Mapping USE Specifications into Spec#

Jagadeeswaran Thangaraj[1] and Senthil Kumaran U[2]

[1] DFAT, Dublin, Ireland
jagadeest@gmail.com
[2] School of Information Technology and Engineering,
VIT University, Vellore, TN, India
usenthilkumaran@vit.ac.in

Abstract. The UML model is easy to describe the object oriented program components clearly in graphical notation. OCL allows users to express textual constraints about the UML model. The USE tool allows specification to be expressed in a textual format for all features of the UML model with OCL constraints. Spec# is a formal language, which extends C# with constructs for non-null types, preconditions, post conditions, and object invariants. It allows programmers to document their design decisions in the code. Spec# has run time verifier to verify the specification constraints over the C# code. This paper describes the mapping of USE specifications into Spec# which helps to improve the quality of both UML/OCL and Spec#.

Keywords: USE, UML, OCL, Spec#.

1 Background & Motivation

The Unified Modelling Language(UML) model is easy to describe the object oriented program components clearly at the system design stage. The UML's class diagram depicts the details of a class of the model in an object oriented system. The relationship restrictions with other classes can be described by associations which are called UML constraints. Association multiplicities define the connection relation of classes to each other. Object Constraint Language(OCL) allows users to express textual constraints about the UML model [8]. So the UML class diagram with OCL constraints can describe all the elements of object program constructs with their specification.

The UML-based Specification Environment (USE) tool describes the program's specification at the specification level. The USE tool is based on a subset of UML and OCL. The USE tool allows specification to be expressed in a textual format for all features of a model, e.g., classes, attributes in the UML class diagrams. Additional constraints are written using OCL expressions [6]. The USE specification can easily convert to corresponding graphical representations using textual editor: Class diagram, Object diagram. Also it performs the verification of OCL constraint structures easily.

Spec# has run time verifier to verify the specification constraints over the C# code. Spec#'s specifications are not just comments, but those are executable [10].

In recent years, model based transformation is getting more popular [2] i.e.code generation from system design. At the moment, there is no explicit tool to generate Spec# code from UML/OCL.

In this paper, we map the UML/OCL properties of USE specification in order to generate Spec# code. Motivation behind this mapping is to find out what properties can add at the design phase in order to improve the quality of UML/OCL. In the same manner, this paper helps improving Spec# to support full UML/OCL properties. Remainder of the paper is organised as follows. Next section maps the properties between USE specification and Spec#. Class specification mapping is illustrated in section 2.1 and constraints in section 2.2. Unmapped properties are explained in section 2.3. Finally section 3 explains the conclusion of the mapping and recommended future works.

2 Mapping UML/OCL properties between USE and Spec#

For code generation, we need the corresponding references to the elements of both the source and the target languages [3]. This section presents the structural correspondence of USE(UML/OCL) and Spec#.

2.1 Mapping Class diagrams to Spec# Classes

This section explains the mapping of class properties between the USE specification and Spec#.

Primitive types: Integer, Real, Boolean and String [8] are primitive types in USE. The USE primitive types are directly mapped on predefined Spec# types. Thus the primitive USE types *Boolean, Integer, Real* and *String* are, respectively, mapped into *Boolean, Integer, Float* and *String* of Spec#.

Collection types: Collection types are used to group the elements together in some formal manner: *Set, Bag, OrderedSet* and *Sequence* [8]. The Spec# generic class, 'System.Collections.Generic.List' stores the elements in the format of *Sequence*. The OCL constraints are constructed on a UML diagram using these collection types, but, Spec# only supports the collection type *List*. The collection operations for all USE types are mapped into the corresponding operations of Spec#'s list.

Class & Enumeration: In an USE specification, class diagrams define the static characteristics of the system by specifying all classes, attributes and methods of each class and interrelations between the objects of these classes. In Spec#, whole program implementations mirror the role of the class diagram. The class construct is used to define all aspects of the class model with attributes, method

USE	*Spec#*

enum Color{silver, gold}
 	
class Customer
attributes
 	
name : String;
title : String;
isMale : Boolean;
age : Integer;
 	
operations
 	
age():Integer;
birthdayHappens();
 	
end
 	
class CustomerCard
end
 	
association holds between
Customer[1] role owner
CustomerCard [0..*] role cards
end
	public enum Color{silver, gold};
	public class Customer
	{
	[Rep][ElementsRep] List<CustomerCard>
	cards = new List<CustomerCard>();
	
	protected String name;
	protected String title;
	protected bool isMale;
	protected int age;
	public int age()
	{
	---- 	
	}
	public void birthdayHappens()
	{
	---- 	
	}
	
	}
	public class CustomerCard
	{
	
	[Rep]protected Customer owner;
	...
	
	}

Table 1. Class & Enumeration Representation

UML	*Spec#*

class Transaction
 	
attributes
points : Integer;
operations
earnPoints(points);
 	
end
class Burning < Transaction
end
 	
class Earning < Transaction
 	
earnPoints(points);
 	
end
	public class Transaction
	{
	[Additive] public int points;
	[Additive] public void earnPoints(int points)
	{ additive expose (this){ }
	}
	}
	public class Burning:Transaction
	{
	}
	public class Earning:Transaction
	{
	[Additive] public void earnPoints(int points)
	{ additive expose (this) { }
	}
	}

Table 2. Inheritance Representation

definitions, inheritance and association relations. In USE, operations of a class are represented after attribute declaration using the keyword *operations* as an example shown in Table 1. In Spec#, operations are written as standard method definitions.

Enumeration is used to hold the predefined constants to declared variables. The UML supports enumeration types using keyword *enum* [8]. The Spec# also supports the *enum* keyword as shown in Table 1.

Associations and Aggregations: An *association* describes the static relationship between the classes. In USE, associations are represented using the keyword *association* followed by the association name representing the link between the classes with role names. In Spec#, the associations are represented by constructing the objects of the association in the class definition. An association with multiplicity '1' is represented as a single object, and the association with multiplicity '*' is represented as a list of object declarations. References to other objects are represented with the ownership type annotations ([Rep] & [Peer]).

Any object can refer to other objects. Aliasing occurs when one object is reachable through multiple paths, i.e. more than one reference is referred by the same object. Ownership helps to control aliasing and assists in structuring object relationships in a program. By using this ownership representation, an owner object can access the reference objects. Ownership types help the programmer track information about object aliasing. Ownership types representation mainly specified in two types: Rep & Peer. Same ownership objects are represented 'peers' or 'siblings' [1]. Some objects are represented as reference of an owner object, are called 'reference' objects, i.e. an object can referred by owner. Sometimes multiple references can exist to an object. A `[Rep]` attribute which stands for representation [5]. `[ElementsRep]` specifies the '*' multiplicity as list of objects. An example is shown in Table 1.

Inheritance: Inheritance is an important concept in object-oriented design, which allows identical functionality of a class to be inherited into another class. New functionality can then be added to the class which inherits [7]. For example, `Burning` and `Earning` are subclasses of the `Transaction` class. Subclass is that may inherit the properties of superclass. In USE, inheritance is represented by the '<' operator.

In Spec#, inheritance is represented by the ':' operator. Subclasses attributes which need to access superclass attributes must be declared with [Additive] keyword [5]. If an object of a subclass needs to access attributes of its superclass, then those attributes must be annotated with the keyword `[Additive]`. In the example shown in Table 2, an attribute *points* is overridden in the `earnPoints()` method of the subclass `Earning`. Therefore, it needs to access the superclass `Transaction`'s attribute points. Therefore it is annotated as `[Additive]`.

USE supports multiple inheritance by comma as an example follows:

```
Earning < Transaction, Burning
```

Here, `Earning` class inherits from classes `Transaction` and `Burning`. As C#, Spec# does not support multiple inheritance.

2.2 Mapping OCL constraints to Spec#

Constraints are conditions or restrictions over a model or state. In USE, constraints are specified by Boolean expressions which must be side effect free. That means, the constraint must be evaluated to true or false and it must not change the state over the system. In a correct system, constraints must be evaluated to true. In USE, constraints are defined over various elements of the class diagram. *Invariants, preconditions* and *postconditions* are major constraints [8] which are specified by the operators *inv, pre* and *post*. These are checked via a validation process. These constraints are represented in Spec# as assertions. This section explains the mapping of these properties between the USE specification and Spec#.

Preconditions: A precondition is a condition that must be true before calling a method in a context in order to get the expected behaviour from the method. In Design by Contract (DBC), the method's client must meet the precondition. In a university, a student must be older than 23 years to enroll into a course as a mature applicant. This is described as a constraint as follows in Table 3.

OCL	*Spec#*

`context MatureProgram`
`::enroll(stu : Student)`
`pre: stu.age >23`
	`public class MatureProgram`
	`{`
	
	`public void enroll(Student stu)`
	`requires stu.age >23;`
	`{`
	`}`
	
	`}`

Table 3. Precondition Representation

The precondition declares that for any Student `stu`, who will be enrolled into a course as a mature student, his age must be greater than 23. The keyword `requires` is used to represent preconditions in Spec#.

Postconditions: A postcondition is a condition that should be true after executing a method in a context if the method behaves as expected when executed with a true precondition. In DBC, a designer establishes the postcondition. For example, as shown in Table 4, the method postcondition declares that the result of the method `enroll()` must add the student `stu` to the `MatureProgram` if

he/she has already not enrolled into the program. The keyword `ensures` is used to represent the postcondition in Spec#.

OCL	*Spec#*

`context MatureProgram`
`::enroll(stu : Student)`
`  post: numStudents =`
`  numStudents@pre + 1`
 	
`context MatureProgram`
`::allocate()`
` post: self.numPlaceAvail`
`  = self.numPlaceAvail@pre - 1`
	`public class MatureProgram`
	`{`
	
	`  public void enroll(Student stu)`
	`  ensures numStudents.Count`
	`  == old(numStudents.Count) +1`
	`  {`
	`  }`
	
	`  public void allocate()`
	`  ensures this.numPlaceAvail`
	`  == old(this.numPlaceAvail)-1;`
	
	`}`

Table 4. Postcondition Representation

The special property '@pre': In USE, `@pre` is used to hold previous value of an element before methods execution. The keyword `old` performs the same function in Spec#. An example follows in Table 4. In this, if a student enrolled into a course, the number of available places must be reduced by one.

Keyword 'self': In USE, the keyword `self` is used to refer the current instance of certain object of a class. The keyword `this` is used for the same function in Spec#.

Class Invariants: A class invariant is a condition that should be true during the entire life cycle of the class instances. That means, the class invariants must be hold for entire life of objects created. For example, a class invariant could be that a student must be more than 18 years old to enter into 3rd level education as shown in Table 5. The keyword `invariant` is used to represent the invariants in Spec# as shown in Table 5. During inheritance in Spec#, an overriding method may add additional postconditions with the superclass's preconditions and postconditions but cannot add new preconditions in order to keep the property of strengthening postconditions and weakening preconditions. A subclass may also strengthen the invariant.

2.3 Unmapped properties

Table 6 shows the correspondence of UML/OCL proerties between USE and Spec#. Based on this comparison, Spec# needs to define the generic collection

OCL	*Spec#*

`context Student`
`inv : age > 18`
	`Class Student`
	`{`
	`.....`
	`invariant age>18;`
	`......`
	`}`

Table 5. Class Invariant Representation

types (Set, Bag, Sequence) and Meta types (OclAny, OclExpression, OclType). Also it needs to define 'Collection' Operations. On the other hand, Spec# provides ownership type constraints (Rep, Peer) in association relations and Inheritance properties from one class to another to specify conditions using [Additive]. It has special feature, Non null types, that eradicates all non null dereference errors. In Spec#, type `T!` contains only references to objects of type `T`, which cannot be null.

UML/OCL properties	**USE**	**Spec#**
Precondition	pre	requires
Postcondition	post	ensures
Invariant	inv	invariant
Attributes	attributes	✓
Collection	Set, Bag, Sequence	List
Old	@pre	old
Quantifiers	✓	forAll, exists
Multiple Inheritance	✓	✗
OCL Types	✓	✗
Metatypes	✓	✗
Initial	✗	constructor
Derived	✗	✓
Non Null	✗	✓
Termination	✗	✗

Table 6. Correspondence of UML/OCL proerties between USE and Spec#
✓: Support & ✗: No Support

3 Conclusion

This paper has presented the mapping of USE specifications with Spec# for generating the Spec# code skeletons. It gives an idea to introduce some properties in software design and implementation towards to support the verification. Based

on our study, USE does not allow the addition of ownership type constraints (Rep, Peer) in software design phase. We have introduced these ownership type information to UML/OCL [4]. In this paper, we developed an approach to introduce ownership type constraints to USE specifications.

3.1 Future work

To support OCL directly, Spec# needs the collection operations. So our next aim is to generate a library to support the generic collection data types (Set, Bag and Sequence) and the different operations on the collection types (size, isEmpty, notEmpty, sum, count, includes and includesAll). Also our work will support Meta types like OclAny, OclExpression, OclType and OCL statements (OCLKindof, OCLTypeof).

References

1. Dave Clarke, Johan Östlund, Ilya Sergey and Tobias Wrigstad: Ownership Types: A Survey. In Aliasing in Object Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg: Springer, 15-58, ISBN: 978-3-642-36946-9, DOI:10.1007/978-3-642-36946-9-3, http://dx.doi.org/10.1007/978-3-642-36946-9-3 (2013)
2. Frank Hilken, Philipp Niemann, Martin Gogolla and Robert Wille: From UML/OCL to Base Models: Transformation Concepts for Generic Validation and Verification. In Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, held as Part of STAF 2015, L'Aquila, Italy, July 20-21, (2015)
3. Hiroaki Shimba, Kentrao Hanada, Kozo Okano and Shinji Kusumoto: Bidirectional Translation between OCL and JML for Round-Trip Engineering, Software Engineering Conference (APSEC, 2013) 20th Asia-Pacific: IEEE 49 - 54 (2013)
4. Jagadeeswaran Thangaraj, SenthilKumaran U: Introducing Ownership Type constraints to UML/OCL. In International Workshop on Aliasing, Capabilities and Ownership, IWACO17 co located with the 31st European Conference on Object-Oriented Programming, ECOOP 2017. (Barcelona, Spain, Jun 2017)
5. K. Rustan M. Leino, Peter Müller : Using the Spec# language, methodology, and tools to write bug-free programs, LASER Summer School 2007/2008: Springer-Verlag, (2008)
6. Martin Gogolla, Fabian Buttner, Mark Richters: USE: A UML-based specification environment for validating UML and OCL, In Science of Computer Programming (2007): Elseveir, 27 – 34 (2007)
7. Mike Barnett, Rustan Leino, Wolfram Schulte: The Spec# programming system: An overview. In Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg, 49–69, ISBN:978-3-540-30569-9, DOI:10.1007/978-3-540-30569-9-3,http://dx.doi.org/10.1007/978-3-540-30569-9-3 (2005)
8. OMG: Object Constraint Language(OCL):Version 2.3.1.Object Management Group,http://www.omg.org/spec/OCL/2.3.1 (2012)
9. OMG: Unified Modeling Language(UML):Version 2.4.1, Object Management Group, http://www.omg.org/spec/UML/2.4.1, (2011)
10. Rosemary Monahan, K. Rustan M. Leino: Program Verification using the Spec# Programming System, ECOOP Tutorial (2009)