Translating UML-RSDS OCL to ANSI C

K. Lano, S. Yassipour-Tehrani, H. Alfraihi
Dept. of Informatics, King's College London
S. Kolahdouz-Rahimi,
Dept. of Software Engineering, University of Isfahan, Iran

Abstract. In this paper we describe a transformation from a subset of OCL to ANSI C code, and we show that the generated code provides improved efficiency compared to OCL execution using Java, C# or C++. The transformation is itself formally specified using OCL.

1 Introduction

In this paper we describe a transformation that maps a substantial subset of OCL 2.4 to ANSI C. C has limited expressiveness compared to more modern languages such as Java or C#, but it has the benefits of high efficiency and small code size. The generator can be used as a 'virtual machine' for the execution of UML/OCL, as an alternative to the more usual Java/JVM implementation route for OCL [11]. The generator is itself an example of a large scale application defined using OCL, it consists of 125 operations and transformation rules defined by OCL constraints.

The OCL to C translator is a subtransformation of a UML to C code generator, UML2C, for the UML-RSDS MDE language [6]. UML2C maps instance models of the UML-RSDS class diagram (Figure 1), OCL (Figure 3) and activities metamodels to C language metamodels (Figures 2, 4 and a C statement metamodel). We target the 1989 ANSI C standard [4].

The mapping of OCL expressions depends upon the prior mapping of types and class structures, however it is relatively independent of the strategy chosen for representing classes in C (eg., how inheritance and dynamic dispatch is expressed), since all access to objects and their features is via an interface of creators, lookup operations and getters/setters which have a standard signature independent of their implementation details. For example, any instance-scope attribute f of class C is accessed via operations $getC_f$ and $setC_f$, for both owned and inherited attributes (cf., Table 3, case F1.2.5). The application API is defined in the header file app.h for each application. A library ocl.h of C functions for OCL operators is also defined, and evaluation/execution of particular OCL expressions is based upon app.h and ocl.h.

1.1 UML-RSDS

UML-RSDS enables applications to be defined using class diagrams, use cases, constraints and activities (pseudocode). It is similar to fUML [9] in being a

subset of UML, however, unlike fUML, it is oriented to declarative specification, with OCL constraints being used to define use cases and operations by pre and post conditions, instead of activities. The UML-RSDS tools can synthesise a procedural platform-independent design from such specifications, and this design is then mapped to program code by code generators (3 generators exist for Java versions, and there are C++ and C# generators in the latest UML-RSDS version 1.7 at nms.kcl.ac.uk/kevin.lano/uml2web).

Design synthesis operationalises the declarative specification. If an OCL expression is used in a specification to specify an update, eg.: by a postcondition $s \rightarrow includes(x)$, this is transformed into an activity, such as $s := s \rightarrow including(x)$, by the UML-RSDS design generator.

Specifiers are recommended to optimise their application functionality *at the specification level*, eg., by using let-variables to avoid duplicated expression evaluations. These optimisations then apply regardless of the eventual target platform. Optimisation is also performed during the design synthesis stage, eg., to use bounded loops instead of fixpoint iteration where possible [6].

The tools have been extensively used since 2006, particularly in the financial domain and for defining transformations. There are a number of restrictions and variations in the language compared to full UML and OCL (Table 1). We have found these variations helpful in simplifying specifications and improving the capability for verifying specifications.

UML/OCL	*UML-RSDS subset/variant*
Ternary associations, Multiple inheritance	Omitted
General n..m multiplicities on association end	Only 1, 0..1, or * multiplicities permitted
Integer type Real type null, invalid OclMessage, Tuple	int, long computational numeric types double computational type Omitted Omitted
Implicit conversion of single elements to collections	Omitted. 0..1 association ends are treated as collections
4-valued logic General *iterate* OclAny, oclType()	Classical 2-valued logic Omitted Omitted
	Lookup of objects by primary key value E[value] Additional collection operators $\rightarrow sort()$, $\rightarrow front()$, $\rightarrow tail()$, etc

Table 1. Differences between UML-RSDS and UML

Collections are assumed not to contain null elements. String-valued attributes can be declared as *identity* attributes, ie., as primary keys for a class. Classes with a subclass must be abstract.

Minor syntactic variations are the use of $\Rightarrow$ for OCL *implies*, $\rightarrow exists1$ for $\rightarrow one$, and & for *and*. $E.allInstances$ is abbreviated to E when used as the LHS of a $\rightarrow$ operator. $s \rightarrow includes(x)$ can also be written as $x : s$, $s \rightarrow includesAll(x)$ as $x\ <:\ s$, and $s \rightarrow excludes(x)$ as $x\ /:\ s$.

1.2 Paper structure

Section 2 describes the mapping of types and class structures to C, Section 3 describes the OCL expression mapping, Section 4 gives an evaluation, Section 5 describes related and future work, and Section 6 gives conclusions.

2 Mapping of types and classes

Figure 1 shows part of the UML-RSDS class diagram metamodel. This is closely based upon UML 2.2. Instances of this metamodel are mapped to instances of a C metamodel by UML2C. The base target language is a simplified version of the abstract syntax of C programs (Figure 2).

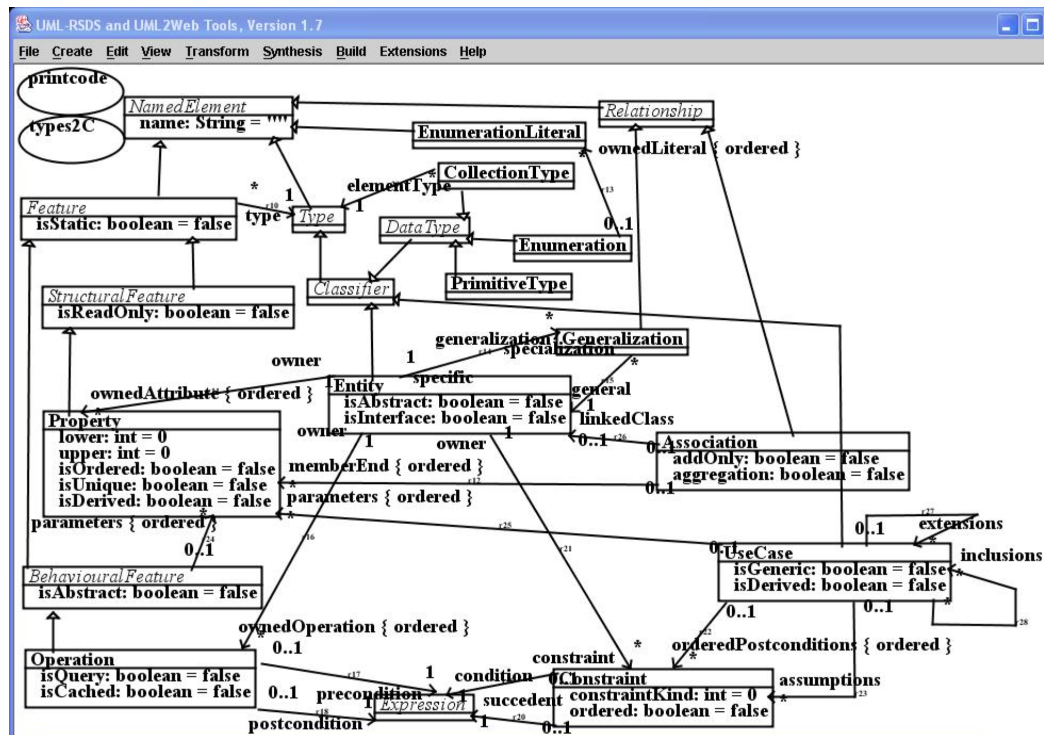

Fig. 1. UML class diagram metamodel (subset)

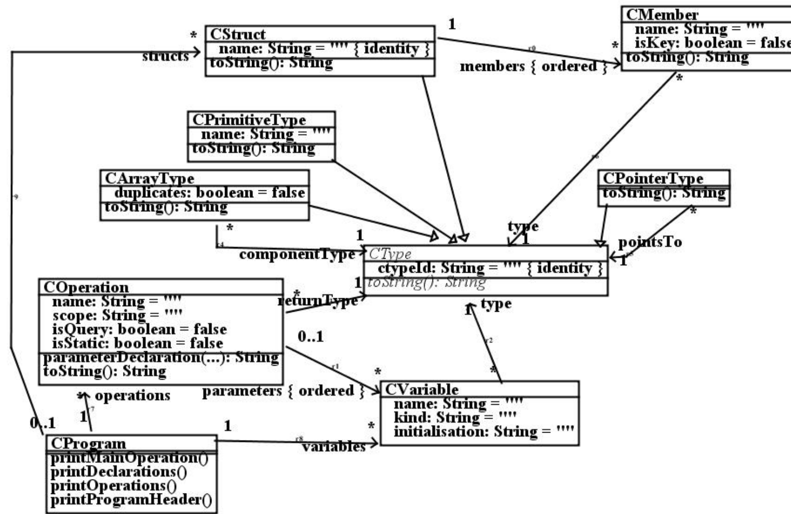

Fig. 2. C language metamodel

Table 2 shows the informal mapping of UML *Types* to C. The T* operator directly interprets Collection(T), for sequences and ordered sets of string and entity types T. Collections of collections can be mapped down to 2 levels (eg., Sequence(Sequence(double)) for matricies is mapped to double**). Unordered sets and bags are implemented as binary search trees.

To achieve bidirectionality and traceability of the transformation, a new identity attribute $typeId : String$ was introduced into *Type*, and *ctypeId* into *CType*. This enables *Type* and *CType* instances to be looked-up by key value: $CType[id]$ is the C type instance t with $t.ctypeId = id$. An instance $t : Type$ corresponds to an instance $c : CType$ if $t.typeId = c.ctypeId$.

An example transformation rule of the UML2C transformation, formalising case F1.1.1.1 from Table 2, is:

```
PrimitiveType::
  name = "String"  =>
    CPointerType->exists( p | p.ctypeId = typeId &
         CPrimitiveType->exists( c | c.name = "char" & p.pointsTo = c ) )
```

This rule applies to objects $self : PrimitiveType$. Whenever the lhs of the rule is true, the rhs is made true, ie., the relevant C types are looked-up or created if they do not already exist. The semantics of $E \rightarrow exists(e \mid e.eId = v \ \& \ P)$ in the case that *eId* is an identity attribute of E is that the E object $E[v]$ with *eId* value equal to v is looked up, if it exists, and is then modified according to P. If the object does not exist, it is created and then modified.

Classes, features and inheritances are mapped to C as shown in Table 3.

Case	*UML/OCL element*	*C representation* e'
F1.1.1.1	*String* type	`char*`
F1.1.1.2	int, long, double types	same-named C types
F1.1.1.3	boolean type	`unsigned char`
F1.1.2	Enumeration type	C `enum`
F1.1.3	Entity type E	`struct E*` type
F1.1.4.1	*OrderedSet*(T) type	`T'*` (NULL-terminated array of T', without duplicates)
F1.1.4.2	*Sequence*(T) type	`T'*` (NULL-terminated array of T', possibly with duplicates)
F1.1.4.3	*Set*(T) type	BSTs of T' elements without duplicates
F1.1.4.4	*Bag*(T) type	BSTs of T' elements possibly with duplicates

Table 2. Informal mappings of types to C

Case	*UML element* e	*C representation* e'
F1.2.1	Class diagram D	C program with D's name
F1.2.2	Class E	`struct E { ... };` Global variable `struct E** e_instances;` Global variable `int e_size;` `struct E* createE(void)` operation
F1.2.3.1	Instance property $p : T$ (not principal identity attribute)	Member `T' p;` of the struct for p's owner, E where `T'` represents T Operations T' $getE_p(E'\ self)$ and $setE_p(E'\ self, T'\ px)$
F1.2.3.2	Principal identity attribute p : *String* of class E	Operations $getE_p$, $setE_p$, struct E* getEByPK(char* v) Key member `char* p;` of the struct for E
F1.2.4	Operation $op(p : P) : T$ of E (non-static)	C operation `T' op_E(E' self, P' p)` with scope = "entity"
F1.2.5	Inheritance of A by B	Member `struct A* super;` of struct B Operations $getB_att(x)$ for inherited *att* invoke $getA_att(x \rightarrow super)$ Operations $op_B(x, p)$ for inherited *op* invoke $op_A(x \rightarrow super, p)$ unless redefined in B
F1.2.6	Operation $op(p : P) : T$ of E (static)	C operation `T' op(P' p)` with scope = "entity"

Table 3. Informal mapping of UML class diagrams to C

For each entity type E, getters and setters for each feature of E are produced, together with creation and deletion operations createE and killE, and lookup operations getEByPK, getEByPKs in the case that E has a principal primary key (identity attribute). These form the object API for E. Operations for OCL collection operators acting on collections of E instances are also generated: collectE, selectE, rejectE, intersectionE, unionE, reverseE, frontE, tailE, asSetE, concatenateE, removeE, removeAllE, subrangeE, isUniqueE, insertAtE. An operation opE is only generated for OCL operator *op* if there is an occurrence of $\rightarrow op$ applied to a collection of E elements in the source UML/OCL specification model.

2.1 Mapping of associations and polymorphic operations

We have found that the most complex parts of UML to code mappings are typically: (i) managing object deletion; (ii) maintaining the consistency of opposite association ends. Additionally for C, expressing inheritance and dynamic dispatch are further complex aspects. Deletion and association management operations are created during design synthesis. If an association has both ends named, then these ends need to be maintained in consistency. For example, a *–* association between classes A and B, with ends ar, br will have synthesised design operations

```
A::
static addA_br(ax : A, bx : B)
activity:
  ax.br := ax.br->including(bx) ;
  bx.ar := bx.ar->including(ax)

A::
static removeA_br(ax : A, bx : B)
activity:
  ax.br := ax.br->excluding(bx) ;
  bx.ar := bx.ar->excluding(ax)
```

and similarly for other association multiplicities. Deletion operators *killE* for concrete E are also inserted into the design, these manage the deletion of aggregation part objects linked to the deleted object, and the removal of the object from all association ends. The UML2C generator therefore generates C declarations and code for these operations.

General schemes for representing inheritance in C include an embedded superclass struct instance in each subclass struct, and function pointers for each supported method, or the use of vtables for function pointers. We use a pointer member `struct E* super;` referring from a subclass F to its superclass E.

Dynamic dispatch of an abstract operation $op(p : P) : Rt$ of class E with leaf subclasses A, B, ... is carried out by a C operation *op_E* with the schematic definition

```
Rt' op_E(struct E* self, P' p)
```

```
{ if (oclIncludes((void**) a_instances, (void*) self))
  { return op_A((struct A*) self, p); }
  else if (oclIncludes((void**) b_instances, (void*) self))
  { return op_B((struct B*) self, p); }
  else ...
}
```

This explicit selection of the correct implementing operation corresponds to the semantic model of polymorphic operations used by the UML-RSDS verification tools[1].

3 Mapping of UML-RSDS OCL expressions to C

Figure 3 shows the UML-RSDS OCL metamodel, which is the source language for the transformation. Figure 4 shows the corresponding C expression language abstract syntax. New identity attributes *expId* and *cexpId* are added to *Expression* and *CExpression*, respectively, to support bidirectionality and traceability requirements. *variable* : *String* represents iterator variables x for the cases of $s \rightarrow forAll(x \mid P)$, etc. A $* - *$ association *context* from *Expression* to *Entity* is used to record the context(s) of use of the expression.

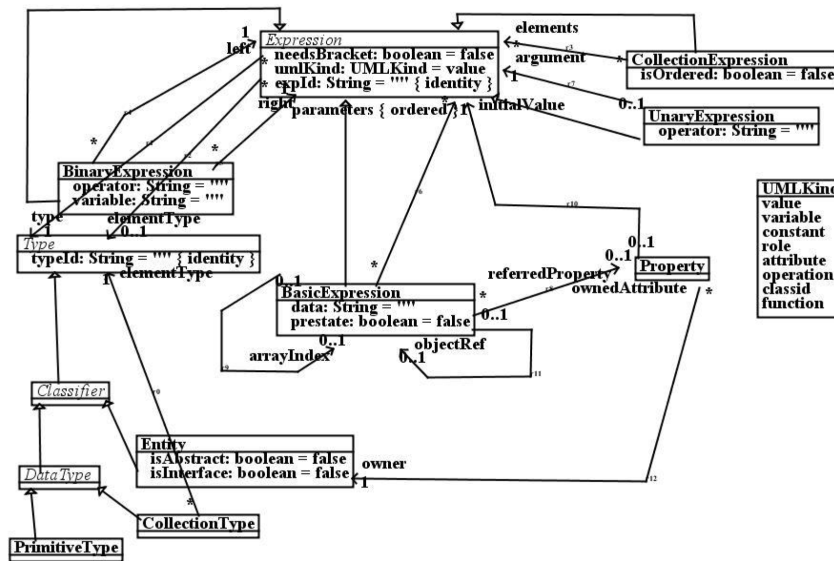

Fig. 3. UML-RSDS OCL metamodel

[1] The operation versions should have the same signatures, overloading is not supported.

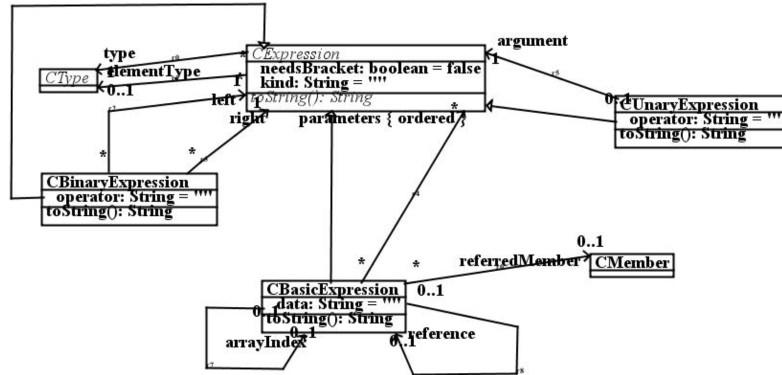

Fig. 4. C expression metamodel

The mappings are divided into four subgroups: (i) mapping of basic expressions; (ii) mapping of logical expressions; (iii) mapping of comparator, numeric and string expressions; (iv) mapping of collection expressions.

The basic expressions of OCL generally map directly to corresponding C basic expressions. Table 4 shows examples of the mapping for these.

Table 5 shows the mapping of logical expressions and operators to C.

The auxiliary operations fP are constructed to only have a single parameter, this means that mapping of forAll, select, etc is only supported where the rhs expressions depend on a single variable. The alternative (used in the UML-RSDS Java, C#, and C++ translators) is to create a specialised iterator implementation for each different use of an iterator operation.

Table 6 lists examples of comparator operators and their mappings to C.

The introduced functions *oclIncludes*, *equalsSet*, etc, are all defined in ocl.h, since they are not specific to particular element types.

Tables 7, 8 show examples of the values and operators that apply to ordered sets and sequences, and their C translations. Some UML-RSDS OCL operators (unionAll, intersectAll, symmetricDifference, subcollections) were considered a low priority, because these are infrequently used, and were not translated. Similar collection operators are provided for sets and bags in ocl.h.

A common form of OCL expression is the evaluation of a reduce operation (min, max, sum, prd) applied to the result of a collect, eg.: $s \rightarrow collect(e) \rightarrow sum()$ where e is double-valued. This is mapped to:

$$sumdouble(collectE_double(s', fe), oclSize((void **)\ s'))$$

because it is not possible to find the length of a collection of primitive values. Likewise, $s.att.sum$ is mapped to $sumdouble(getAllE_att(s'), oclSize((void **)\ s'))$. For a literal sequence s, $oclSize(s')$ can be directly determined and used.

Table 9 shows the translation of select and collect operators on sequential collections. *selectMaximals* and *selectMinimals* are not currently mapped to C.

UML-RSDS OCL expression e	*C representation* e'
self	*self* as an operation parameter
Data feature f of context E with no objectRef E data feature f of instance *ex*	 $getE_f(self)$ $getE_f(ex')$
Operation call $op(e1, ..., en)$ or $obj.op(e1, ..., en)$ of instance entity scope op of E	op_E((struct E*) self, e1', ..., en') op_E((struct E*) obj', e1', ..., en')
Call $op(e1, ..., en)$ of static/application scope op	op(e1', ..., en')
E attribute/role f of collection *exs*	$getAllE_f(exs')$ (duplicate values preserved)
$col \rightarrow at(ind)$ Sequence/ordered set *col*	(col')[ind' - 1]
$E[v]$ v single-valued $E[vs]$ vs collection-valued	getEByPK(v') getEByPKs(vs')
$E.allInstances$ concrete E	*e_instances*
boolean true, false	TRUE, FALSE

Table 4. Mapping specifications for basic expressions

<table>
<tr><th>UML-RSDS OCL expression e</th><th>C expression e'</th></tr>
<tr><td><code>A & B</code>
<code>A or B</code>
<code>not(A)</code></td><td><code>A' && B'</code>
<code>A' || B'</code>
<code>!A'</code></td></tr>
<tr><td><code>E->exists(P)</code>
<code>e->exists(P)</code></td><td><code>existsE(e_instances,fP)</code> fP evaluates P'
<code>existsE(e',fP)</code></td></tr>
<tr><td><code>E->forAll(P)</code>
<code>e->forAll(P)</code></td><td><code>forAllE(e_instances,fP)</code> fP evaluates P'
<code>forAllE(e',fP)</code></td></tr>
</table>

Table 5. Mapping specifications for logical expressions

OCL expression e	*C representation* e'
`s->includes(x)` s sequential collection	$oclIncludes((void **)\ s', (void*)\ x')$
`s->excludes(x)` s sequential collection	$oclExcludes((void **)\ s', (void*)\ x')$
$x = y$ Numerics, booleans, objects Strings Ordered sets Sequences	$x' == y'$ $strcmp(x', y') == 0$ $equalsSet((void **)\ x', (void **)\ y')$ $equalsSequence((void **)\ x', (void **)\ y')$
$x < y$ numerics Strings	$x' < y'$ $strcmp(x', y') < 0$

Table 6. Mapping specifications for comparator expressions

Expression e	*C translation e'*
`x->size()`	oclSize((void**) x')
`x->reverse()`	reverseE(x')
`x->sort()`	(struct E**) oclSort((void**) x', compareTo_E)
	x of entity element type E
	(char**) oclSort((void**) x', compareTo_String)
	x of String element type
`x->sum()`	sumString(x',n), sumint(x',n), sumlong(x',n), sumdouble(x',n)
	n is the size of x
`x->max()`	maxint(x',n), maxlong(x',n),
	maxdouble(x',n), or maxString(x',n)
	n is the size of x.

Table 7. Translation of collection unary operators

Expression e	*C translation e'*
`s->including(x)`	insertE(s',x') or appendE(s',x')
`s->excluding(x)`	removeE(s',x')
`s - t`	removeAllE(s',t')
`s->append(x)`	appendE(s',x')
`s->count(x)`	oclCount((void**) s', (void*) x')
`s->at(i)`	(struct E*) (s')[i'-1]
`s->indexOf(x)`	oclIndexOf((void**) s', (void*) x')
`s->union(t)`	unionE(s',t')
`s->intersection(t)`	intersectionE(s', t')
`s->sortedBy(e)`	(struct E**) oclSort((void**) s', comparee)
	comparee defines e-order on E objects

Table 8. Translation of binary collection operators (s of entity element type E)

OCL expression e	*C translation e'*
`s->select(P)`	selectE(s', fP)
	where E is entity element type of *s*, fP evaluates P':
	unsigned char fP(struct E* self) { return P'; }
`s->select( x | P )`	as above, fP is:
	unsigned char fP(struct E* x) { return P'; }
`s->collect(e)`	collectE_et(s', fe)
e of primitive type et	fe evaluates e'
`s->collect( x | e )`	(et'*) collectE(s', fe)
Non-primitive type et	as above

Table 9. Mapping of selection and collection expressions

Unlike the types and class diagram mappings, a recursive functional style of specification is needed for the expressions mapping (and for activities). This is because the subordinate parts of an expression are themselves expressions. For each category of expression, the mapping is decomposed into cases, for example:

```
BasicExpression::
query mapBasicExpression(ob : Set(CExpression),
              aind : Set(CExpression),
              pars : Sequence(CExpression)) : CExpression
pre:
  ob = CExpression[objectRef.expId] &
  aind = CExpression[arrayIndex.expId] &
  pars = CExpression[parameters.expId]
post:
  (umlKind = value  =>
     result = mapValueExpression(ob,aind,pars)) &
  (umlKind = variable  =>
     result = mapVariableExpression(ob,aind,pars)) &
  (umlKind = attribute  =>
     result = mapAttributeExpression(ob,aind,pars)) &
  (umlKind = role  =>
     result = mapRoleExpression(ob,aind,pars)) &
  (umlKind = operation  =>
     result = mapOperationExpression(ob,aind,pars)) &
  (umlKind = classid  =>
     result = mapClassExpression(ob,aind,pars)) &
  (umlKind = function  =>
     result = mapFunctionExpression(ob,aind,pars))
```

The operation precondition of *mapBasicExpression* asserts that the parameters correspond to the sub-parts of the basic expression. The *kind* attribute records the origin of the C expression. This enables an inverse operation to be defined. The operations can be inverted clause-by-clause. This is possible since the target expression encodes all necessary information to derive the source expression. For example, an assignment $c.parameters = Sequence\{s\} \frown pars$ inverts to $pars = c.parameters.tail \ \& \ s = c.parameters.first$.

The mapping transformation consists of 92 operations and 33 transformation rules. The expression mapping is then further used by the mappings of UML activities and use cases to C code.

The efficiency of the expression translator has been tested on a range of UML/OCL models (Table 10).

The semantic correctness of the mapping was checked by reasoning inductively on expression structure that $Sem_C(e')$ is equivalent to $Sem_{OCL}(e)$ for OCL expressions e, if $e' = CExpression[e.expId]$, where Sem_C is a mathematical semantics for C programs, and Sem_{OCL} is the UML-RSDS semantics for expressions, defined by a mapping from OCL to the B AMN formalism [6]. We assume that *malloc* and *calloc* always succeed, and that equivalent numeric types are used in the specification and implementation.

#classes	*#attributes per class*	*Execution time*
10	10	90ms
10	50	170ms
50	50	591ms
50	100	881ms
100	100	1.7s

Table 10. Execution times for OCL to C transformation

4 Evaluation

In this section we evaluate the effectiveness of the translation approach. The Visual Studio (2012) and lcc[2] (2016) C compilers were used to evaluate the generated C code. All tests were carried out on a standard Windows 7 laptop with Intel i3 2.53GHz processor using 25% of processing capacity.

In order to test the efficiency and compactness of generated code, we considered different UML specifications with different computational characteristics. The first was a small-scale application involving a fixed-point computation of the maximum-value node in a graph of nodes. This application has one entity type A, with an attribute $x : int$ and a self-association $neighbours : A \rightarrow Sequence(A)$. There is a use case *maxnode* with the postcondition

```
A::
  n : neighbours & n.x > x@pre  =>  x = n.x
```

This updates a node to have the maximum x value of its neighbours. Because this constraint reads and writes $A :: x$, a fixed-point design is generated by the UML-RSDS tools. It is an example of object-oriented specification with intensive use of navigation from object to object.

The generated C code of the use case and its auxiliary functions is:

```
void maxnode1(struct A* self, struct A* n)
{ setA_x(self, getA_x(n)); }

unsigned char maxnode1test(struct A* self, struct A* n)
{ if (getA_x(n) > getA_x(self))
  { return TRUE; }
  return FALSE;
}

unsigned char maxnode1search(void)
{ int ind_boundedloopstatement_80 = 0;
  int size_boundedloopstatement_80 = oclSize((void**) a_instances);
  for ( ; ind_boundedloopstatement_80 < size_boundedloopstatement_80;
          ind_boundedloopstatement_80++)
  { struct A* ax = (a_instances)[ind_boundedloopstatement_80];
    int ind_boundedloopstatement_85 = 0;
```

[2] www.cs.virginia.edu/~lcc-win32

```
    int size_boundedloopstatement_85 = oclSize((void**) getA_neighbours(ax));

    for ( ; ind_boundedloopstatement_85 < size_boundedloopstatement_85;
          ind_boundedloopstatement_85++)
    { struct A* n = (getA_neighbours(ax))[ind_boundedloopstatement_85];
      if (maxnode1test((struct A*) ax, n))
      { maxnode1((struct A*) ax, n);
        return TRUE;
      }
    }
  }
  return FALSE;
}

void maxnode(void)
{ unsigned char maxnode1_running = TRUE;
  while (maxnode1_running)
  {  maxnode1_running = maxnode1search(); }
}
```

Table 11 compares the code size (for the complete applications, including OCL library code) and the efficiency of the C code with the Java code produced by the UML-RSDS Java code generator. These show that code size is halved by using C, and that efficiency is improved.

	C version	*Java version*
Code size	17Kb	35Kb
Execution time		
A.size = 20	0	30ms
A.size = 50	15ms	70ms
A.size = 100	240ms	330ms
A.size = 200	1750ms	2500ms

Table 11. Generated C code versus Java code, case 1

In a second case, the efficiency test from [5] was used. This computes prime numbers in a given range using a double iteration. Table 12 compares the generated code in Java, C, C# and C++ on this case. In this purely numerical example, C is significantly more efficient than the alternative implementations for larger cases.

The main causes of inefficiency in generated C code are (i) repeated linear traversals of collections to calculate the sizes of collections; (ii) the cost of allocating and reallocating large contiguous blocks of memory for array-based collections. An alternative array collection representation could use the first element of an array to store the collection length. This also has the advantage that C and OCL indexing of collections would coincide. However it would hinder the compatibility of the generated code with standard C code. For sets and

Testing primes up to	*C version*	*Java version*	*C# version*	*C++ version*
10000	5ms	7ms	3ms	8ms
20000	9ms	15ms	8ms	16ms
50000	22ms	47ms	27ms	31ms
100000	47ms	63ms	54ms	62ms
200000	109ms	125ms	274ms	112ms
500000	143ms	374ms	472ms	405ms

Table 12. Generated C code versus Java, C#, C++ code, case 2

bags non-contiguous memory blocks can be used, and this reduces the memory allocation costs.

We also compared the C and Java implementations using the OCL benchmarks of [1]. Table 13 shows the execution time for adding n elements to a collection, using $\rightarrow$*including*.

n	*Sequence*	*OrderedSet*	*Bag*	*Set*	*Java Sequence*	*Java OrderedSet*
4000	65ms	150ms	16ms	26ms	4ms	180ms
8000	220ms	486ms	47ms	49ms	5ms	720ms
16000	660ms	895ms	99ms	101ms	10ms	2.5s
32000	2.1s	8.9s	202ms	231ms	10ms	10s

Table 13. C and Java efficiency results for $\rightarrow$*including*

Table 14 shows the execution time for testing the membership of 2000 elements in a collection of size n, using $\rightarrow$*includes*.

n	*Sequence/OrderedSet*	*Bag/Set*	*Java Sequence/OrderedSet*
1000	8ms	5ms	46ms
2000	21ms	10ms	62ms
4000	46ms	11ms	140ms
8000	67ms	12ms	312ms
16000	109ms	12ms	710ms
32000	169ms	16ms	1.4s

Table 14. C/Java efficiency results for $\rightarrow$*includes*

There are the following restrictions on the UML-RSDS input specification for UML2C: (i) no overloading of operation names within a class; (ii) quantifiers, collect, select/reject predicates can only depend on one context object; (iii) no static attributes; (iv) collection values and types can only be nested to 2 levels (collections of collections of non-collection types); (v) use cases cannot have input parameters; (vi) root classes must contain at least one property, and only

single inheritance is represented; (vii) there are no interfaces, association classes or qualified associations.

Restrictions (iii) and (v) will be removed in release 1.8 of UML-RSDS.

5 Related work

Code generation from UML to ANSI C is an unusual topic, with only one recent publication describing such a translator [3]. This code generator is described in a high-level manner, and it is not clear how OCL expressions or UML activities are mapped to C using the transformation. In contrast, we have implemented mappings for all elements of a substantial subset of UML, including a large subset of OCL. Formal specification approaches for MT are described in [10] and [2]. The constructive logic approach of [10] does not appear to have been applied to large scale transformations. The approach of [2] is focussed on the specification of architectural choices. Our approach enables large-scale transformations to be specified using OCL, with their implementations being verified as correct-by-construction.

A Java VM is the usual target for OCL execution [11]. Compared to [11] we consider a subset of OCL which (i) omits OclAny, null and invalid values, (ii) uses classical logic, (iii) uses computational numeric types. These modifications make the correspondence between a (UML-RSDS) OCL specification and a Java/C#/C++/C implementation more direct and also simplify specification verification, eg., using the B formal method or other classical logic theorem prover.

The code generator specification can be used as the basis of alternative C translators. In particular, there is interest in mapping to the high-integrity MISRA C subset [7]. For this subset, dynamic memory allocation is not permitted, so for each class, a maximum bound must be provided for the number of objects of the class.

6 Conclusions

The UML to C translator is the largest transformation which has been developed using UML-RSDS, in terms of the number of rules (of the order of 250 OCL rules/operations in 5 subtransformations). The translator provides efficient implementation of OCL using a direct translation approach which supports traceability and bidirectionality. The translator has been incorporated into the UML-RSDS tools version 1.7 at nms.kcl.ac.uk/kevin.lano/uml2web. UML-RSDS specifications are type-checked and converted to designs prior to export for code generation. The translator is itself defined using UML class diagrams and the UML-RSDS subset of OCL, demonstrating that purely declarative OCL specifications can be sufficient for large and complex applications: no activities or other procedural elements were needed in the specification. We found substantial benefits in reduced development time and improved correctness and flexibility compared to the manually-coded translators for Java, C# and C++. The UML2C

OCL code is less than 25% of the size of the Java code of the manually-coded C++ translator, and required half the development effort.

References

1. J. Cuadrado, F. Jouault, J. Molina, J. Bezivin, *Deriving OCL optimisation patterns from benchmarks*, OCL 2008.
2. A. Dieumegard, A. Toon, M. Pantel, *Model-based formal specification of a DSL library for a qualified code generator*, OCL 2012.
3. M. Funk, A. Nysen, H. Lichter, *From UML to ANSI-C: an Eclipse-based code generation framework*, RWTH, 2007.
4. B. Kernighan, D. Ritchie, *The C Programming Language*, Prentice Hall, 1988.
5. M. Kuhlmann, L. Hamann, M. Gogolla, F. Buttner, *A benchmark for OCL engine accuracy, determinateness and efficiency*, SoSyM vol. 11, 2012.
6. K. Lano, *Agile Model-based Development using UML-RSDS*, Taylor and Francis, 2016.
7. MIRA Ltd., *MISRA-C:2004 Guidelines for the use of the C language in critical systems*, 2004.
8. OMG, *OCL Version 2.4*, 2014.
9. OMG, *Semantics of a Foundational Subset for Executable UML Models (FUML)*, v1.1, 2015.
10. S. Zschaler, I. Poernomo, J. Terrell, *Towards using constructive type theory for verifable modular transformations*, FREECO' 11.
11. E. Willink, *An extensible OCL virtual machine and code generator*, OCL '12, 2012.