Introducing Ownership Type Constraints to UML/OCL

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. In an object oriented program, 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 for verification purpose. Ownership types help the programmer track information about object aliasing. This paper aims to introduce ownership types information to UML/OCL for design specification. This helps the implementations easier to develop and less prone to error.

Keywords: Aliasing, Ownership, USE, UML, OCL, Spec#.

1 Introduction & Motivation

In recent years, model based transformation is getting more popular [3], i.e. code generation from system design. The Unified Modeling Language (UML) model makes it 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 [10]. So the UML class diagram with OCL constraints can describe all the elements of object program constructs with their specification. At the moment, UML/OCL does not allow mentioning the object references with ownership type in the current context directly. In this paper, we explicitly allow the reference of other object by adding ownership types to the UML/OCL, so that we can implement further with no bother about ownership type constraints.

1.1 Motivation

Nowadays, software is developed via automatic code generation from software designs to implementation when using formal specification and static analysis to reduce the development effort [4] [7]. The modelling approaches are used to describe the client's specification.

In a program implementation, we document objects and those objects which they own that means have exclusive write access. We refer to these objects as the "owned objects". It is important to know what objects an object owns for purpose of static verification. The correct software maintains the consistency of a program's data throughout its verification. If it fails to maintain the consistent details about ownership, the system may fail and lead to a number of errors during program development. If we know the information about the ownership during the design phase, our implementation will be easier and less prone to errors.

This paper makes this information available in the software design phase to improve the quality of design specification. It presents ownership type constructs at the software design phase for dealing with aliasing in programming languages. Then it transfers these ownership type constructs to the implementation phase for actual development and practical evaluation of these constructs. We chose the USE (The UML-based Specification Environment) specification to describe the program's specification and Spec# to develop the code at the implementation level.

2 Background

2.1 USE

The USE tool, which is based on a subset of UML and OCL, 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 [8].

The USE specification describes the program's specification at the specification phase. The reason behind this selection is its feature, that is written in the text format and can easily convert to corresponding graphical representations using textual editor: Class diagram, Object diagram. Also it performs the verification of OCL constraint structures easily. In the text format of USE specification, we can add the ownership constraints as comments with no changes made in USE tool. Therefore it makes it easy to implement the ownership addition in their specification.

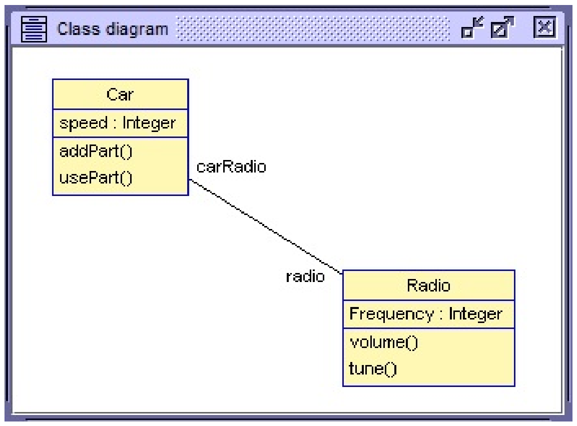
UML Model Specification: Every UML *model* in USE has a name and an optional body. A model may contain *Enumerations* and *Classes*. Each class has a name. It may has optional *attribute* and *operation* definitions. Classes can be linked together via associations. It is possible to define *Association multiplicities* and role names along with *Association definitions*. Fig.1 shows an USE specification of 'CarSystem' and corresponding UML diagram generated in USE tool.

```
model CarSystem

class Car
attributes
speed: Integer;
operations
addPart();
end

class Radio
attributes
Frequency: Integer;
operations
volume();
end

association Music between
Car[0..1] role carRadio
Radio[0..1] role radio
end
```



```
constraints
context Car
inv: self.maxSpeed <= 180
```

Fig. 1. An USE Specification and Corresponding Class Diagram

Constraints: The constraints segment of a specification follows after the keyword *constraints* in USE specifications. Any number of *invariants* may be defined in a class context. In addition, we may define *preconditions* and *postconditions* to specify the conditions over operations. We can add names for every constraint in the constraint definition segment.

2.2 Spec#

We chose Spec# to develop the code at the implementation level. The reason behind the Spec# selection is that provides support for encoding ownership relationship to tackle the aliasing [6]. 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 [13].

Dynamic ownership in Spec#: Formal specifications are mathematically based techniques which are used to ensure the correctness of software by precisely expressing a program's properties. These are not executable specifications. Specification properties are typically simple safety properties, *non functional* properties or full behaviour properties. Nowadays, number of tools and languages has been introduced for formal specifications e.g., Key system for JML verification [1].

Dynamic ownership systems enable ownership transfer in the *expose blocks* during program execution. Dynamic ownership has been implemented in the

Spec# language [5]. This dynamic ownership is supported by three major constructs: Object topology, ownership types and representation exposure. In Spec#, an object can refer to other objects for the internal definition of its data. The `[Rep]` keyword is used to annotate such attributes. Therefore the `this` object is declared as the owner of `Rep` referenced objects. Generally an invariant is a constraint of a type over an element of the model, i.e., expressed by the OCL expression [10]. Object invariant is a constraint of the object during its instance. Object invariants must be true all the time for an object instance. During execution of a Spec# program, it is necessary to break some object invariants for the purpose of verification [12]. Therefore Spec# introduces a block statement called *expose block*. Invariants are temporarily broken by exposing an object using the *expose* construct. The object invariants may be broken within an expose block [9] i.e., the object invariant cannot be proved as a logically true inside the expose block. In the expose block, an owner is mutable. Therefore the current owner is the owner of the referenced object. At the end of an expose block, the object invariant must hold. This ownership transfer supports the program verification in the dynamic ownership system.

2.3 Properties of Ownership Types

In this paper, ownership types representation mainly specified in three annotations: Rep, Peer & Additive. Same ownership objects are represented 'peers' or 'siblings' [2]. Some objects are referred as reference of an owner object, are called 'reference' objects. Additive is used in specification inheritance. These are explained in detail as follows.

Rep: 'Rep' [14] expresses that a referenced object is owned by current object, that is, if a class context has a 'Rep' reference then 'this' objects is the owner of referred object. This enables one owner which can access other objects to modify during the verification. If the 'Rep' field refers number of objects as array, then each element in that array can hold by this owner.

Peer: 'Peer' expresses that the owner is same [14] for current object and reference object. The current object and the referenced object share the same owner and are therefore in the same ownership context or same aggregate [5]. These objects have equal relationships. That means, the class has a reflexive association of it as 'Peer'. If 'Peer' field refers number of objects then those elements express as the array of the peer objects.

Additive: Specification contracts can be inherited in Spec#. Spec# supports specification inheritance by *strengthening* postconditions and class invariants and *weakening* preconditions. Therefore we can add additional postconditions and invariants which specify properties of superclass attributes. If an attribute can be overridden in a subclass, this must be highlighted in the superclass. Spec#

introduces the `[Additive]` keyword to highlight the attributes those mention in the subclass invariants. To access superclass attributes or methods, `[Additive]` ownership is used. An *additive expose* is needed in method inheritance.

3 Our approach

In this paper, we introduce an approach, called U2S#, which allows the specification of references to other objects and expose blocks during the software design phase. Then U2S# helps to generate the code skeleton with ownership details and expose blocks inserted in the correct place in the implementation code. An object's property is accessed by other objects mainly during constraint specification. The objective of this paper is to highlight references to other object during the specification phase and transform the corresponding ownership type constraints to the implementation. This section describes the modifications that we provide in the USE specification language and the support which we add to the USE tool to allow addition of ownership details.

3.1 Adding Ownership type constraints to USE

U2S# adds the ownership annotations of Spec# to the given UML/OCL model based provided by the USE tool. To add ownership type constraints in the USE tool, we introduce a new grammar for the definitions of Attribute, Operation and Association. The modified syntax is shown in next sections.

Association Syntax: An association is an inter relationship between two `classes` or `models` in a UML diagram. It shows the logical or physical combinations or links of instances of those models in some formal manner [11]. An association relationship in UML describes the `role names` between the classifiers and number of objects acts as `role`. The main challenge here is the addition of ownership type constraints to UML. Within a current context, a graphical dot in the association link is used to denote the ownership. Standard UML notation does not allow the specification of explicit ownership. But in this paper, we introduce the ownership type notion which allows accessing the object of another class.

In this paper, ownership types are specified using two keywords in the association: `[Peer]` and `[Rep]`. This corresponds to the Spec# syntax for the same. In our USE specifications, association ends have the provision to specify these ownership type constraints. Normally association relations are represented in the USE tool by naming association names, classes and role names. This paper introduces a new keyword `ownership` followed by `ownershiptype` to name ownership type constraints as shown in the following syntax.

Syntax: `<associationdefinition> ::= ( association|composition|aggregation)` `<associationname>` *between*

```
<classname>[<multiplicity>][role<rolename>][ordered][--ownership<ownershiptype>]
<classname>[<multiplicity>][role<rolename>][ordered][--ownership<ownershiptype>]
end
<multiplicity> :== (*| <digit> { <digit> }[..(*| <digit>{<digit>})])
{,(*| <digit> {<digit>}[..(*| <digit> { <digit>}])}
<associationname> :== <name>
<rolename>:==<name>
<ownershiptype>:==(Rep|Peer)
```

Attribute and Method Syntax: Like association definition, this paper introduces the `[Additive]` keyword in the definition of attributes and operations as comments. Each attribute is followed by `--[Additive]` if it is an `additive element` which will be inherited by its subclasses. In the subclasses, the inherited operations are represented by the `[Additive]` keyword. If attributes and methods are not additive, then they are represented as empty followed by a semicolon.

Syntax: `<classdefinition> ::=[abstract]class<classname>[< <classname>`
```
{,<classname>}]
[attributes { <attributename>:<type>--[Additive]}]
[operations: {<operationdeclaration>--[Additive] . . . }]
end
<classname>::=<name>
<attributename>::=<name>
```

In our U2S# approach, the ownership annotations can be specified in the USE specifications as comments, based on the modified grammar. Therefore we can generates the Spec# code skeleton. When generating the Spec# code skeleton, U2S# takes the ownership types: `[Rep]`, `[Peer]` and `[Additive]` as input. It also takes association relation's multiplicities into account.

4 Adding Ownership type constraints to UML/OCL and Mapping to Spec#

This section explains the addition of ownership type constraints according to the modified grammar of USE specifications as discussed in section 3. In a U2S# implementation, a given UML model is transformed into its corresponding Spec# code skeleton. U2S# adds the correct ownership types to the UML model according to the client's requirements for the relationship between classes and attributes. This is achieved via annotations to the USE specifications. U2S# deals with three major ownership types: `[Peer]`, `[Rep]` and `[Additive]`. Therefore U2S# adds these ownership type annotations as comments.

4.1 Ownership addition with Association Ends

As discussed in section 3.1, an association relation between the classes plays an important role in determining ownership type. Ownership types are referred by the keywords `[Peer]`, `[Rep]` and `[Additive]`. In the USE specifications, *association ends* normally record the property details such as association names, classes involved and role names. In addition, we add the ownership type with *association end* as an example in Table 1.

Representation in USE	*U2S# approach*

`association holds between`
`Customer[1] role owner`
`CustomerCard [0..*] role cards`
`end`
	`association holds between`
	`Customer[1] role owner--ownership [Rep]`
	`CustomerCard [0..*] role cards--ownership [Peer]`
	`end`

Table 1. Ownership Representation in Association Ends

In U2S# approach, it adds the ownership type followed by each *role name* as comments. The corresponding representation with ownership types is shown in the right side of Table 1. Here, the role name `owner` is the `Rep` owned object of class `CustomerCard`. In same manner, the role name `cards` is the `Peer` owned object of class `Customer` in `CustomerCard`. As discussed in section 2.2, it is not necessary that object invariants evaluate to be true through out on execution. Therefore, Spec# supports the introduction of a frame called *expose block*. Object invariants do not need to evaluate to true within an expose block [9]. At the end of each expose block, the invariant must hold. `Rep` objects indicate that these are two owners. Therefore an expose block must be present in the implementation. But `Peer` object indicates that the object belongs to same owner. Therefore it does not need an expose block in its implementation.

4.2 Ownership addition on Inheritance

We add the `[Additive]` annotation for each class attributes and operations as comments in USE to specify these additive properties. Each attribute is followed by `--[Additive]` if it is an additive elements which will be inherited by subclasses. In subclasses also, the inherited operations are represented by the `[Additive]` keyword to denote ownership. If attributes and operations are not additive then they are left as empty as in the example in Table 2.

This code has two classes: `Customer` and `CustomerSon`. The operation `addMoney` is overridden in the subclass and have access to its superclass operation and attributes. Therefore they mentioned as `[Additive]`.

Representation in USE	*U2S# approach*
`class Customer`	`class Customer`
`attributes`	`attributes`
`name : String;`	`name : String;`
`amount : Integer;`	`amount : Integer;--[Additive]`
`operations`	`operations`
`addMoney():Integer;`	`addMoney():Integer;--[Additive]`
`end`	`end`
`class CustomerSon < Customer`	`class CustomerSon < Customer`
`attributes`	`attributes`
`---`	`---`
`operations`	`operations`
`addMoney():Integer;`	`addMoney():Integer; --[Additive]`
`end`	`end`

Table 2. Ownership Representation in Inheritance

5 Conclusion

This paper has presented an approach, named U2S#, for generating the Spec# code skeletons by adding ownership type constraints to UML/OCL at the design phase of software development.

5.1 Properties supported by U2S#

U2S# supports the following properties:
1. *Ownership type* constraints can be added during the software design phase
2. *Additive* constraints can be added during the software design phase
3. It helps to generate the *Spec# code skeletons* with correct ownership type constraints for actual development in the implementation phase.
4. The Spec# code skeletons will have the *expose* and *additive expose* blocks in the right place to avoid ownership exposure errors.

5.2 Results

U2S# allows users to specify the ownership type constraints at the software design phase. It avoids complicating code development i.e., tracking the ownership type constraints in the code implementation phase. U2S# ensures the consistency of ownership types during code generation. It helps to transform correct ownership type in the target language according to its specification at the design phase. U2S# ensures the consistency of program elements during code generation.

References

1. Bernhard Beckert, Reiner Hähnle, Martin Hentschel, Peter H. Schmitt: Formal Verification with KeY: A Tutorial. In Volume 10001 of Lecture Notes in Computer Science Programming and Software Engineering, Springer (2017).
2. Dave Clarke, Johan Östlund, Tobias Wrigstad: Ownership Types: A Survey. In Aliasing in Object Oriented Programming. LNCS: Springer, 15-58 (2013).
3. 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, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, (2015).
4. Hiroaki Shimba, Kentrao Hanada, Kozo Okano and Shinji Kusumoto: Bidirectional Translation between OCL and JML for Round-Trip Engineering. In Software Engineering Conference (APSEC, 2013) 20th Asia-Pacific: IEEE 49 - 54 (2013).
5. K. Rustan M. Leino, Peter Müller: Using the Spec# language, methodology and tools to write bug-free programs. In LASER Summer School 2007/2008: Springer-Verlag, (2008).
6. K. Rustan M. Leino and Peter Müller: Object Invariants in Dynamic Contexts. In ECOOP Object-Oriented Programming, 491-515 (2004).
7. L. Carnevali, D. D'Amico, L. Ridi, E. Vicario: Automatic Code Generation from Real-Time Systems Specifications. In International Symposium on Rapid System Prototyping, IEEE/IFIP (2009).
8. Martin Gogolla, Fabian Büttner, Mark Richters: USE: A UML-based specification environment for validating UML and OCL. In Science of Computer Programming (2007): Elseveir, 27 – 34 (2007).
9. Mike Barnett, Rustan Leino, Wolfram Schulte: The Spec# programming system: An overview. In CASSIS 2004: Springer (2004).
10. OMG: Object Constraint Language(OCL: Version 2.3.1. Object Management Group, http://www.omg.org/spec/OCL/2.3.1 (2012).
11. OMG: Unified Modeling Language(UML: Version 2.4.1. Object Management Group, http://www.omg.org/spec/UML/2.4.1 (2011).
12. Peter Müller: Modular Specification and Verification of Object-Oriented programs. PhD thesis, Fern Universität Hagen, Germany (2001).
13. Rosemary Monahan, K. Rustan M. Leino: Program Verification using the Spec# Programming System. In ECOOP Tutorial (2009).
14. Werner Dietl, Sophia Drossopoulou, Peter Müller: Separating ownership topology and encapsulation with Generic Universe Types. In ACM Transactions on Programming Languages and Systems: ACM:20:1–20:62, (2011).