Towards Code Generation from Design Models

Pengyi Li*, Jing Sun†
*Department of Electrical and Computer Engineering
†Department of Computer Science
The University of Auckland, New Zealand
Emails: *pli552@aucklanduni.ac.nz, †j.sun@cs.auckland.ac.nz

Hai Wang
School of Engineering and Applied Science
Aston University, Birmingham B47ET
United Kingdom
Email: h.wang10@aston.ac.uk

***Abstract*—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 depended 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 explores the possibility 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.**

I. 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]. 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 humanpower into machine-power. However, the current software construction is still largely depended 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 [5], 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 and hence improve the overall quality of modern software development.

DOI reference number: 10.18293/SEKE2017-133

Currently, there are few theories being developed to implement the automatic code generation. Most related works are based on diagrammatic notations and generating partial code framework, such as UML to Java [2], [11]. Advantages of such an approach is that UML is nowadays the most commonly used notation for documenting design models and therefore this approach could become a widely acceptable and practical way in industry. However, this traditional model language is contributing relatively little to productivity [4]. Despite the informal description of the design notations, the generated programs are skeleton code only, which are not executable. Refinement [12] is another related field that aims at deriving less abstract models from formal specifications. However, to the best of our knowledge, there has been no work in refining a formal model directly into executable programs.

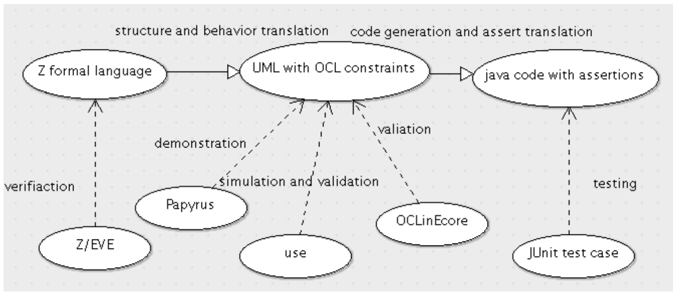

structure and behavior translation
code generation and assert translation
Z formal language
UML with OCL constraints
java code with assertions
demonstration
valiation
verifiaction
simulation and validation
testing
Papyrus
OCLinEcore
Z/EVE
use
JUnit test case

Fig. 1. Overview of our approach

In this paper, we explore the possibility 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. Figure 1 illustrates a summary of our approach.

Firstly, the design models are documented using the Z formal specification language [10]. Z is a first order logic and set theory based notation, which has a number of tools in supporting the verification and validation of on the design, such as Z/EVES, CZT, ProB, etc. Due to the high level abstraction of the Z language and its lacks 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 support the generations of skeleton code from UML to 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 generate 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 proving 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. And assertions can also be used together with JUnit for testing purposes.

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 concludes the paper and outlooks the future work.

II. TRANSLATING Z INTO UML/OCL

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

A. Structure Translation

1) 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 belongs to 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.

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

a) Given, basic and free types: Z uses [] to express an 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., nature 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.

b) 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 "#".

c) 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 represents 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: $A \rightarrow B$ in Z differs from a relation in the number of valid mappings through the relationship, i.e., for each $a \in A$ there is at most one $b \in B$. 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.

d) 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.

B. 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 structure translation, design specification need to be mapped by 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. 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 are part of a 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.

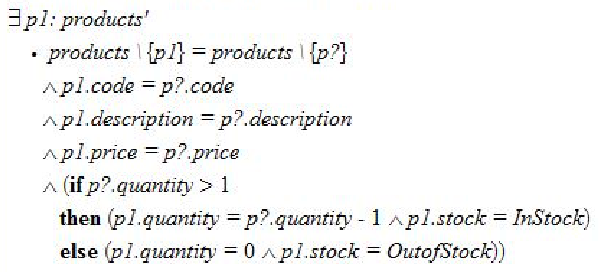
Finally, in order to translate the entire predicate into its OCL counterpart, we need to provide the mappings for various Z operators.

a) Logic and relational operators: suc as 'Not (¬),And ($\bigwedge$),Or ($\bigvee$),Implies ($\Rightarrow$) and Equivalence ($\Leftrightarrow$)' can be translated into 'not, and, or, implies and two direction implies in UML/OCL. When we translate A $\Leftrightarrow$ 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.

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

c) Quantifiers: Universal Quantifier ($\forall$) can be translated into the →forAll(...) operation in OCL; and the Existential Quantifier ($\exists$) can be translated into the →exist(...) operation in OCL. For example, '$\forall$p1, p2: products •p1 $\neq$p2 $\Leftrightarrow$ p1.code$\neq$ 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 this predicate is in the state schema OLShoppingSys, so it should be translated into invariants in OCL with context of the class OLShoppingSys.

Let us exam another more complete example in Z shown in Figure 2.



$\exists\, p1 : products'$

$\bullet\ products \setminus \{p1\} = products \setminus \{p?\}$

$\wedge\, p1.code = p?.code$

$\wedge\, p1.description = p?.description$

$\wedge\, p1.price = p?.price$

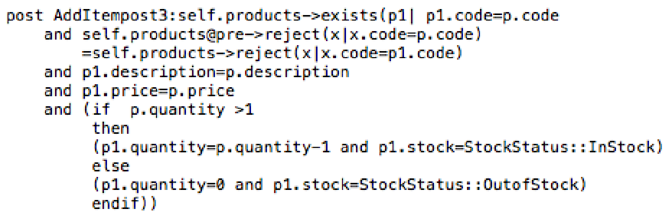
$\wedge\, (\mathbf{if}\ p?.quantity > 1$

$\quad \mathbf{then}\ (p1.quantity = p?.quantity - 1 \wedge p1.stock = InStock)$

$\quad \mathbf{else}\ (p1.quantity = 0 \wedge p1.stock = OutofStock))$

Fig. 2. Example for quantifies

Firstly, '$\exists$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 3.



```
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. 3. Translation result for quantifies example

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".

d) 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$\leftrightarrow$B and S $\subseteq$ A and T $\subseteq$ B; then S$\lhd$R is the set {(a,b):R|a$\in$S} R$\rhd$T is the set {(a,b):R|b$\in$T}. In UML/OCL, domain and range restrictions can be translated into invariant constraints imposed onto the association class. Relational composition: R : A $\leftrightarrow$ B and S : B $\leftrightarrow$ C R ; S ={(a,c):A$\times$C|$\exists$b:B•a R b $\wedge$ 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.

e) Function overriding: f,g:A→B;then f $\oplus$g is the function (dom g $\ntriangleleft$f)$\bigcup$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).

f) 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'.

Based on the above mapping rules, we can translate a Z formal specification into its corresponding UML/OCL model.

III. Translating OCL into 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.

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.

a) 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.

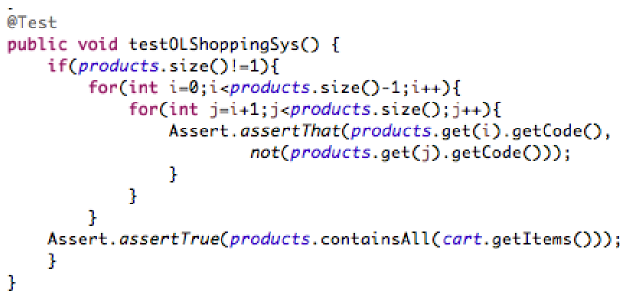
- operators such as: <=,>=, <>,+ and -, are directly supported by Java.
- "=", "not": In OCL constraints "=" translates to "==" in Java and "not" translates to "!" in Java.
- →size(): is to calculate the number of elements in a collection. We use Java List to represent a collection type in OCL. List 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.

b) 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.

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. It can also simulate any situation for the system. The built-in assertion mechanism of JUnit is provided by the class org.junit.Assert. It provides more static methods than java assertion. The most important 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 at assertion; matcher is an matcher for assertion, its logic determines the actual objects whether satisfies assertion or not. Let us see a simple example for creating Junit test case. 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 below at Figure 4.



```
@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. 4. Invariants test case

The two invariants are 'self.products→forAll(p1,p2|(p1<>p2 implies p1.code<>p2.code)' and '(p1.code<>p2.code implies p1<>p2))'. In this example, invariants want to check any two variable in products. We create two for loops to compare all the vaules in products' list. For every two values of products list, their code should not be the same. The "Assert.assertThat(products.get(i).getCode(), not(products.get(j).getCode()))" tests if two object references not point to the same object. "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.

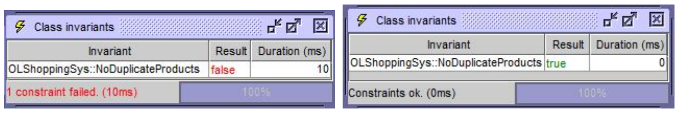
IV. Verification, Simulation and Checking

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.

A. Design level validation with 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) [9] 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.

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 5.



Invariant	Result	Duration (ms)
OLShoppingSys::NoDuplicateProducts	false	10

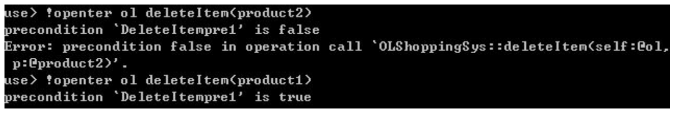
1 constraint failed. (10ms)

Invariant	Result	Duration (ms)
OLShoppingSys::NoDuplicateProducts	true	0

Constraints ok. (0ms)

Fig. 5. 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 6, if user want to delete an item from the shopping cart, the item must be in the cart. For instant, 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 operations.



```
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
```

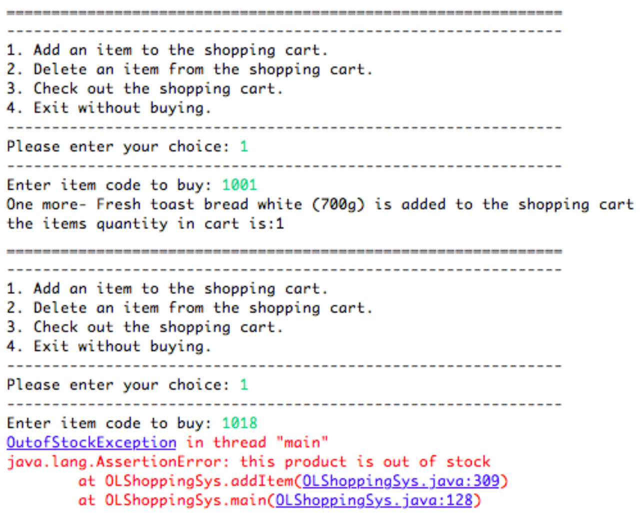
Fig. 6. Success and failure example for deleteItem

B. 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.

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, then java will calculate the expression value. If the value is false, then this expression will throw a AssertionError.

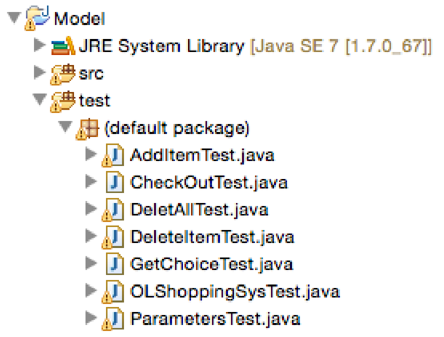
Here is an example for using Java assertion in the online shopping system, which checks assertions in the addItem operation. The error denotes that input ‘p’ may not in the products list or currently out of stock, which was guaranteed in the assertion definition. If the item code does not meet this requirement, the system will not invoke the addItem method. For instant, we add a product (code is 1018) to cart, which is out of stock, the result will throw an AssertionError, as is shown at Figure 7.



```
==============================================================
--------------------------------------------------------------
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. 7. AddItem operation assertions

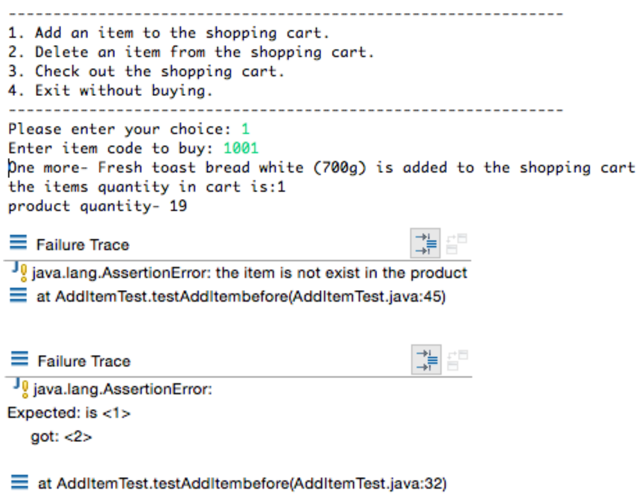
2) JUnit Test: JUnit testing can be performed independently of the the actual implementation. Eclipse supports JUnit test cases generated from Java code by right-click choose new JUnit test case. Users can also choose which classes and methods they want to test in the set. JUnit test case can be run by choosing to run as 'JUnit Test' in Eclipse. Several test be created cases for checking online shopping system's classes and methods, as is shown at Figure 8.



```
▼ Model
  ▶ JRE System Library [Java SE 7 [1.7.0_67]]
  ▶ src
  ▼ test
    ▼ (default package)
      ▶ AddItemTest.java
      ▶ CheckOutTest.java
      ▶ DeletAllTest.java
      ▶ DeleteItemTest.java
      ▶ GetChoiceTest.java
      ▶ OLShoppingSysTest.java
      ▶ ParametersTest.java
```

Fig. 8. Test cases for online shopping system

Let us exam an example of simulating a situation of adding items to the shopping cart, as is shown in Figure 9. This test case is not just testing the addItem method, 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.



```
----------------------------------------------------------------
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
product quantity- 19
```

```
Failure Trace
java.lang.AssertionError: the item is not exist in the product
at AddItemTest.testAddItembefore(AddItemTest.java:45)
```

```
Failure Trace
java.lang.AssertionError:
Expected: is <1>
     got: <2>

at AddItemTest.testAddItembefore(AddItemTest.java:32)
```

Fig. 9. JUnit testing for the AddItem 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.

V. Conclusions

In this paper, we explore the possibilities of automatically derive implementation from formal design models. We translate Z specifications into UML/OCL and then 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 transformations. 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 Control and Automation, 2005. ICCA'05. International Conference on (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] 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.

[10] 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.

[11] 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.

[12] 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.