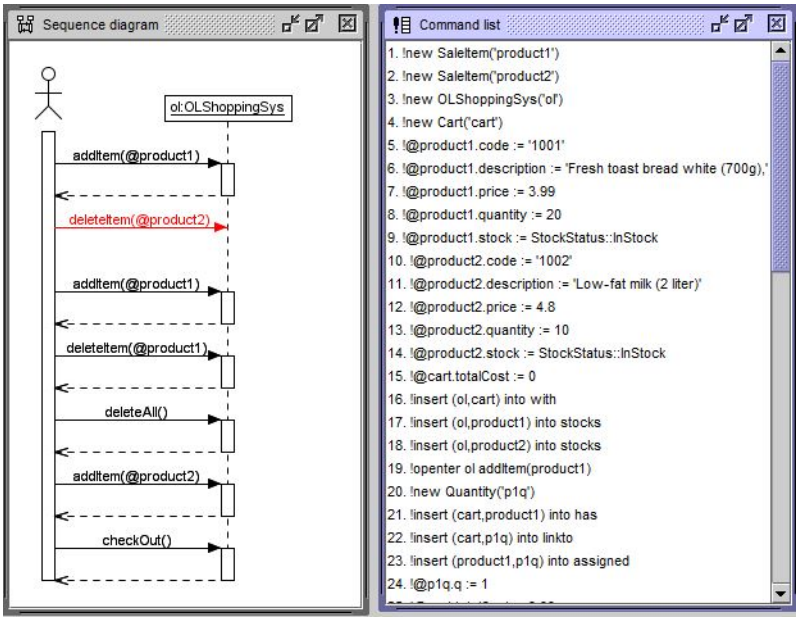

Fig. 19. Sequence of operations

4.3. *Code Level Checking with Assertions*

The reason that we translate OCL constraints to Java assertions and JUnit test is to validate and check the implementations of classes and operations. Java assertions

and JUnit test cases have same meaning with Z predicate and OCL constraints, and they can both check invariants and pre/post-conditions of a design model.

4.3.1. *Java Assertion*

Assertion is a normal debug style, and lots of languages support this function like C, C++, Eiffel and so on. In syntax, Java provides a keyword 'assert' to insert assertion. Java uses '-enableassertions/disableassertions' to open/close assertion function. When a programming closes assertion function, assertion expression will lose their effects. If assertion function is open, Java will calculate the expression value. If the value is false, this expression will throw an AssertionError.

Here is an example for using Java assertion in the online shopping system, which checks assertions in the addItem operation. The error denotes that input 'p' may not in the products list or currently out of stock, which was guaranteed in the assertion definition. If the item code does not meet this requirement, the system will not invoke the addItem method. For instance, we add a product (code is 1018) to cart, which is out of stock, the result will throw an AssertionError, as shown in Figure 20.

```
============================================================
------------------------------------------------------------
1. Add an item to the shopping cart.
2. Delete an item from the shopping cart.
3. Check out the shopping cart.
4. Exit without buying.
------------------------------------------------------------
Please enter your choice: 1
------------------------------------------------------------
Enter item code to buy: 1001
One more- Fresh toast bread white (700g) is added to the shopping cart
the items quantity in cart is:1

============================================================
------------------------------------------------------------
1. Add an item to the shopping cart.
2. Delete an item from the shopping cart.
3. Check out the shopping cart.
4. Exit without buying.
------------------------------------------------------------
Please enter your choice: 1
------------------------------------------------------------
Enter item code to buy: 1018
OutofStockException in thread "main"
java.lang.AssertionError: this product is out of stock
        at OLShoppingSys.addItem(OLShoppingSys.java:309)
        at OLShoppingSys.main(OLShoppingSys.java:128)
```

Fig. 20. AddItem operation assertions

Here is another example to check Java assertions in class. This asser-

tion "assert !products.get(i).getCode().equals(products.get(j).getCode().toString( )):"product is not unique";" is from one Z's state schema predicate and one OCL invariant. When we load two products that have some product code, the system will throw a exception, as shown in Figure 21.

```
Exception in thread "main" java.lang.AssertionError: product is not unique
        at OLShoppingSys.<init>(OLShoppingSys.java:56)
        at OLShoppingSys.main(OLShoppingSys.java:94)
```

Fig. 21. Wrong result for assertion in class

4.3.2. *JUnit Test*

JUnit testing can be performed independently of the the actual implementation. Eclipse supports JUnit test cases generated from Java code by right-click choose new JUnit test case. Users can also choose which classes and methods they want to test in the set. JUnit test case can be run by choosing to run as 'JUnit Test' in Eclipse. Several test be created cases for checking online shopping system's classes and methods, as shown in Figure 22.

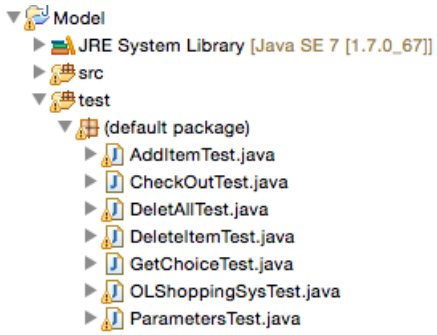

Fig. 22. Test cases for online shopping system

Here is an example for testing class OLShoppingSys, we choose to validate one class. This test case aims to validate predicate of state schema in Z and invariant of class in OCL. Because JUnit test cases' test method does not support method with parameters. We load the products list before testing OLShoppingSys class in this test case. The result will be successful. Then we change two product code to one same value, The result is shown in Figure 23. Green means test cases successful, Failure is test case failure and error is a mistake from program.

Let us exam an example of simulating a situation of adding items to the shopping cart, as is shown in Figure 24. This test case is not just testing the addItem method,

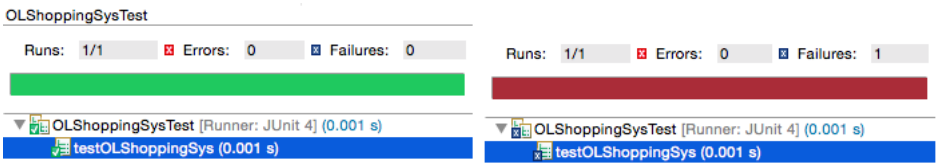

Fig. 23. Test case result for class

it also tests the getChoice method. It prints out the choices and wait for the user input. When the user inputs '1', the system will know that the user would like to add an item to cart, it will ask for input of the product code. If the item code exists in the product list, the result will be successful. In this test case, we can check two parts: one for input product code, another is for the input choice. If the product code is not valid or its stock status is not 'InStock', the result will be failure. If the input choice is not '1', which means the user does not want to add an item, the result will also be failure in this particular test case.

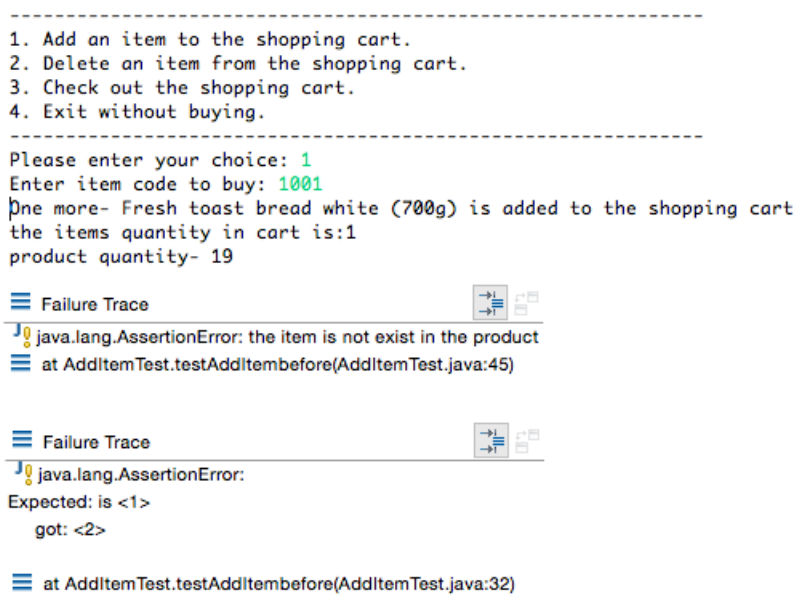

Fig. 24. JUnit testing for the AddItem method

Test cases can simulate more situations by just changing a little for an exist test case. We create a test case for checking deleteItem method. These test cases can be changed for lots situations in online shopping system such as when we do not add any items in cart, we run the test case. It will definitely fail because the

input product to be deleted has no chance in current cart. We can also simulate another situation such as we already added one product in cart, then there will be four possibilities for this test case's input, as one can see in Figure 25. If the input choice is '2', the result will be successful, if not , it will fail. If the input product is in shopping cart, then this result will success, if not, result will be a failure.

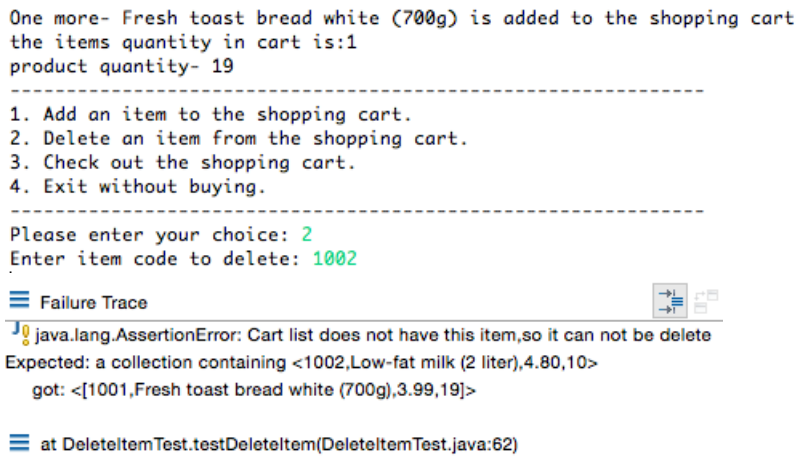

Fig. 25. Validation for deleteItem method

The above case study of the online shopping system aims to validate the feasibility of our approach. The associated tools, such as Z/EVE, ProB, USE, Java assertions and JUnit test cases also provide extra evaluations to the results of each steps in our approach. We use the runnable case to demonstrate all translations as well as verification results. Therefore, the case study provides the evaluation for our approach, which is feasible and effective.

5. Related work

Our project explore the transformation from Z to UML/OCL and UML/OCL to Java assertions to achieve the code generation. There are some other previous work and approach for code generation.

5.1. *Transformation between UML and Z*

Sun et al. [12] proposed the approach to develop a web environment for Object-Z and find out a web transformation between Object-Z and UML. They first extend Z to Object-Z in XML and then translate it to XMI in UML by using XSL. They also develop Object-Z to present precise design of web environment.

Kim and Carington [13] provided a formal semantic transformation between UML models and Object-Z specifications. They design a meta-model for Object-Z

and defines a formal structure for UML. This meta-model enhance the definition of the Object-Z and create a basic rule for mapping from Object-Z to other languages.

5.2. *Code Generation from UML*

Knapp and Merz [14] proposed the approach to generating code from UML states with formal analysis. The process reduces incorrectness in resulting system. They use HUGO to verify UML with a collaboration or sequence diagram. HUGO allows all features of UML state machines except for time and change events.

Long et al. [15] developed an algorithms to generate the Relation Calculus of object System(rCOS) code which is really similar with Java code from UML class diagram and sequence diagram. They also want to check the consistency from a class diagram to a sequence diagram. At the end of their research, they defined the semantics of the generated code and had a case study to prove it.

Grother and Schulze [16] defined a translation from design model to Aspect Code(AO). They choose AspectJ as their approach's target language. The code generation process is between UML model and AspectJ concepts.The UML model is write at CASE-Tool. It is available for user to validate design without generating code, after that, code generator will generate code from a validated aspect-oriented model.

For our approach, whatever the system requirements are, it can to be documented in Z model and translated to UML/OCL model. The UML model can automatic generate code with the help of exising tools, and OCL constraints can be translated to Java assertions. Java assertions act as extra checking facility to guarantee the qualify of the implementation. At the each level of translation, users can verify, simulate and validate the model to ensure correctness.

6. Conclusions

In this paper, we explore the synergy of automatically derive implementation from formal design models. We translate Z specifications into UML/OCL and map them into Java code. We use Java assertions to express OCL constraints at the code level. Java assertions can be used to generate JUnit test cases that are independent from the implementation and can be run repeatedly. Java assertions act as extra checking facilities to make sure the code correctness. At the each level of translation, user can verify, simulate, validate and debugging the model/code to ensure the quality of the final product.

Due to the limitation of time, some improvements for our project can be made in the future. We defined the translation guidelines from Z to UML/OCL, OCL to Java assertions. However, the translation is still manual based. It would be beneficial to develop a tool to provide the automated trans- formations. Since OCLinEcore and JUnit have Eclipse plug-in tools, and Z/EVE has an Eclipse plug-in version in CZT, there is a possibility to create an Eclipse plug-in tool that integrates the associated

tools into one coherent interface. Thus, the entire design modelling, transformation, code generation and testing could all be achieved in Eclipse.

References

[1] Monin, J. F. (2012). Understanding formal methods. Springer Science & Business Media.

[2] Vogel-Heuser, B., Witsch, D., & Katzke, U. (2005, June). Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In International Conference on Control and Automation (ICCA'05), Vol. 2, pp. 1034-1039. IEEE.

[3] Budinsky, F. J., Finnie, M. A., Vlissides, J. M., & Yu, P. S. (1996). Automatic code generation from design patterns. IBM systems Journal, 35(2), 151-171.

[4] Gray, J., Tolvanen, J. P., Kelly, S., Gokhale, A., Neema, S., & Sprinkle, J. (2007). Domain-specific modeling. Handbook of Dynamic System Modeling, 7-1.

[5] Hinchey, M. G., Rash, J. L., & Rouff, C. A. (2005). Requirements to design to code: Towards a fully formal approach to automatic code generation. NASA tech. monograph TM-2005-212774, NASA Goddard Space Flight Center.

[6] Massol, V., & Husted, T. (2003). Junit in action. Manning Publications Co.

[7] Hui Liang, Jin Song Dong, Jing Sun and W. Eric Wong. "Software Monitoring through Formal Specification Animation". *Innovations in Systems and Software Engineering: A NASA Journal*, Volume 5, Issue 4, pages 231-241, Springer, December 2009.

[8] Cabot, J., & Gogolla, M. (2012). Object constraint language (OCL): a definitive guide. In Formal Methods for Model-Driven Engineering (pp. 58-90). Springer Berlin Heidelberg.

[9] Chimiak–Opoka, J., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., & Willink, E. (2010). Workshop on OCL and Textual Modelling (OCL 2010). Electronic Communications of the EASST, 36.

[10] Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming, 69(1), 27-34.

[11] Bézivin, J. (2004). In search of a basic principle for model driven engineering. Novatica Journal, Special Issue, 5(2), 21-24.

[12] Sun, J., Dong, J. S., Liu, J., & Wang, H. (2001, April). Object-Z web environment and projections to UML. In Proceedings of the 10th international conference on World Wide Web (pp. 725-734). ACM.

[13] Kim, S. K., & David, C. (1999). Formalizing the UML class diagram using Object-Z. In «UML»'99—The Unified Modeling Language (pp. 83-98). Springer Berlin Heidelberg.

[14] Knapp, A., & Merz, S. (2002). Model checking and code generation for UML state machines and collaborations. Proc. 5th Workshop on Tools for System Design and Verification, 59-64.

[15] Long, Q., Liu, Z., Li, X., & Jifeng, H. (2005, April). Consistent code generation from UML models. In Software Engineering Conference, 2005. Proceedings. 2005 Australian (pp. 23-30). IEEE.

[16] Richters, M., & Gogolla, M. (2000). Validating UML models and OCL constraints. In «UML» '2000—The Unified Modeling Language (pp. 265-277). Springer Berlin Heidelberg.

[17] H. Zhu, J. Sun, J. S. Dong, and S.-W. Lin, "From Verified Model to Executable Program: the PAT Approach," *Innovations in Systems and Software Engineering*, pp. 1–26, 2015. [Online]. Available: `http://dx.doi.org/10.1007/s11334-015-0269-z`

[18] Scott Uk-Jin Lee, Gillian Dobbie, Jing Sun and Lindsay Groves, Theorem Prover Approach to Semistructured Data Design. Formal Methods in System Design, Volume 37, Issue 1, pages 1-60, Springer, November 2010. [Online]. Available: `http://dx.doi.org/10.1007/s10703-010-0099-4`