C# 3.0 makes OCL redundant!

D.H.Akehurst[1], W.G.Howells[1], M. Scheidgen[2], K.D.Mcdonald-Maier[3]

[1] University of Kent, Canterbury, UK
{D.H.Akehurst, W.G.J.Howells}@kent.ac.uk
[2] Institut für Informatik, Humboldt-Universität zu Berlin, Germany
scheidge@informatik.hu-berlin.de
[3] University of Essex, Essex, UK
kdm@essex.ac.uk

Abstract. Other than its 'platform independence' the major advantages of OCL over traditional Object Oriented programming languages has been the declarative nature of the language, its powerful navigation facility via the iteration operations, and the availability of tuples as a first class concept. The recent offering from Microsoft of the "Orcas" version of Visual Studio with C# 3.0 and the Linq library provides functionality almost identical to that of OCL. This paper examines and evaluates the controversial thesis that, as a result of C# 3.0, OCL is essentially redundant, having been superseded by the incorporation of its advantageous features into a mainstream programming language.

1 Introduction

The thesis of this paper is intended to be a controversial statement that will generate discussion at the OCL4All workshop. The statement is actually more general than that implied by the title; C# is not the first or only language to include features that render it very similar to OCL. However, seeing as it is one of the latest and possibly highest profile languages to do so, it is interesting to consider its impact on OCL.

For example, consider the potential statement by a C# programmer:

> "Why should I use or learn OCL when I can write concise constraints using C#?"

As most readers of this paper will be aware, OCL is the textual expression language part of the OMG's UML related standards. Without OCL, much of the precision and expressiveness of Modelling using the UML set of languages would be lost.

At its core, OCL is basically a navigation language. In the context of an environment of variables, OCL expressions can be written that navigate, from starting variables, across properties and operations to other objects. All expressions can be seen simply as navigations in this manner. Such expressions can be used for a number of purposes within a UML specification, defining for example, invariants, pre/post conditions, or operation bodies.

The primary differences in expressive power or brevity, between OCL and other OO programming languages, have come from the declarative nature of the language and facilities for closure (or lambda expression) style iterations, tuples, and explicit collection initialisation.

Now that these facilities are starting to be seen in mainstream programming languages, this paper asks the question “how is OCL effected?”, and intends to raise the issue of how the OCL language community should respond in the context of the new languages that include many of the OCL features.

The paper proceeds, in section 2, by first reminding us of the formal origins of the major concepts associated with OCL. This is followed by a comparison, between OCL and C#, of the new language features in Sections 3, 4 and 5. Conclusions and significant discussion points are given in Section 6.

2 Background

2.1 Programming without Loops

Imperative programming uses loops as control structures which determine repeated execution of commands. They are good at describing program flow but are not necessarily good at expressing queries over data structures. Unfortunately, they are often the only construct in most imperative programming languages that can be used to apply the same program code to the different elements of a collection. Collections are represented as part of the program state (data). When a collection is iterated using a loop construct, the current state of iteration is part of the program state.

Functional programming [1] uses functions as first class concepts. Functions are used to iterate through collections by means of recursion. Collections are represented by terms over functions like *nil* (empty list *[]*) and *cons* (appending an element to a list *(:)*). Collection functions can be applied to the elements of a collection by using higher-order functions. Higher-order functions are, simply stated, functions that use other functions as arguments or return other functions as results. One example is *fold(f, z, list)* which takes a function f and folds it in between the elements of a list. Other examples of higher order functions are *map* or *filter* [1].

In usual procedural or object-oriented programming languages, a function or method has a fixed context. In procedural languages, functions are defined globally where there is only one environment and all functions use the same global variables. In object-oriented programming, methods are defined within classes and executed within objects which may access the objects member variables. A procedure or method may change its context, and therefore change the program state.

An expression in functional programming is referentially transparent, i.e. it can be replaced by the value it represents. The only environment a function is provided with are the parameter arguments given by an enclosing function.

The example OCL expression:

```
self.ownedElements->forAll(m| m.name != "foo"))
```

uses the expression:

```
m.name != "foo"
```

as an argument to the forAll operation. During evaluation of this expression, the expression `n.name != "foo"` is paired with the environment given by the enclosing function (the `forAll` operation). This environment provides all elements of the collection `self.ownedElements`, which is referenced through the parameter `m`. The concept of functions enclosed in other functions paired with an environment provided

by the outer function is known as *closure*. Function definitions like the `m.name != "foo"` expression are often referred to as *closures*. Closures are the basis for most functional programming languages, such as Lisp, Haskell or query languages like OCL or SQL.

A similar (but mathematical crude, untyped, and unsafe) mechanism is used in many interpreted or script languages. These languages, often provide an `eval` statement, which allows the execution of a piece of code which is provided as a string containing the program code. In a way, these languages treat pieces of program code (strings) as first class objects. Probably the most prominent of those languages is TCL [2], where simply everything, objects, primitive values, and of course code are just strings.

Even though most procedural and object-oriented programming languages do not treat functions as first class objects, there are often ways to mimic the closure concept using the existing constructs of those languages.

Some existing approaches

The first author has implemented the OCL collections library in Java (available on request). This implementation uses anonymous classes to realise closures and provides the standard OCL iterator operations.

FunctionalJ [3] is a general functional programming API for Java. It uses anonymous classes to realise closures. Those anonymous classes implement simple `function` interfaces, which only define one `execute` method. This method provides the closure for the function to be defined. Functions defined as anonymous classes are connected to a concrete environment by using the parameters of `execute`. Remember that, in the OCL example above, `m` was used to access the collection elements. The variable `m` would be realised using an according parameter. FunctionalJ uses generic parameters in its closure interfaces to allow type safe parameters. Unfortunately, Java methods can only have a statically defined number of arguments. FunctionalJ therefore defines several closure interfaces `function1`, `function2`, …, with different numbers of generic parameters and `execute` methods with the same number of parameters. It is possible to combine closures to realise advanced functional programming concepts, such as folding.

The Jakarta Commons Collections [4] library follows a similar approach, but tailors it for collections. There are several specialisations of function interfaces that realise specific closures. For example `predicate` which defines a function with Boolean return type and is used to define Boolean expressions over collections. The library works as an extension to the original java.util library. Closures are not applied directly to collections: the library provides several iterator functions, such as `collect`, `select`, or `forAll` as static methods of a collection utility class. The Jakarta Commons has no generics support and is therefore not statically type safe.

There are discussions about extending the Java language with closures. Groovy [5, 6], even though it is a new language, can be seen as such a closure featuring variation of Java. Groovy represents closures as anonymous blocks of code. Closures can be assigned to variables or used as arguments. Groovy closures are typed and can be used like any other object or value. There are other languages that successfully combine object-oriented programming with functional programming; the Ruby language [7] is an example.

2.2 Tuples

Tuples were introduced into the OCL language in 2001 influenced by publications such as [8]. A tuple is basically an un-named (anonymous) type, than can be instantiated as needed within an expression.

A tuple is a very well-known and used concept from mathematics. Generically, a tuple is a simple syntactic sugar which represents a conjunction of two or more types to form a single compound type. In this general case, two objects $a:A$ and $b:B$, may be combined to form a conjunctive type $(a,b):(A \wedge B)$ where (a,b) is known as a two-tuple of the two objects *a* and *b*. In the general case, the types of these objects may be arbitrary and may include further tuples although the component items are treated as independent sub-items. To illustrate this, note that the three-tuple $(a,b,c):(A \wedge B \wedge C)$ is of distinct type to either of the two-tuples $((a,b),c):((A \wedge B) \wedge C)$ or $(a,(b,c)):(A \wedge (B \wedge C))$ both of which are tuples which contain further tuples as component items.

The concept of tuples has been present within most Declarative programming languages since their inception although it is also typical for equivalent concepts, often termed along the lines of *Records*, to be present within Imperative languages although often lacking the conciseness of syntax associated with the tuples of Declarative languages and possibly requiring the components of the tuple to be named.

3 Inferred Types

OCL let statements have always had the semantics of allowing the type of the declared variable to be inferred from the expression assigned to it. Mainstream typed programming languages have usually not allowed this. C# 3.0 does, using the 'var' keyword.

```
let
   x = address.person.name
```

Table 1 OCL inference of variable type

In this OCL expression, the type of the variable x is inferred from the type of the expression – in this case, assuming the 'name' property is a string, the type of the variable x is inferred to be a string.

```
var x = address.person.name
```

Table 2 C# inference of variable type

Likewise in this C# statement, the type of the variable x is inferred from the resulting type of the navigation expression.

4 Collection Initialisation

Initialising collections is not a particularly exciting concept. It is simply a declarative way of stating the existence of (or constructing) a collection object

containing certain other objects. As a declarative language, this is of course essential in OCL.

```
seq = Sequence { 1, 2, 3, 4 }
```

Table 3 OCL initialisation of a Sequence

In the past, imperative programming languages have provided array initialisers, but not generally provided a means to initialise user defined collection classes. Instead, collection elements have had to be explicitly added. The notion of a variable number of arguments passed to a constructor (as exemplified by Java 5 in Table 4) has given a means to do this, but, oddly, such constructors have not been provided by the standard collection classes (in Java).

```
class MyListImpl<E> implements List<E> {
  public MyListImpl(E ... elements) { … }
  …
}

List<Integer> seq = new MyListImpl( 1, 2, 3, 4 );
```

Table 4 Java 5 initialisation of a List

In C# 3.0, we can now initialise objects of any class that implements the IEnumerable interface, and which provide an 'Add' method. This enables us to create collection objects in a manner almost identical to OCL, as illustrated in Table 5.

```
var seq = new List<int> { 1, 2, 3, 4 }
```

Table 5 C# initialisation of a List

C# 3.0 also facilitates object initialisation, which could be a feature provided in OCL, though this does raise the issue of 'side-effect-free', is creating a new tuple or a new collection object any different to creating a new object of a user type?

5 Iterations

The main expressive power and conciseness of OCL comes from the iterator operations. There are many of these built into the language: iterate, select, reject, collect, any, one, forAll, exists, etc. A few are illustrated in Table 6.

```
let
  seq = Sequence { 1, 2, 3, 4, 5, 6, 7, 8, 9 },
  evenSeq = seq->select( i | i mod 2 = 0 ),
  allEven = seq->forAll( i | i mod 2 = 0 ),
  existsEven = seq->exists( i | i mod 2 = 0 ),
  sum = seq->sum()
```

Table 6 OCL iterations

C# 3.0 has introduced a language concept of a 'lambda expression' that now facilitates the use of iterator operations in the same manner as OCL. The equivalent C# statements to the OCL of Table 6 are shown in Table 7

```
var seq = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var evenSeq = seq.Where( i => i % 2 == 0 );
var allEven = seq.All( i => i % 2 == 0 );
var existsEven = seq.Exists( i => i % 2 == 0 );
var sum = seq.Sum();
```

Table 7 C# iterations

As we can see from Table 7, there is very little difference between the C# and OCL expressions. There are subtle syntax differences, and the names of the operations are different (i.e. *Where* replaces *select*, and confusingly (for OCL experts), *Select* replaces *collect*).

A very useful syntactic notion that OCL has, which C# 3.0 does not, is the automatic implication of *collect* when navigating over a collection property using the '.' operator. This gives a significant conciseness to navigation expressions, not present in the C# 3.0 approach, at least as yet!.

6 Tuples

```
let
  t1 = Tuple { name = 'john', age=37 }
  t2 = Tuple { a = a1, b = b1 }
  t3 = Tuple { a:A = a1, b:A = b1 }
```

Table 8 OCL definition of tuples

Tuples in OCL are constructed using the keyword *Tuple*, followed by a list of "name=value" pairs. In addition we can explicitly provide the type of the named part of tuple.

```
var t1 = new { name="john", age=37 };
var t2 = new { a = a1, b = b1 };
var t3 = new { a = a1, b = (A)b1 };
```

Table 9 C# definition of tuples

In C# 3.0, the equivalent of a tuple is provided using the notion of an anonymous class. The *new* keyword is used to construct an object with no specified type; the initialiser for the object is used to imply the property names and types, and give the properties a value.

In OCL we can explicitly define the type of a tuple, allowing tuples to be passed as operation parameters for instance; or simply to facilitate validation of the tuple type of an expression. Table 10 illustrates this.

```
let
   t1 : Tuple(name:String, age:Integer)
       = Tuple { name = 'john', age=37 }
```

Table 10 OCL use of tuple type

In C# however, we cannot explicitly define the type of the anonymous class properties, they are always inferred. We cannot therefore use tuples as parameters to operations, other than by employing a pass by example work-around, which may not be possible in the final version of the language. We can explicitly give the type of the properties by including a cast to the required type.

```
var t1 = new { name=(string)"john", age=(int)37 };
```

Table 11 C# cannot explicitly define the type of the tuple

7 Conclusion

The stated thesis of this paper was that OCL is redundant now that new language features are present in C#. That is to say, given that people can write nice concise navigation expression using iterators and tuples in the C# programming language, why would anyone want to write them in OCL?

An initial argument to counter that thesis is the idea that OCL is in some way 'platform independent'. Using a programming specific language, such as C#, ties the specification to a Microsoft/.Net/C# implementation of the specification. If we write the specification using the 'standard' language of OCL, then we can use MDD techniques to provide alternative implementations. The new C# features thus makes a mapping to that language much simpler.

Both C# 3.0 and Java 7 both propose the integration of closures or lambda expressions into the core language. This suggests that the OCL community could also make that step. Having included the notion in the syntax of the language, since its beginning, as built-in iterator operations, perhaps we should promote the notion to a first class concept as has been done in C# 3.0 and Java 7. This would then enable users to define their own operations that make use of the concept.

To summarise, and to initiate some discussion points, the following three lists highlight: reasons that OCL is not redundant, OCL concepts that programming languages such as C# now have, and potential improvements for OCL.

OCL is still useful because:

1. It is platform/programming language independent
2. It enables concise navigation over collections

OCL concepts now in C# are:

1. Tuples
2. Iterator operations
3. Lambda Expressions
4. Collection initialisation
5. Inference of variable types

Questions regarding future of OCL:

1. Is OCL redundant? *No, we need something platform independent!*
2. Should we enable users to write their own closure operations? I.e. provide explicit notion of lambda expressions? *Yes, this would be very useful!*
3. Should OCL allow Object Initialisation, or is this definitely seen as causing a side-effect? *Open to discussion.*

References

[1] R. Bird and P. Wadler, *Introduction to functional programming*: Prentice Hall, 1988.

[2] J. K. Ousterhout, *Tcl/Tk*: Addison-Wesley, 1997.

[3] F. Daoud and Javelot Inc, "FunctionalJ" functionalj.sourceforge.net. 2006

[4] T. M. O'Brian, *Jakarta Commons Cookbook*: O'Reilly, 2004.

[5] K. Barclay and J. Savage, *Groovy Programming; An Introduction for the Java Programer*: Morgan Kaufmann, 2006.

[6] C. Castalez, "Closures in Groovy; Good Fences Make Good Functions" today.java.net. 2005

[7] Y. Matsumoto, "Ruby Programming Language," Addison Wesley Publishing Company, 2002.

[8] D. H. Akehurst and B. Bordbar, "On Querying UML data models with OCL," presented at <<UML>> 2001 - The Unified Modeling Language: Modelling Languages, Concepts and Tools, 2001.