Transformation from UML to C

K. Lano

December 14, 2016

1 Introduction

In this report we describe the requirements and specification of a code generator for mapping UML-RSDS to ANSI C. The translator has the top-level functional requirement

> F1: *Translate UML-RSDS designs (class diagrams, OCL, activities and use cases) into ANSI C code.*

The identified stakeholders included: (i) the UML-RSDS development team; (ii) users of UML-RSDS who require C code for embedded or limited resource systems; (iii) end-users of such systems.

Direct access was only possible to stakeholders (i). Access to other stakeholders was substituted by research into the needs of such stakeholders.

An initial phase of requirements elicitation for this system used document mining (research into the ANSI C language and existing UML to C translators) and a semi-structured interview with the principal stakeholder. This produced an initial set of requirements, with priorities.

The functional requirement was decomposed into five high-priority subgoals, each of which is the responsibility of a separate subtransformation (Figure 1):

- F1.1: Translation of types
- F1.2: Translation of class diagrams
- F1.3: Translation of OCL expressions
- F1.4: Translation of activities
- F1.5: Translation of use cases.

Each translation in this list depends upon all of the preceding translations. In addition, the translation of operations of classes depends upon the translation of expressions and activities.

The development was therefore organised into five iterations, one for each translator part, and each iteration was given a maximum duration of one month. The overall development bound was 6 months (process requirement NF10).

Other high-priority requirements identified for the translator were the following functional and non-functional system (product) requirements:

- NF1: Termination, given correct input.
- F2: Syntactic correctness: given correct input, a valid C program will be produced.
- F3: Model-level semantic preservation: the semantics of the source and target models are equivalent.
- F4: Traceability: a record should be maintained of the correspondence between source and target elements.

Medium-level priority requirements were:

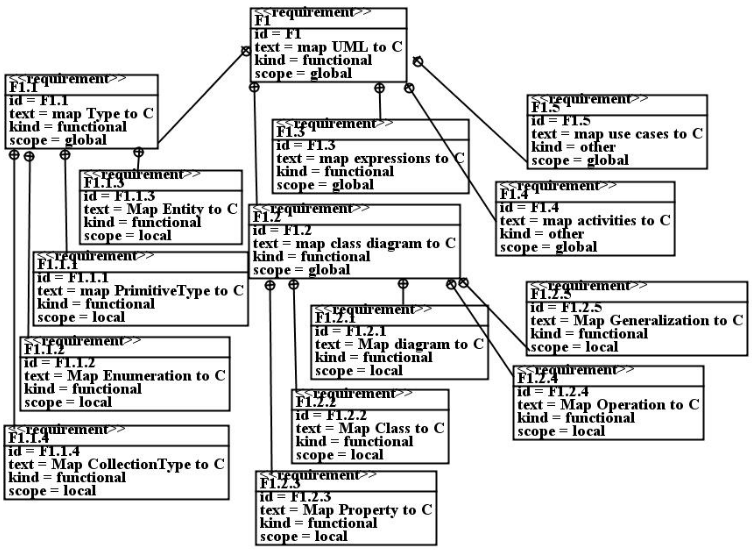

Figure 1: Functional requirements decomposition in SysML

- F5: Bidirectionality between source and target.
- NF2: Efficiency: input models with 100 classes and 100 attributes should be processed within 30 seconds.
- NF3: Modularity of the transformation.

Low-priority requirements were:

- F6: Confluence.
- NF4: Flexibility: ability to choose different C interpretations for UML elements.

The project attributes are as follows:

1. Size: medium
2. Complexity: high
3. Volatility: low
4. Customer relationship: low
5. Safety: low
6. Quality: high
7. Cost constraint: medium
8. Time constraint: medium
9. Domain knowledge: medium

It was identified that a suitable overall architecture for the transformation was a sequential decomposition of a model-to-model transformation $design2C$, and a model-to-text transformation $genCtext$. Decomposing the code generator into two sub-transformations improves its modularity, and simplifies the constraints, which would otherwise need to combine language translation and

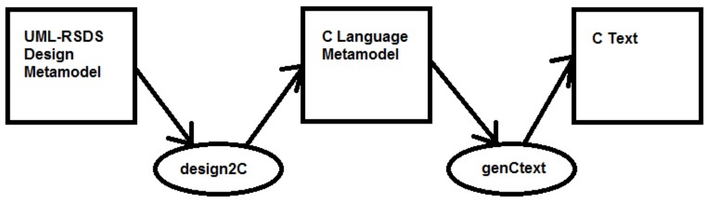

Figure 2: C code generator architecture

text production. Figure 2 shows the resulting transformation architecture. This is an example of the architectural pattern Factor Code Generation into Model-to-model and Model-to-code.

This decomposition means that each of the high-level requirements need to be satisfied by both *design2C* and *genCtext*. The requirements for bidirectionality and traceability are however specific to *design2C*.

After a further interview, the application of model-based testing and bx to achieve F3 was identified as an important area of work. Tests for the synthesised C code should, ideally, be automatically generated based on the source UML model. The bx property can be utilised for testing semantic equivalence by transforming UML to C, applying the reverse transformation, and comparing to identify if the two UML models are isomorphic.

The following agile practices were used in the development: (i) short iterations; (ii) refactoring; (iii) emphasis on simplicity; (iv) product and iteration backlogs; (v) Scrumboards; (vi) continuous integration and testing.

The following MDD practices were used: (i) metamodelling; (ii) transformations; (iii) executable modelling.

Within each iteration, the following process is generally followed:

1. Exploratory prototyping to assess feasibility of possible C programming constructs to express UML and OCL elements.

2. Informal specification in concrete grammar of the mappings from UML to C. Discussion of informal specification with stakeholders/team members.

3. Formalisation of the mappings as UML-RSDS rules and operations.

4. Specification review/refactoring.

5. Prototyping and testing of these specifications; integration with other software elements; revision of specifications as necessary to pass tests and efficiency requirements.

6. Deployment/delivery as Java jar executables.

We grouped iterations 1 and 2 in one executable (uml2Ca.jar), and iterations 3, 4 and 5 in another (uml2Cb.jar).

2 Iteration 1: Type mapping

This iteration was divided into three phases: detailed requirements analysis; specification; testing.

Detailed requirements elicitation used structured interviews to identify (i) the source language; (ii) the mapping requirements; (iii) the target language; (iv) other functional and non-functional requirements for this sub-transformation. Scenarios and test cases were prepared.

The source language was identified as the *Type* class and its subclasses in the standard UML-RSDS class diagram metamodel (Figure 3).

The initial target language is a simplified version of the abstract syntax of C programs, sufficient to represent UML types (Figure 4). This language is open to further elaboration and extension during the development.

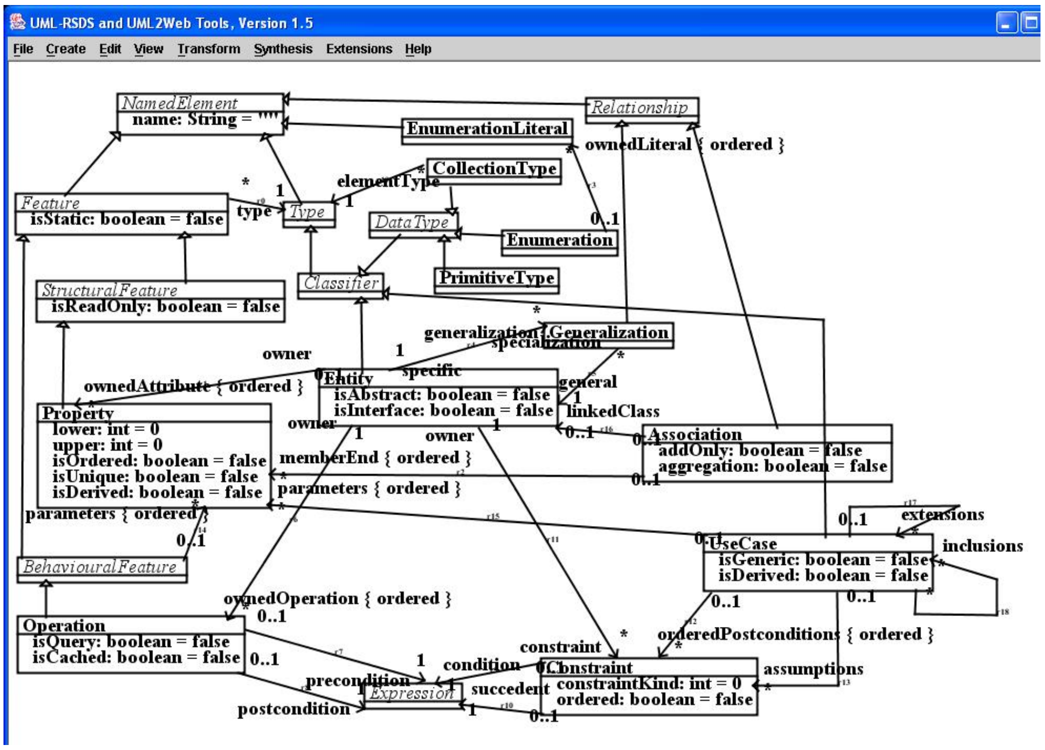

Figure 3: UML-RSDS class diagram metamodel

Using goal decomposition, the requirements were decomposed into specific mapping requirements, these are the local functional requirements F1.1.1 to F1.1.4 in Figure 1. Table 1 shows the informal scenarios for these local mapping requirements, based on the concrete metaclasses of *Type* and the different cases of instances of these metaclasses. The schematic concrete grammar is shown for the C elements representing the UML concepts. As a result of requirements evaluation and negotiation with the principal stakeholder, using exploratory prototyping and scenario analysis, it was determined that all of these local requirements are of high priority except for the mapping F1.1.5 of enumerations (medium priority). The justification for this is that enumerations are not an essential UML language element. Bidirectionality was considered a high priority for this subtransformation. It was identified that to meet this requirement, all source model Property elements must have a defined type – and specifically that elements representing many-valued association ends must have some CollectionType representing their actual type. A limitation of the proposed mapping is that mapping collections of primitive values (integers, doubles, booleans) to C is not possible, because there is no means to identify the end of the collection in C (NULL is used as the terminator for collections of objects and collections of strings). This means that expressions such as $objs \rightarrow collect(att) \rightarrow sort()$ must be coded as $objs \rightarrow sortedBy(att) \rightarrow collect(att)$. Likewise, $objs \rightarrow collect(att) \rightarrow count(val)$ should be expressed as $objs \rightarrow select(att = val) \rightarrow size()$.

Requirements specification formalises these mappings as UML-RSDS rules, defining the postconditions of a transformation $types2C$. Scenario analysis and evolutionary prototyping were used for this stage. The Auxiliary Correspondence Model pattern was used to achieve the bidirectionality requirement, and the traceability requirement. A new identity attribute $typeId : String$ was introduced into *Type*, and *ctypeId* into *CType*.

An example of a scenario expressed in the SBVRSE notation for SysML [3] is (for F1.1.1):

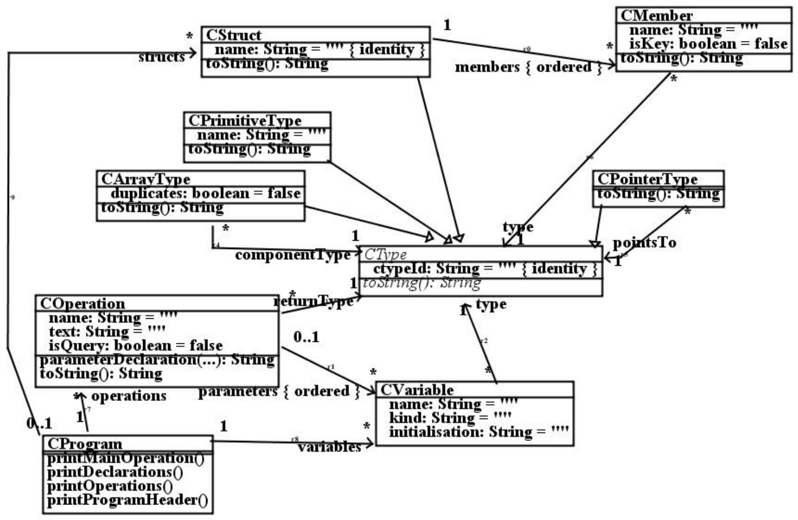

Figure 4: C language metamodel

Scenario	*UML 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	$Set(E)$ type	`struct E**` (array of E', without duplicates)
F1.1.4.2	$Sequence(E)$ type	`struct E**` (array of E', possibly with duplicates)

Table 1: Informal scenarios for types2C

> It is necessary that each string type PrimitiveType instance s maps to a CPointerType instance p such that $p.ctypeId = s.typeId$, and to a CPrimitiveType instance c such that $p.pointsTo = c$ and $c.name =$ "char".
>
> Each PrimitiveType is considered to be a string type PrimitiveType if it has name "String".

Such representations can be directly mapped to UML-RSDS rule specifications, listed below.

Primitive types, entity types and collection types are successively mapped. *typeId* : *String* and *ctypeId* : *String* are new identity attributes introduced for Auxiliary Correspondence Model. These provide a correspondence between the occurrences of types in the source UML-RSDS design and in the target C implementation. For model.txt files, *typeId* values are strings consisting of digits. It is assumed that collection types can only have entity types or primitive types as their element types:

$$\begin{array}{l} CollectionType :: \\ \qquad elementType : Entity \ or \ elementType : PrimitiveType \end{array}$$

This ensures that, in the final two rules, *CType*[*elementType.typeId*] does exist at the point where it is looked-up: it has been created by earlier rules operating on the element type.

```
PrimitiveType::
  name = "int"  =>
     CPrimitiveType->exists( p | p.ctypeId = typeId & p.name = "int" )

PrimitiveType::
  name = "long"  =>
     CPrimitiveType->exists( p | p.ctypeId = typeId & p.name = "long" )

PrimitiveType::
  name = "double"  =>
     CPrimitiveType->exists( p | p.ctypeId = typeId & p.name = "double" )

PrimitiveType::
  name = "boolean"  =>
     CPrimitiveType->exists( p | p.ctypeId = typeId & p.name = "unsigned char" )

PrimitiveType::
  name = "void"  =>
     CPrimitiveType->exists( p | p.ctypeId = typeId & p.name = "void" )

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

Entity::
  CPointerType->exists( p | p.ctypeId = typeId &
         CStruct->exists( c | c.name = name & c.ctypeId = name & p.pointsTo = c ) )

CollectionType::
   name = "Sequence"  =>
      CArrayType->exists( a | a.ctypeId = typeId & a.duplicates = true &
                       a.componentType = CType[elementType.typeId] )

CollectionType::
   name = "Set"  =>
      CArrayType->exists( c | c.ctypeId = typeId & c.duplicates = false &
                        c.componentType = CType[elementType.typeId] )
```

During requirements validation and verification, model-level semantic preservation can be shown based on the bx properties. The above constraints can be inverted to:

```
CPrimitiveType::
  name = "int"  =>
     PrimitiveType->exists( t | t.typeId = ctypeId & t.name = "int" )

CPrimitiveType::
  name = "long"  =>
     PrimitiveType->exists( t | t.typeId = ctypeId & t.name = "long" )

CPrimitiveType::
  name = "double"  =>
     PrimitiveType->exists( t | t.typeId = ctypeId & t.name = "double" )

CPrimitiveType::
  name = "unsigned char"  =>
     PrimitiveType->exists( t | t.typeId = ctypeId & t.name = "boolean" )

CPrimitiveType::
  name = "void"  =>
     PrimitiveType->exists( t | t.typeId = ctypeId & t.name = "void" )

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

CPointerType::
  c : CStruct & pointsTo = c  =>
     Entity->exists( e | e.typeId = ctypeId & e.name = c.name )

CArrayType::
   duplicates = true  =>
      CollectionType->exists( t | t.typeId = ctypeId & t.name = "Sequence" &
                           t.elementType = Type[componentType.ctypeId] )

CArrayType::
   duplicates = false  =>
      CollectionType->exists( t | t.typeId = ctypeId & t.name = "Set" &
                           t.elementType = Type[componentType.ctypeId] )
```

This establishes the bx property for *types2C*.

In addition, reasoning by cases shows that the C code resulting from a valid input UML model does conform to C syntax. The corresponding part of *genCtext* was developed alongside *types2C*. Various *toString*() : *String* operations in the different C language classes carry out the basic actions of *genCtext*.

Testing was also used for validation, in addition to inspection and formal arguments for the satisfaction of the requirements.

The iteration took 1 month, with several cycles of specification and testing needed until all functional and non-functional requirements were met. The bx requirement F5 was the main source of difficulties. In this iteration, the specification effort included construction of the C metamodels for use in subsequent iterations. The transformation size was 9 rules and 7 operations.

The estimated effort for this iteration is shown in Table 2.

Stage	*Effort (person days)*
Req. Elicitation	2
Eval./Negotiation	1
Specification	7
Review/Validation	8
Implementation/ Testing	10
Total	28

Table 2: Development effort for Iteration 1

3 Iteration 2: Class diagram mapping

This iteration also used a three-phase approach, to define a subtransformation *classdiagram2C*. The class diagram elements *Property*, *Operation*, *Entity*, *Generalization* from Figure 3 were identified as the input language. Exploratory prototyping was used for requirements elicitation and evaluation. During requirements evaluation and negotiation it was agreed that the metafeatures *isStatic*, *isReadOnly*, *isDerived*, *isCached* would not be represented in C, nor would *addOnly*, *aggregation*, *constraint* or *linkedClass*. This means that aggregations, association classes and static/constant features are not specifically represented in C. Interfaces are not represented. Only single inheritance is represented. Maintaining the mutual consistency of opposite association ends is not included, but is placed as a high priority for future work.

The C representation in Figure 4 is sufficient as a target language for this subtransformation, with the addition of *isKey* : *boolean* and *isQuery* : *boolean* attributes to *CMember* and *COperation*, respectively. A *scope* : *String* attribute is added to *COperation* to distinguish operations representing entity methods (scope = "entity") from those representing use cases (scope = "application").

The scenarios of the local mapping requirements for class diagram elements are shown in Table 3.

Associations are represented by the memberEnd[2] Property, which is a feature of the entity at end 1 of the association. In the case of a bidirectional association, the memberEnd[1] Property is also defined in the design, and is also represented in C. The maintenance of the mutual consistency of the opposite ends is *not* included in this translation scheme. Global variables are elements of the *variables* list of the *CProgram* instance representing the class diagram. All of these local mapping requirements F1.2.1 to F1.2.5 are of high priority. As with iteration 1, bx properties are of high priority for this subtransformation.

The scenarios for this transformation are more complex, and were considered during evaluation and negotiation, and during requirements specification. The scenarios included:

- F1.2.2, F1.2.3.1, F1.2.3.2: simple and primary key attributes of a single class; access and update to these, and creation of class instances. Consideration of this scenario led to simplification of the representation of primary keys. It was noted that for future improvement, a hash-based or BST-based key lookup should be used.

- F1.2.4: operations with the same name and parameters but in different classes. It was confirmed that the proposed mapping produced a valid C program. However, some C compilers do not permit such operations, so it was decided to use distinct names *opE*, *opF*, etc, if an operation *op* appears in several classes E, F.

- F1.2.5: a typical situation where client class A has a many-many association with ordered role br at an (abstract) superclass B of concrete classes C and D. Navigations of the form br[i].x for a property x of B must be possible. This was confirmed: the navigation would be $getB\_x(br[i-1])$. However, C and D elements can only be stored in br by 'upcasting' them: $br = appendB(br, cx{\rightarrow}super)$, and likewise for D. This means that C and D instances are treated as B instances when in br, and operation polymorphism cannot be used.

Scenario	*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()` operation `struct E** appendE(struct E**, struct E*)` operation `struct E** newEList()` operation
F1.2.3.1	Property $p : T$ (not principal identity attribute)	Member `T' p;` of the struct for p's owner, 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	C operation `T' op(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 $getA\_att(x{\rightarrow}super)$, etc.

Table 3: Informal scenarios for the mapping of UML class diagrams to C

The detailed form of the *createE* operation was determined: for classes without primary keys, this operation has no parameter and has the form:

```
struct E* createE()
{ struct E* result = (struct E*) malloc(sizeof(struct E));
  // initialisations of E members
  e_instances = appendE(e_instances, result);
  e_size++;
  return result;
}
```

The initialisations are the standard defaults: 0 for numerics, the empty string for char*, NULL for roles, FALSE for booleans. If a super member is present, this is initialised by calling createF() for its class F:

```
CMember::
query initialiser() : String
post:
  (isKey = true  =>  result = "") &
  (name = "super"  =>
      result = "  result->super = create" + type.pointsTo.name + "();\n") &
  (type : CPointerType or type : CArrayType  =>
      result = "  result->" + name + " = NULL;\n") &
  (type : CPrimitiveType  =>
      result = "  result->" + name + " = 0;\n")
```

If a primary key attribute *pk* exists, then its value is provided as a parameter:

```
struct E* createE(char* v)
{ struct E* result = NULL;
```

```
  result = getEByPK(v);
  if (result != NULL) { return result; }
  result = (struct E*) malloc(sizeof(struct E));
  setE_pk(result, v);
  // initialisations of E members
  e_instances = appendE(e_instances, result);
  e_size++;
  return result;
}
```

getEByPK(*str*) does a linear search of *e_instances* for an instance *struct E* $*$ *e* with $e \rightarrow pk$ equal to *str*:

```
struct E* getEByPK(char* ex)
{ int n = length((void**) e_instances);
  int i = 0;
  for ( ; i < n; i++)
  { char* attv = e_instances[i]->key;
    if (attv != NULL && strcmp(attv,ex) == 0)
    { return e_instances[i]; }
  }
  return NULL;
}

struct E** getEByPKs(char* col[])
{ int n = length((void**) col);
  struct E** result = (struct E**) calloc(n+1, sizeof(struct E*));
  int i = 0;
  int j = 0;
  for ( ; i < n; i++)
  { char* attv = col[i];
    struct E* ex = getEByPK(attv);
    if (ex != NULL)
    { result[j] = ex; j++; }
  }
  result[j] = NULL;
  return result;
}
```

The mappings were formalised as UML-RSDS rules. The corresponding part of *genCtext* was also written, enabling complete C programs to be produced: operations *printProgramHeader*(), *printDeclarations*(), *printOperations*() and *printMainOperation*() of *CProgram* display the text of these C program parts.

For F1.2.2, the definition of structs for entities is carried out by types2C. The global variables are created by the following constraint of *printDeclarations*():

```
CStruct::
  ("struct " + name + "** " + name.toLowerCase() + "_instances = NULL;")->display() &
  ("int " + name.toLowerCase() + "_size = 0;")->display()
```

Getters and setters are created by *getterOp*, *setterOp*, *getAllOp*:

```
CMember::
query getterOp(ent : String) : String
post:
 result =
    type + " get" + ent + "_" + name +
    "(struct " + ent + "* self) { return self->" + name + "; }\n"

CMember::
query inheritedGetterOp(ent : String, sup : String) : String
post:
 (name /= "super"  =>
    result =
      type + " get" + ent + "_" + name +
      "(struct " + ent + "* self) { return get" +
         sup + "_" + name + "(self->super); }\n") &
  (name = "super"  =>
      result = self.ancestorGetterOps(ent,sup) )

CMember::
query inheritedGetterOps(ent : String) : String
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 sup = type.pointsTo &
 result = sup.members->collect( m | m.inheritedGetterOp(ent, sup.name) )->sum()

CMember::
query ancestorGetterOps(ent : String, sup : String) : String
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 anc = type.pointsTo &
 result = anc.members->collect( m | m.inheritedGetterOp(ent, sup) )->sum()


CMember::
query setterOp(ent : String) : String
post:
  result =
    "void set" + ent + "_" + name +
    "(struct " + ent + "* self, " + type + " _value) { self->" + name + " = _value; }\n"

CMember::
query inheritedSetterOp(ent : String, sup : String) : String
post:
 (name /= "super"  =>
    result =
      "void set" + ent + "_" + name +
      "(struct " + ent + "* self, " + type + "_value) { set" +
           sup + "_" + name + "(self->super, _value); }\n") &
  (name = "super"  =>
      result = self.ancestorSetterOps(ent,sup) )

CMember::
query inheritedSetterOps(ent : String) : String
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 sup = type.pointsTo &
 result = sup.members->collect( m | m.inheritedSetterOp(ent, sup.name) )->sum()

CMember::
query ancestorSetterOps(ent : String, sup : String) : String
```

```
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 anc = type.pointsTo &
 result = anc.members->collect( m | m.inheritedSetterOp(ent, sup) )->sum()


CMember::
query getAllOp(ent : String) : String
pre: type : CPrimitiveType
post:
  result = type + "* getAll" + ent + "_" + name + "(struct " + ent + "* col[])\n" +
    "{ int n = length((void**) col);\n" +
    "  " + type + "* result = (" + type + "*) calloc(n, sizeof(" + type + "));\n" +
    "  int i = 0;\n" +
    "  for ( ; i < n; i++)\n" +
    "  { result[i] = get" + ent + "_" + name + "(col[i]); }\n" +
    "  return result;\n" +
    "}\n"

CMember::
query getAllOp1(ent : String) : String
pre: type : CPointerType
post:
  result = type + "* getAll" + ent + "_" + name + "(struct " + ent + "* col[])\n" +
    "{ int n = length((void**) col);\n" +
    "  " + type + "* result = (" + type + "*) calloc(n+1, sizeof(" + type + "));\n" +
    "  int i = 0;\n" +
    "  for ( ; i < n; i++)\n" +
    "  { result[i] = get" + ent + "_" + name + "(col[i]); }\n" +
    "  result[n] = NULL;\n" +
    "  return result;\n" +
    "}\n"


CMember::
query inheritedAllOp(ent : String, sup : String) : String
post:
  (name /= "super" & type : CPrimitiveType =>
    result = getAllOp(ent)) &
  (name /= "super" & type : CPointerType  =>
    result = getAllOp1(ent)) &
  (name = "super"  =>
      result = self.ancestorAllOps(ent,sup) )

CMember::
query ancestorAllOps(ent : String, sup : String) : String
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 anc = type.pointsTo &
 result = anc.members->collect( m | m.inheritedAllOp(ent, sup) )->sum()

CMember::
query inheritedAllOps(ent : String) : String
pre: type : CPointerType &
   type.pointsTo : CStruct
post:
 sup = type.pointsTo &
 result = sup.members->collect( m | m.inheritedAllOp(ent, sup.name) )->sum()


CMember::
query getPKOp(ent : String) : String
post:
  e = ent.toLowerCase &
  result =
    "struct " + ent + "* get" + ent + "ByPK(char* ex)\n" +
    "{ int n = length((void**) " + e + "_instances);\n" +
    "  int i = 0;\n" +
    "  for ( ; i < n; i++)\n" +
    "  { char* attv = get" + ent + "_" + name + "(" + e + "_instances[i]);\n" +
    "    if (attv != NULL && strcmp(attv,ex) == 0)\n" +
    "    { return " + e + "_instances[i]; }\n" +
    "  }\n" +
    "  return NULL;\n" +
    "}\n"

CStruct::
query getPKsOp() : String
post:
  result = "struct " + name + "** get" + name + "ByPKs(char* col[])\n" +
    "{ int n = length((void**) col);\n" +
    "  struct " + name + "** result = (struct " + name +
           "**) calloc(n+1, sizeof(struct " + name + "*));\n" +
    "  int i = 0; \n" +
    "  int j = 0; \n" +
    "  for ( ; i < n; i++)\n" +
    "  { char* attv = col[i];\n" +
    "    struct " + name + "* ex = get" + name + "ByPK(attv);\n" +
    "    if (ex != NULL) \n" +
    "    { result[j] = ex; j++; }\n" +
    "  }\n" +
    "  result[j] = NULL; \n" +
    "  return result;\n" +
    "}\n"
```

These are then used in the *printcode* use case:

```
CStruct::
f : members & f.name /= "super"  =>  f.getterOp(name)->display()

CStruct::
f : members & f.name = "super"  =>  f.inheritedGetterOps(name)->display()

CStruct::
members->exists( k | k.isKey ) & key = members->select( isKey )->any()  =>
                key.getPKOp(name)->display() & self.getPKsOp()->display()

CStruct::
f : members & f.name /= "super"  =>  f.setterOp(name)->display()

CStruct::
f : members & f.name = "super"  =>  f.inheritedSetterOps(name)->display()


CStruct::
```

```
f : members & f.type : CPrimitiveType => f.getAllOp(name)->display()

CStruct::
f : members & f.name /= "super" & f.type : CPointerType => f.getAllOp1(name)->display()

CStruct::
f : members & f.name = "super" & f.type : CPointerType => f.inheritedAllOps(name)->display()

CStruct::
members->exists( k | k.isKey ) & key = members->select( isKey )->any()  =>
               self.createPKOp(name, key.name)->display()

CStruct::
true  =>
  self.createOp(name)->display()
```

This use case also generates the createE and newEList operations, and other operations specific to E, such as collectE, selectE, rejectE, intersectionE, unionE, reverseE, frontE, tailE, asSetE, concatenateE, removeE, removeAllE, subrangeE, isUniqueE, insertAtE, etc:

```
CStruct::
query createOp(ent : String) : String
post:
  einst = ent.toLowerCase + "_instances" &
 result = "struct " + ent + "* create" + ent + "()\n" +
   "{ struct " + ent + "* result = (struct " + ent +
       "*) malloc(sizeof(struct " + ent + "));\n" +
    members->collect( m | m.initialiser() )->sum() +
   "  " + einst + " = append" + ent + "(" + einst + ", result);\n" +
  "  " + ent.toLowerCase + "_size++;\n" +
  "  return result;\n" +
  "}\n"

CStruct::
query createPKOp(ent : String, key : String) : String
post:
  einst = ent.toLowerCase + "_instances" &
 result = "struct " + ent + "* create" + ent + "(char* _value)\n" +
   "{ struct " + ent + "* result = NULL;\n" +
   "  result = get" + ent + "ByPK(_value);\n" +
    "  if (result != NULL) { return result; }\n" +
    "  result = (struct " + ent + "*) malloc(sizeof(struct " + ent + "));\n" +
    members->collect( m | m.initialiser() )->sum() +
    "  set" + ent + "_" + key + "(result, _value);\n" +
   "  " + einst + " = append" + ent + "(" + einst + ", result);\n" +
  "  " + ent.toLowerCase + "_size++;\n" +
  "  return result;\n" +
  "}\n"
```

Selective generation of OCL operators will be introduced in a further refinement, so that operations opE are only generated if there is an occurrence of $\rightarrow op$ applied to a collection of E elements in the source model.

The attributes (and association ends) owned by a class are mapped to members of its corresponding struct (F1.2.3, F1.2.4):

```
Entity::
  p : ownedAttribute & p.name.size > 0  =>
```

```
    CStruct->exists( c | c.name = name &
      CMember->exists( m | m.name = p.name & m.isKey = p.isUnique &
             m.type = CType[p.type.typeId] & m : c.members ) )
```

This could alternatively be written as:

```
Entity::
    CStruct->exists( c | c.name = name &
      ownedAttribute->forAll( p | p.name.size > 0  =>
           CMember->exists( m | m.name = p.name & m.isKey = p.isUnique &
             m.type = CType[p.type.typeId] & m : c.members ) ) )
```

F1.2.4 is specified by:

```
Operation::
    COperation->exists( op | op.name = name & op.opId = name + "_" + owner.name &
        CVariable->exists( p | p.name = "self" &
              p : op.parameters & p.kind = "parameter" & p.type = CType[owner.typeId] ) &
              parameters->forAll( x | CVariable->exists( y | y.name = x.name &
                    y.kind = "parameter" &
                    y.type = CType[x.type.typeId] &
                    y : op.parameters ) ) &
              op.isQuery = isQuery &
              op.scope = "entity" &
              op.returnType = CType[type.typeId] )
```

Update operations are given a *void* result type in the UML model data file model.txt (if they do not have a specific result type). To support bx properties, a new attribute *isQuery* needs to be introduced to the C meta model class *COperation* and set as above. To support lookup of COperations in the mapping of activities to C, a new identity attribute *opId* is introduced.

If an inheritance exists from entity E to entity F, then an additional member of type `struct F*` is inserted into the struct for E (F1.2.5):

```
Generalization::
    CMember->exists( m | m.name = "super" &
        CStruct->exists( sub | sub.ctypeId = specific.name &
                    m : sub.members & m.type = CPointerType[general.typeId] ) )
```

Care has been taken to ensure that the constraints for members, operations and generalisations are invertible. They have the following inverses:

```
CStruct::
  m : CMember & m : members  =>
    Entity->exists( e | e.name = ctypeId &
      Property->exists( p |
        p.name = m.name & p.isUnique = m.isKey &
        p.type = Type[m.type.ctypeId] ) )
```

and:

```
COperation::
  scope = "entity"  =>
    Operation->exists( op | op.name = name &
        parameters->forAll( x | x.name /= "self"  =>
             Property->exists( y | y.name = x.name &
                    y.type = Type[x.type.ctypeId] &
                    y : op.parameters ) ) &
              op.isQuery = isQuery &
              op.type = Type[returnType.ctypeId] )
```

and:

```
CMember::
  sub : CStruct & name = "super" & self : sub.members  =>
    Generalization->exists( g | g.specific.name = sub.ctypeId &
                  g.general = Entity[type.ctypeId] )
```

With the alternate form of property to member constraint, the inverse mapping for properties is:

```
CStruct::
    Entity->exists( e | e.name = name &
      members->forAll( m |
           Property->exists( p | p.name = m.name & p.isUnique = m.isKey &
             p.type = Type[m.type.ctypeId] & p : e.ownedAttribute ) ) )
```

In total there are 14 rules and 33 operations.

Testing, inspection and formal arguments were used for validation and verification. The estimated effort for this iteration is shown in Table 4.

Stage	*Effort (person days)*
Req. Elicitation	2
Eval./Negotiation	1
Specification	12
Review/Validation	12
Implementation/ Testing	10
Total	37

Table 4: Development effort for Iteration 2

The result of iterations 1 and 2 is a transformation that operates on the 3 UML-RSDS metamodels (Figure 3, Figure 5 and Figure 7) as inputs, and on the C general metamodel (Figure **??**) as output. Statement and OCL data is implicitly copied from the source to the target model by the model loading/model saving mechanisms, an example of the Implicit Copy pattern.

The completed prototype after iterations 1 and 2 has been implemented as a jar file *uml2Ca.jar* at www.dcs.kcl.ac.uk/staff/kcl/uml2Ca/. This reads an input file model.txt, produced by the Save As Model option of UML-RSDS. The C code is written to standard output:

```
java -jar uml2Ca/uml2Ca.jar
```

The specification metamodel for uml2Ca is in the file mmUML2Ca, this is 35KB in size. The generated code is over 500KB in size.

4 Iteration 3: Expression mapping

In this iteration, the detailed requirements for mapping OCL expressions to C are identified, then this subtransformation, *expressions2C*, is specified and tested. Due to the large size of this transformation, it was not possible to complete it within 1 month.

Figure 5 shows the OCL metamodel, which is the source language for the subtransformation. Figure 6 shows the corresponding C expression language abstract syntax. The OCL and C expression languages are quite similar, however C lacks many operations on collections, and these need to be provided as new library functions. A C library file `ocl.h` is defined to contain general-purpose C functions for OCL. An important semantic issue is that the NULL pointer must be treated as equal to an empty collection in the generated code. New identity attributes *expId* and *cexpId* are added to *Expression* and *CExpression*, respectively, to support the bx and traceability requirements. An additional attribute *variable* : *String* is included in the *BinaryExpression* class to represent iterator expression variables x for the cases of $s \rightarrow forAll(x \mid P)$, etc. It has the value "self" for iterator expressions without an explicit variable. A $* - - - 1$ association *context* from *Expression* to *Entity* is needed to record the context of use of the expression. Downcast expressions (where an object is used as if it is an instance of a subclass of the context) are not supported by this translator.

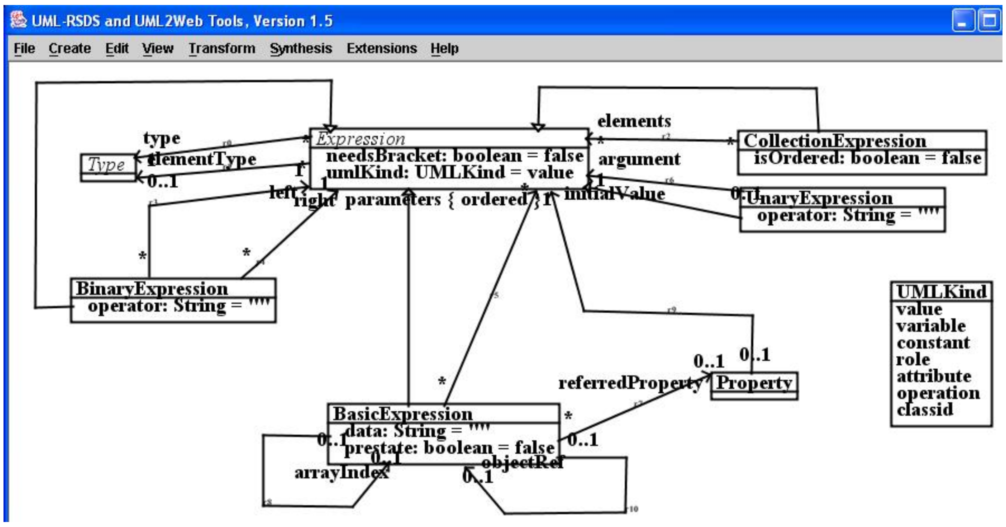

Figure 5: UML-RSDS OCL metamodel

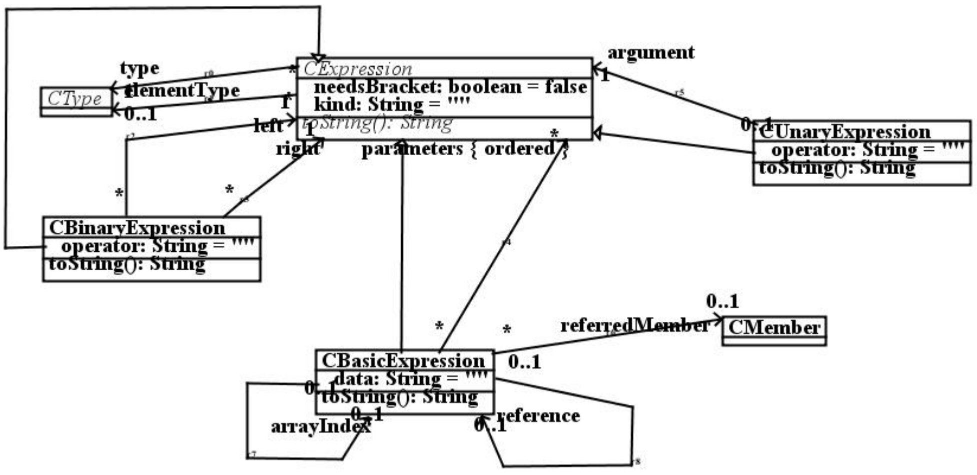

Figure 6: C expression metamodel

There are many cases to consider in the mapping requirements, so we divided these into four subgroups: (i) mapping of basic expressions; (ii) mapping of logical expressions; (iii) mapping of comparitor, numeric and string expressions; (iv) mapping of collection expressions. These were considered the natural groupings of operations and operators, and these follow in part the metaclass organisation of UML expressions in Figure 5.

4.1 Basic expressions

The basic expressions of OCL generally map directly to corresponding C basic expressions. Table 5 shows the mapping for these. These mapping requirements are grouped together as requirement F1.3.1.

OCL expression e	*C representation e'*
self	*self* as an operation parameter
Variable v or $v[ind]$	v $v[ind' - 1]$
Data feature f of context E with no objectRef E data feature f of instance *ex*	*self*→*f* (E is root) *getE_f*(*self*) (otherwise) ex'→*f* (E is root) *getE_f*(ex') (otherwise)

Operation call $op(e1, ..., en)$
or $obj.op(e1, ..., en)$
	op(self, e1', ..., en')
	op(obj', e1', ..., en')

E attribute f of collection *exs*	*getAllE_f*(exs') (duplicate values preserved)
Single-valued role $r : F$ of E collection *exs*	*getAllE_r*(exs') defined by (*struct F* $**$) *collectE*(exs', *getE_r*)
$col[ind]$ ordered collection *col*	(col')[ind' - 1]

$E[v]$
v single-valued
$E[vs]$
vs collection-valued
	getEByPK(v')
	
	getEByPKs(vs')

E.allInstances	*e_instances*
value of enumerated type, numeric or string value	*value*
boolean true, false	TRUE, FALSE

Table 5: Mapping scenarios for basic expressions

x.oclAsType(T) needs to be considered separately. For T as long, int or double, a C cast can be used. For String, a cast to *char*$*$ can be used. For an entity type, navigation using the *super* member will be necessary. The decision was made to omit this operator from the first version of the deliverable.

4.2 Logical expressions

Table 6 shows the mapping of logical expressions and operators to C. These mappings are grouped together as requirement F1.3.2.

The generation of the fP functions is carried out by:

```
CProgram::
defineCOp(b : CExpression, par : String, pt : CType) : COperation
post:
  COperation->exists( op | op.name = "op_" + operations@pre.size &
        op.returnType = b.type & op.isQuery = true & op.scope = "auxiliary" &
        CReturnStatement->exists( rs | rs.cstatId = b.cexpId + "_return" & rs.returnValue = b & c
        CVariable->exists( v | v.name = par & v.type = pt &
              v : op.parameters ) &
        op : operations & result = op )

CExpression::
```

OCL expression e	*C expression e′*
`A => B`	`!A' || B'`
`A & B`	`A' && B'`
`A or B`	`A' || B'`
`not(A)`	`!A'`
`E->exists(P)`	`existsE(e_instances,fP)` fP evaluates P
`e->exists(P)`	`existsE(e',fP)`
`E->exists1(P)`	`exists1E(e_instances,fP)` fP evaluates P
`e->exists1(P)`	`exists1E(e',fP)`
`E->forAll(P)`	`forAllE(e_instances,fP)` fP evaluates P
`e->forAll(P)`	`forAllE(e',fP)`

Table 6: Mapping scenarios for logical expressions

```
static defineCOpRef(op : COperation) : CExpression
post:
  CBasicExpression->exists( be | be.cexpId = op.name + "_ref" &
           be.data = op.name &
           be.type = op.returnType &
           be.elementType = op.elementType & result = be )

CExpression::
static defineCOpRefCast(op : COperation, cst : String) : CExpression
post:
  CBasicExpression->exists( be | be.cexpId = op.name + "_ref" &
           be.data = op.name &
           be.type = op.returnType &
           be.elementType = op.elementType & result = Expression.cast(cst, be) )

CExpression::
static defineCOpReference(op : String, typ : String) : CExpression
post:
  CBasicExpression->exists( be | be.cexpId = op.name + "_ref" &
           be.data = op.name &
           CPrimitiveType->exists( t | t.name = typ & t.ctypeId = op.name + "_" + typ &
               be.type = t & be.elementType = t & result = be ) )
```

The auxiliary operations created for iterator expressions are not mapped back to UML via the inverse transformation.

4.3 Comparitor, numeric and string expressions

Table 7 lists the comparitor operators and their mappings to C. These mappings are grouped as requirement 1.3.3.

The introduced functions *isIn*, *equalsSet*, etc, are all defined in ocl.h, since they are not specific to particular element types. The cast operator $(T)\ e$ is considered as a unary operator in C, with argument e and type T.

Numeric operators for integers and real numbers are shown in Table 8. The types `int`, `double` and `long` are not guaranteed to have particular sizes in C. All operators take double values as arguments except mod and Integer.subrange, which have int parameters. Three OCL operators: ceil, round, floor, take a double value and return an int, in contrast to the corresponding C functions. The interpretations use standard C mathematical functions from `<math.h>`, except for round and Integer.subrange. The mappings are grouped as requirement F1.3.4.

OCL expression e	*C representation e′*
$x : E$ E entity type $x : s$ s collection	$isIn((void*)\ x', (void**)\ e\_instances)$ $isIn((void*)\ x', (void**)\ s')$
`s->includes(x)` s collection	Same as $x : s$
$x\ / : E$ E entity type $x\ / : s$ s collection	$!isIn((void*)\ x', (void**)\ e\_instances)$ $!isIn((void*)\ x', (void**)\ s')$
`s->excludes(x)` s collection	Same as $x\ / : s$
$x = y$ Numerics, booleans Strings objects Sets Sequences	$x' == y'$ $strcmp(x', y') == 0$ $x' == y'$ $equalsSet((void**)\ x', (void**)\ y')$ $equalsSequence((void**)\ x', (void**)\ y')$
$x < y$ numerics Strings	$x' < y'$ $strcmp(x', y') < 0$
Similarly for >, <=, >=, / =	>, <=, >=, ! =
$s <: t$ s, t collections	$containsAll((void**)\ t', (void**)\ s')$
$s / <: t$ s, t collections	$!containsAll((void**)\ t', (void**)\ s')$
`t->includesAll(s)`	Same as $s <: t$
`t->excludesAll(s)`	*disjoint((void**) t', (void**) s')*

Table 7: Mapping scenarios for comparitor expressions

OCL expression e	*Representation in C*
-x	-x'
x + y numbers	x' + y'
x - y	x' - y'
x * y	x' * y'
x / y	x' / y'
x mod y	x' % y'
x.sqr	(x' * x')
x.sqrt	sqrt(x')
x.floor	oclFloor(x') defined as: ((int) floor(x'))
x.round	oclRound(x')
x.ceil	oclCeil(x') defined as: ((int) ceil(x'))
x.abs	fabs(x')
x.exp	exp(x')
x.log	log(x')
x.pow(y)	pow(x',y')
x.sin, x.cos, x.tan	sin(x'), cos(x'), tan(x')
Integer.subrange(st,en)	intSubrange(st',en')

Table 8: Mapping scenarios for numeric expressions

Other math operators directly available in C are: log10, tanh, cosh, sinh, asin, acos, atan. These are double-valued functions of double-valued arguments. cbrt is missing and needs to be implemented as pow(x', 1.0/3). The math operators may also be written as unary expressions in the style $e \rightarrow op()$. The translation to C is the same for this format.

String operators are shown in Table 9. Strings are \0-terminated sequences of characters in C. New functions subString, firstString, lastString, tailString, frontString, toLowerCase, toUpperCase, insertAtString, reverseString, subtractString, countString, startsWith, endsWith on strings need to be introduced. They are defined in ocl.h. The mapping requirements for string expressions are grouped as requirement F1.3.5.

Expression e	*C translation e′*
`x + y`	concatenateStrings(x', y')
`x->size()`	strlen(x')
`x->first()`	firstString(x') defined as subString(x',1,1)
`x->front()`	frontString(x') defined as subString(x', 1, strlen(x')-1)
`x->last()`	lastString(x') defined as subString(x', strlen(x'), strlen(x'))
`x->tail()`	tailString(x') defined as subString(x', 2, strlen(x'))
`x.subrange(i,j)`	subString(x', i', j')
`x->toLowerCase()`	toLowerCase(x')
`x->toUpperCase()`	toUpperCase(x')
`s->indexOf(x)`	indexOfString(s',x')
`s->hasPrefix(x)`	startsWith(s',x')
`s->hasSuffix(x)`	endsWith(s',x')
`s->characters()`	characters(s')
`s.insertAt(i,s1)`	insertAtString(s',i',s1')
`s->count(s1)`	countString(s1', s')
single character s1	
`s->reverse()`	reverseString(s')
`e->display()`	displayString(e') defined as printf("%s",e')
	for String-valued e,
	displayNumeric(e') defined as printf("%d",e')
	for numeric e
`s1 - s2`	subtractString(s1', s2')
`e->isInteger()`	–
`e->isReal()`	–
`e->toInteger()`	`atoi(e')`
`e->toReal()`	`atof(e')`

Table 9: Mapping scenarios for string expressions

4.4 Collection expressions

Table 10 shows the values and operators that apply to sets and sequences, and their C translations. Some operators (unionAll, intersectAll, symmetricDifference, subcollections) were considered a low priority, because these are infrequently used, and were not translated. The requirements are grouped as F1.3.6.

For all operations such as appendE(col,x) involving a collection and an object, x will need to be upcast to the element type of col, if it belongs to a subclass of this element type. A collection x of strings can be sorted by supplying *strcmp* as the comparison function: qsort((void**) x', length((void**) x'), sizeof(char*), (int (*)(const void*, const void*)) strcmp). Otherwise, a specifier must include a suitable $compareToE(other : E) : int$ operation in E. For *sortedBy*, if e is of a numeric or boolean type, $compare(struct\ E * \ self,\ struct\ E * \ other)$ returns e evaluated for *other* - e evaluated for *self*. For String-valued e, *strcmp* is used, and for objects, the appropriate

Expression e	*C translation e'*
$Set\{\}$	newEList()
$Sequence\{\}$	newEList()
$Set\{x1, x2, ..., xn\}$	insertE(... insertE(newEList(), x1'), ..., xn')
$Sequence\{x1, x2, ..., xn\}$	appendE(... appendE(newEList(), x1'), ..., xn')
`s->size()`	length((void**) s')
`s->including(x)`	insertE(s',x') or appendE(s',x')
`s->excluding(x)`	removeE(s',x')
`s - t`	removeAllE(s',t')
`s->prepend(x)`	–
`s->append(x)`	appendE(s',x')
`s->count(x)`	count((void*) x', (void**) s')
`s->indexOf(x)`	indexOf((void*) x', (void**) s')
$x\backslash/y$	unionE(x',y')
$x/\backslash y$	intersectionE(x',y')
$x \frown y$	concatenateE(x',y)
`x->union(y)`	unionE(x',y')
`x->intersection(y)`	intersectionE(x', y')
`x->unionAll(e)`	–
`x->intersectAll(e)`	–
`x->symmetricDifference(y)`	–
`x->any()`	x'[0]
`x->subcollections()`	–
`x->reverse()`	reverseE(x')
`x->front()`	frontE(x') defined as subrangeE(x',1,length((void**) x')-1)
`x->tail()`	tailE(x') defined as subrangeE(x',2,length((void**) x'))
`x->first()`	firstE(x') defined as x'[0]
`x->last()`	lastE(x') defined as x'[length((void**) x')-1]
`x->sort()`	(struct E**) treesort((void**) x', compareToE)
	x of entity element type E
	(char**) treesort((void**) x', strcmp)
	x of String element type
`x->sortedBy(e)`	(struct E**) treesort((void**) x', comparee)
	comparee defines e-order
`x->sum()`	sumString(x',n), sumint(x',n), sumlong(x',n), sumdouble(x',n)
	n is length of x
`x->prd()`	prdint(x',n), prdlong(x',n), prddouble(x',n)
	n is length of x
`Integer.Sum(a,b,x,e)`	sumInt(a',b',fe), sumDouble(a',b',fe) fe computes e'(x')
`Integer.Prd(a,b,x,e)`	prdInt(a',b',fe), prdDouble(a',b',fe)
`x->max()`	maxint(x',n), maxlong(x',n),
	maxdouble(x',n), maxString(x',n)
	n is length of x.
`x->min()`	minint(x',n), minlong(x',n),
	mindouble(x',n), minString(x',n)
	n is length of x.
`x->asSet()`	asSetE(x')
`x->asSequence()`	x'
`s->isUnique(e)`	isUniqueE(s',fe)
`x->isDeleted()`	killE(x')

Table 10: Scenarios for the translation of collection operators

compareToF operation.

A common form of OCL expression is 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:

$sumdouble((double*)\ collectE(s', fe), length((void **)\ s'))$

because it is not possible to find the length of a collection of primitive values. Likewise, $s.att.sum$ is mapped to $sumdouble(getAllE\_att(s'), length(void **)\ s'))$.

After evaluation and negotiation, it was decided that full implementation of *delete* should be deferred, because of the complex semantics of data deletion in C. *Integer.Sum* and *Integer.Prd* were also deferred. In addition, prototyping on different platforms revealed that compiler differences made the use of qsort impractical, and instead a custom sorting algorithm, *treesort*, was implemented. This has signature

```
void** treesort(void* col[], int (*comp)(void*, void*))
```

and the translation of $x \rightarrow sort()$ is then: (rt) treesort((void**) x', comp) for the appropriate result type *rt* and comparitor function *comp*.

For max, min, sort, the entity type E must have a compareTo(other : E) : int operation defined, this will become a function int compareToE(struct E* self, struct E* other) in C.

Table 11 shows the translation of select and collect operators. These mappings are grouped as requirement F1.3.7.

UML expression e	*C translation e'*
`s->select(P)`	selectE(s', fP)
	where E is entity element type of *s*, fP evaluates P:
	int fP(struct E* self) { return P'; }
`s->select( x | P )`	as above, fP is:
	int fP(struct E* x) { return P'; }
`s->reject(P)`	rejectE(s', fP), fP as for select
`s->reject( x | P )`	as above
`s->collect(e)`	(et'*) collectE(s', fe)
e of type et	fe evaluates e'
`s->collect( x | e )`	as above
`s->selectMaximals(e)`	–
`s->selectMinimals(e)`	–

Table 11: Scenarios for the mapping of selection and collection expressions

Unlike the types and class diagram mappings, a recursive descent style of specification is needed for the expressions mapping (and for activities). This is because the subordinate parts of an expression are themselves expressions. Thus it is not possible in general to map all the subordinate parts of an expression by prior rules: even for basic expressions, the parameters may be general expressions. In contrast, the element types of collection types cannot themselves be collection types or involve subparts that are collection types, so it is possible to map all element types before considering collection types. A recursive descent style of mapping specification uses operations of each source entity type to map instances of that type, invoking mapping operations recursively to map subparts of the instances.

An operation

```
mapExpression() : CExpression
```

is defined in each Expression subclass.

For each category of expression, the subparts of the expression are mapped to C first, and then composed by a separate operation. For example:

```
BinaryExpression::
mapExpression() : CExpression
post:
  result = mapBinaryExpression(
                     left.mapExpression(),
                     right.mapExpression())

UnaryExpression::
mapExpression() : CExpression
post:
  result = mapUnaryExpression(
                     argument.mapExpression())

BasicExpression::
mapExpression() : CExpression
post:
  result = mapBasicExpression(
            objectRef.mapExpression(),
            arrayIndex.mapExpression(),
            parameters.mapExpression())

CollectionExpression::
mapExpression() : CExpression
post:
  result = mapCollectionExpression(expId,
                   elements.mapExpression())
```

For each category of expression, the mapping is further decomposed into cases:

```
BinaryExpression::
mapBinaryExpression(lexp : CExpression,
            rexp : CExpression) : CBinaryExpression
pre:
  lexp = CExpression[left.expId] &
  rexp = CExpression[right.expId]
post:
     CBinaryExpression->exists( c | c.cexpId = expId &
         c.operator = Expression.cop(operator) &
         c.left = lexp & c.right = rexp &
         c.type = CType[type.typeId] &
         c.elementType = CType[elementType.typeId] &
         result = c )

UnaryExpression::
mapUnaryExpression(arg : CExpression) : CExpression
pre:
  arg = CExpression[argument.expId]
post:
     CUnaryExpression->exists( c | c.cexpId = expId &
         c.operator = Expression.cop(operator) &
         c.argument = arg &
         c.type = CType[type.typeId] &
         c.elementType = CType[elementType.typeId] &
         result = c )

BasicExpression::
query mapBasicExpression(ob : Set(CExpression),
              aind : Set(CExpression),
              pars : Sequence(CExpression)) : CBasicExpression
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))

CollectionExpression::
query mapCollectionExpression(id : String, elems : Sequence(CExpression)) : CExpression
post:
  (elems.size = 0  =>
      result = createCOpCall(id, "new" + elementType.name + "List") ) &
  (elems.size > 0 & type.name = "Set"  =>
     result = createCBinOpCall(id, "insert" + elementType.name,
        mapCollectionExpression(id + "_f", elems.front), elems.last) ) &
  (elems.size > 0 & type.name = "Sequence"  =>
     result = createCBinOpCall(id, "append" + elementType.name,
        mapCollectionExpression(id + "_f", elems.front), elems.last) )
```

This style of specification involves the use of update operations that also return results (or query operations that create objects and have side-effects), which is considered undesirable. Such operations cannot be translated into the B formalism for verification. The operation precondition 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, eg.:

```
CBinaryExpression::
mapCBinaryExpression(lexp : Expression,
             rexp : Expression) : BinaryExpression
pre:
  lexp = Expression[left.cexpId] &
  rexp = Expression[right.cexpId]
post:
    BinaryExpression->exists( c | c.expId = cexpId &
         c.operator = CExpression.uop(operator) &
         c.left = lexp & c.right = rexp &
         c.type = Type[type.ctypeId] &
         c.elementType = Type[elementType.ctypeId] &
         result = c )

CUnaryExpression::
mapCUnaryExpression(arg : Expression) : Expression
```

```
pre:
  arg = Expression[argument.cexpId]
post:
     UnaryExpression->exists( c | c.expId = cexpId &
         c.operator = CExpression.uop(operator) &
         c.argument = arg &
         c.type = Type[type.ctypeId] &
         c.elementType = Type[elementType.ctypeId] &
         result = c )

CBasicExpression::
query mapCBasicExpression(ob : Set(Expression),
              aind : Set(Expression),
              pars : Sequence(Expression)) : BasicExpression
pre:
  ob = Expression[reference.cexpId] &
  aind = Expression[arrayIndex.cexpId] &
  pars = Expression[parameters.cexpId]
post:
    (kind = "value"  =>
       result = mapCValueExpression(ob,aind,pars)) &
    (kind = "variable"  =>
       result = mapCVariableExpression(ob,aind,pars)) &
    (kind = "attribute"  =>
       result = mapCAttributeExpression(ob,aind,pars)) &
    (kind = "role"  =>
       result = mapCRoleExpression(ob,aind,pars)) &
    (kind = "operation"  =>
       result = mapCOperationExpression(ob,aind,pars)) &
    (kind = "classid"  =>
       result = mapCClassExpression(ob,aind,pars)) &
    (kind = "function"  =>
       result = mapCFunctionExpression(ob,aind,pars))
```

An alternative style of specification would be to use the Map Objects Before Links pattern [1], however this would involve separation of the mapping of expression instances and the mapping of relation instances: attribute values of target objects would be set in separate rules to the setting of their associations.

The detailed cases for mapping different forms of basic expression are as follows:

```
query mapValueExpression(ob : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
pre: umlKind = value
post:
  CBasicExpression->exists( c | c.cexpId = expId & c.kind = "value" &
       (data = "true" => c.data = "TRUE") &
       (data = "false" => c.data = "FALSE") &
       (data /= "true" & data /= "false" => c.data = data) &
       c.arrayIndex = aind &
       c.type = CType[type.typeId] &
       c.elementType = CType[elementType.typeId] & result = c)

query mapVariableExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  CBasicExpression->exists( c | c.cexpId = expId & c.kind = "variable" &
      c.data = data &
      c.arrayIndex = aind &
      c.type = CType[type.typeId] &
      c.elementType = CType[elementType.typeId] & result = c)

query mapAttributeExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  (objectRef.size = 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "attribute" &
            c.data = "get" + context.name + "_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            c.arrayIndex = aind &
            CBasicExpression->exists( s | s.data = "self" &
                   s.kind = "variable" & s : c.parameters &
                   s.cexpId = expId + "_self" &
                   s.type = CType[context.typeId] &
                   s.elementType = s.type ) &
            result = c ) )  &
  (objectRef.size > 0 & objectRef.any.type : CollectionType =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "attribute" &
            c.data = "getAll" + objectRef.any.elementType.name + "_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            obs <: c.parameters &
            c.arrayIndex = aind &
            result = c ) ) &
  (objectRef.size > 0 & objectRef.any.type /: CollectionType =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "attribute" &
            c.data = "get" + objectRef.any.elementType.name + "_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            obs <: c.parameters &
            c.arrayIndex = aind &
            result = c ) )
```

Note that *context* must be set for attributes and roles, so that the context entity using the feature can be accessed (this may be a subclass of the owner of the feature).

```
query mapRoleExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  (objectRef.size = 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "role" &
            c.data = "get" + context.name + "\_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            c.arrayIndex = aind &
            CBasicExpression->exists( s | s.data = "self" & s.kind = "variable" &
            s : c.parameters & s.cexpId = expId + "_self" &
            s.type = CType[context.typeId] &
            s.elementType = s.type ) & result = c ) )  &
  (objectRef.size > 0 & objectRef.any.type : CollectionType =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "role" &
            c.data = "getAll" + objectRef.any.elementType.name + "_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            obs <: c.parameters &
            c.arrayIndex = aind &
            result = c ) ) &
  (objectRef.size > 0 & objectRef.any.type /: CollectionType =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "role" &
            c.data = "get" + objectRef.any.elementType.name + "\_" + data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            obs <: c.parameters &
            c.arrayIndex = aind &
            result = c ) )

query mapOperationExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  (objectRef.size = 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "operation" &
            c.data = data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            c.arrayIndex = aind &
            CBasicExpression->exists( s | s.data = "self" & s.kind = "variable" &
                  s : c.parameters & s.cexpId = expId + "_self" &
                  s.type = CType[context.typeId] &
                  s.elementType = s.type ) &
            pars <: c.parameters & result = c ) )  &
  (objectRef.size > 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "operation" &
            c.data = data &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            c.parameters = obs^pars &
            c.arrayIndex = aind &
            result = c ) )

query mapClassExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  (arrayIndex.size > 0 & arrayIndex.any.type : CollectionType  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.kind = "classid" &
            c.data = "get" + elementType.name + "ByPKs" &
            aind <: c.parameters &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] & result = c) ) &
  (arrayIndex.size > 0 & arrayIndex.any.type /: CollectionType  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.kind = "classid" &
            c.data = "get" + elementType.name + "ByPK" &
            aind <: c.parameters &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] & result = c) ) &
  (arrayIndex.size = 0  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.kind = "classid" &
            c.data = data.toLowerCase() + "_instances" &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] & result = c) )
```

```
query mapFunctionExpression(obs : Set(CExpression),
      aind : Set(CExpression),
      pars : Sequence(CExpression)) : CBasicExpression
post:
  (Expression.isCfunction1(data)  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.data = data &
          c.kind = "function" &
          c.parameters = obs^pars &
          c.type = CType[type.typeId] &
          c.elementType = CType[elementType.typeId] & result = c) ) &
  (data = "allInstances"  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.kind = "function" &
            c.data = data.toLowerCase() + "_instances" &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] & result = c) ) &
  (true  =>
       CBasicExpression->exists( c | c.cexpId = expId & c.data = Expression.cfunctionName(data) &
          c.kind = "function" &
          c.parameters = obs^pars &
          c.type = CType[type.typeId] &
          c.elementType = CType[elementType.typeId] & result = c) )
```

Testing of these operations revealed some errors regarding the metamodels, eg., that *elementType* should be of 1 multiplicity, not 0..1, and that *any* is needed with *referredProperty* because this is of 0..1 multiplicity. *name* is needed for the copies of *Type* and *CType* in the OCL metamodel. The generation of model.txt needed to be adjusted in several cases to ensure that appropriate information was available for the code generator.

*isCFunction*1 is true if the function name is one of: sqrt, exp, log, sin, cos, tan, pow, log10, cbrt, tanh, cosh, sinh, asin, acos, atan.

The correspondence of OCL and C operators is given in Table 12.

UML operator	*C operator*	*Condition*
+, - (unary)	+, - (unary)	
not	!	
+, - (binary)	+, -	numeric arguments
$*$, /, mod	$*$, /, %	
and, or	&&, ||	
=	==	numeric arguments
<>, / =	!=	numeric arguments
<, ≤	<, ≤	numeric arguments
≥, >	≥, >	numeric arguments

Table 12: Operator correspondence between UML and C

These mappings are defined in an operation

```
Expression::
static query cop(aop : String) : String
post:
  (aop = "&"  =>  result = "&&") &
  (aop = "or"  =>  result = "||") &
  (aop = "not"  =>  result = "!") &
  (aop = "mod"  =>  result = "%") &
  (aop = "="  =>  result = "==") &
  (aop = "/="  =>  result = "!=") &
  (true  =>  result = aop)
```

```
Expression::
static query cfunctionName(f : String) : String
post:
  (f = "round"  =>  result = "oclRound") &
  (f = "floor"  =>  result = "oclFloor") &
  (f = "ceil"  =>  result = "oclCeil") &
  (f = "abs"  =>  result = "fabs") &
  (f = "->toLowerCase" => result = "toLowerCase") &
  (f = "->toUpperCase" => result = "toUpperCase") &
  (f = "->hasPrefix" or f = "hasPrefix" => result = "startsWith") &
  (f = "->hasSuffix" or f = "hasSuffix" => result = "endsWith") &
  (f = "->characters"  =>  result = "characters") &
  (f = "->toInteger"  =>  result = "atoi") &
  (f = "->toReal"  =>  result = "atof") &
  (f = "->display"  =>  result = "displayNumber") &
  (true  =>  result = f)
```

The final clause here is an 'else' case.
Various convenience operations were introduced during design:

```
Expression::
createCBinOpCall(id : String, name : String, le : CExpression,
        re: CExpression) : CBasicExpression
post:
  CBasicExpression->exists( cbe |
        cbe.cexpId = id & cbe.data = name &
        le : cbe.parameters & re : cbe.parameters &
        cbe.type = CType[type.typeId] &
        cbe.elementType = CType[elementType.typeId] & result = cbe )

Expression::
createCUnaryOpCall(id : String, name : String, arg : CExpression) : CBasicExpression
post:
  CBasicExpression->exists( cbe |
        cbe.cexpId = id &
        cbe.data = name &
        arg : cbe.parameters &
        cbe.type = CType[type.typeId] &
        cbe.elementType = CType[elementType.typeId] & result = cbe )

Expression::
createCOpCall(id : String, name : String) : CBasicExpression
post:
  CBasicExpression->exists( cbe |
        cbe.cexpId = id &
        cbe.data = name &
        cbe.type = CType[type.typeId] &
        cbe.elementType = CType[elementType.typeId] & result = cbe )


Expression::
static cast(typ : String, e : CExpression) : CExpression
post:
  CUnaryExpression->exists( ce | ce.cexpId = e.cexpId + "_cast" &
            ce.operator = "(" + typ + ") " &
            ce.argument = e &
            ce.type = e.type &
            ce.elementType = e.elementType &
            result = ce )
```

These are used to create C representations of binary and unary expressions as operation calls. They are an application of the pattern Factor out Duplicated Expression Evaluations.

For certain kinds of binary expression (+, -, =, /=, comparitors), the type of the arguments determines the representation:

```
BinaryExpression::
query mapAddExpression(le : CExpression, re : CExpression) : CExpression
post:
  (left.type.name = "String" & right.type.name = "String"  =>
         result = createCBinOpCall(expId, "concatenateStrings", le, re)) &
  (true =>
         CBinaryExpression->exists( ce | ce.cexpId = expId & ce.operator = "+" &
                 ce.left = le & ce.right = re &
                 ce.type = CType[type.typeId] &
                 ce.elementType = CType[elementType.typeId] &
                 result = ce ) )

BinaryExpression::
query mapSubtractExpression(le : CExpression, re : CExpression) : CExpression
post:
  (left.type.name = "String" & right.type.name = "String"  =>
         result = createCBinOpCall(expId, "subtractString", le, re)) &
  (left.type.name = "Set" or left.type.name = "Sequence"  =>
         result = createCBinOpCall(expId, "removeAll" + left.elementType.name, le, re)) &
  (true =>
         CBinaryExpression->exists( ce | ce.operator = "-" &
                 ce.cexpId = expId &
                 ce.left = le & ce.right = re &
                 ce.type = CType[type.typeId] &
                 ce.elementType = CType[elementType.typeId] &
                 result = ce ) )

BinaryExpression::
query mapComparitorExpression(le : CExpression, re : CExpression) : CExpression
post:
  (left.type.name = "String" & right.type.name = "String"  =>
         CBinaryExpression->exists( be | be.cexpId = expId &
             be.operator = Expression.cop(operator) &
             be.left = createCBinOpCall(expId + "_strcmp", "strcmp", le, re) &
             CBasicExpression->exists( zero | zero.data = "0" &
                 zero.cexpId = expId + "_0" &
                 be.right = zero ) &
             be.needsBracket = true &
             result = be ) ) &
  (true =>
         CBinaryExpression->exists( ce | ce.cexpId = expId & ce.operator = operator &
                 ce.left = le & ce.right = re &
                 ce.type = CType[type.typeId] &
                 ce.elementType = CType[elementType.typeId] &
                 result = ce ) )

BinaryExpression::
query mapEqualityExpression(le : CExpression, re : CExpression) : CExpression
post:
  (left.type.name = "String" & right.type.name = "String"  =>
         result = mapComparitorExpression(le, re)) &
  (left.type.name = "Set"  =>
         result = createCBinOpCall(expId, "equalsSet",
             Expression.cast("void**", le),
             Expression.cast("void**", re)) ) &
  (left.type.name = "Sequence"  =>
         result = createCBinOpCall(expId, "equalsSequence",
             Expression.cast("void**", le),
             Expression.cast("void**", re)) ) &
  (true =>
         CBinaryExpression->exists( ce | ce.cexpId = expId & ce.operator = "==" &
                 ce.left = le & ce.right = re &
                 ce.type = CType[type.typeId] &
                 ce.elementType = CType[elementType.typeId] &
                 result = ce ) )

BinaryExpression::
query mapInclusionExpression(le : CExpression, re : CExpression) : CExpression
post:
  (operator = ":"  =>
         result = createCBinOpCall(expId, "isIn",
              Expression.cast("void*", le),
              Expression.cast("void**", re)) ) &
  (operator = "->includes"  =>
         result = createCBinOpCall(expId, "isIn",
              Expression.cast("void*", re),
              Expression.cast("void**", le)) ) &
  (operator = "->includesAll"  =>
         result = createCBinOpCall(expId, "containsAll",
              Expression.cast("void**",le),
              Expression.cast("void**",re)) ) &
  (operator = "<:" =>
      result = createCBinOpCall(expId, "containsAll",
              Expression.cast("void**", re),
              Expression.cast("void**", le)) )

BinaryExpression::
query mapExclusionExpression(le : CExpression, re : CExpression) : CExpression
post:
  (operator = "/:"  =>
         CUnaryExpression->exists( nin | nin.cexpId = expId & nin.operator = "!" &
              nin.argument = createCBinOpCall(expId + "_isIn", "isIn",
                  Expression.cast("void*", le),
                  Expression.cast("void**", re)) &
              nin.type = CType[type.typeId] &
              nin.elementType = CType[elementType.typeId] &
              result = nin ) ) &
  (operator = "->excludes"  =>
        CUnaryExpression->exists( nin | nin.cexpId = expId & nin.operator = "!" &
              nin.argument = createCBinOpCall(expId + "_isIn", "isIn",
                     Expression.cast("void*", re),
                     Expression.cast("void**", le)) &
              nin.type = CType[type.typeId] &
              nin.elementType = CType[elementType.typeId] &
              result = nin ) ) &
  (operator = "->excludesAll"  =>
         result = createCBinOpCall(expId, "disjoint",
               Expression.cast("void**", le),
               Expression.cast("void**", re)) ) &
  (operator = "/<:" =>
      CUnaryExpression->exists( nin | nin.cexpId = expId & nin.operator = "!" &
              nin.argument = createCBinOpCall(expId + "_containsAll", "containsAll",
                Expression.cast("void**", re),
                Expression.cast("void**", le)) &
             nin.type = CType[type.typeId] &
             nin.elementType = CType[elementType.typeId] &
              result = nin ) )
```

Binary String expressions are mapped by:

```
query mapStringExpression(le : CExpression, re : CExpression) : CExpression
post:
  (operator = "->count"  =>
         result = createCBinOpCall(expId, "countString", le, re) ) &
  (operator = "->indexOf"  =>
        result = createCBinOpCall(expId, "indexOfString", le, re) ) &
  (true =>
      CBasicExpression->exists( be | be.cexpId = expId &
             be.data = Expression.cfunctionName(operator) &
             be.parameters = Sequence{ le, re } &
             result = be ) )
```

Quantifier, select/reject and collect expressions are mapped by:

```
query mapIteratorExpression(op : String, le : CExpression, re : CExpression) : CExpression
post:
  result = createCBinOpCall(expId, op + left.elementType.name, le,
              CExpression.defineCOpRef(CProgram.allInstances.any.defineCOp(re,
                    variable, le.elementType) ) )

query mapCollectExpression(le : CExpression, re : CExpression) : CExpression
post:
  result = createCBinOpCall(expId, op + left.elementType.name, le,
              CExpression.defineCOpRefCast(CProgram.allInstances.any.defineCOp(re,
                    variable, le.elementType), "void* (*)(" + le.elementType + ")" ) )
```

These create new operations that evaluate the predicate of the iterator/collect (the RHS argument), add it to the program, and create a reference to this operation as an argument for the C operation that evaluates the iterator/collect. For collect, the function reference must be cast to the function pointer type void* (*)(struct E*) where the LHS has element type E.

Sorted-by expressions are mapped by:

```
BinaryExpression::
query mapSortByExpression(arg : CExpression) : CExpression
post:
  result = Expression.cast("struct " + elementType.name + "**",
        createCBinOpCall(expId, "treesort", Expression.cast("void**", arg), CExpression.defineCOp
```

Comparitors *compareToint*, *compareTolong*, etc are already defined in ocl.h.

The definition of binary expression mapping is updated to:

```
BinaryExpression::
query mapBinaryExpression(lexp : CExpression,
            rexp : CExpression) : CBinaryExpression
pre:
  lexp = CExpression[left.expId] &
  rexp = CExpression[right.expId]
post:
  (operator = "+"  =>
     result = mapAddExpression(lexp, rexp)) &
  (operator = "-"  =>
     result = mapSubtractExpression(lexp, rexp)) &
  (operator = "="  =>
     result = mapEqualityExpression(lexp,rexp)) &
  (Expression.isComparitor(operator)  =>
    result = mapComparitorExpression(lexp,rexp)) &
  (Expression.isInclusion(operator)  =>
    result = mapInclusionExpression(lexp,rexp)) &
  (Expression.isExclusion(operator)  =>
    result = mapExclusionExpression(lexp,rexp)) &
  (Expression.isIteratorOp(operator)  =>
    result = mapIteratorExpression(operator.tail.tail, lexp, rexp)) &
  (operator = "->collect"  =>
    result = Expression.cast(rexp.type + "*", mapCollectExpression(lexp, rexp))) &
  (operator = "->sortedBy"  =>
     result = mapSortByExpression(lexp)) &
  (left.type.name = "String" & Expression.isStringOp(operator)  =>
    result = mapStringExpression(lexp,rexp)) &
  ((left.type.name = "Set" or left.type.name = "Sequence") &
    Expression.isCollectionOp(operator)  =>
    result = mapCollectionExpression(lexp,rexp)) &
  (true =>
     CBinaryExpression->exists( c | c.cexpId = expId &
         c.operator = Expression.cop(operator) &
         c.left = lexp & c.right = rexp &
         c.type = CType[type.typeId] &
         c.elementType = CType[elementType.typeId] &
         result = c ) )
```

A string binary op is one of →*indexOf*, →*count*, →*hasPrefix*, →*hasSuffix*. An iterator operator is one of →*forAll*, →*exists*, →*exists*1, →*select*, →*reject*.

A collection binary op is one of: →*including*, →*excluding*, →*append*, →*indexOf*, →*count*, →*union*, →*intersection*, $\cup$, $\cap$, $\frown$, →*isUnique*, →*sortedBy*. These are mapped as follows:

```
BinaryExpression::
query mapCollectionExpression(le : CExpression, re : CExpression) : CExpression
post:
  (operator = "->including" & left.type.name = "Set"  =>
        result = createCBinOpCall(expId, "insert" + left.elementType.name, le, re) ) &
  (operator = "->including" & left.type.name = "Sequence"  =>
       result = createCBinOpCall(expId, "append" + left.elementType.name, le, re) ) &
  (operator = "->excluding"  =>
       result = createCBinOpCall(expId, "remove" + left.elementType.name, le, re) ) &
  (operator = "->append"  =>
       result = createCBinOpCall(expId, "append" + left.elementType.name, le, re) ) &
  (operator = "->count"  =>
       result = createCBinOpCall(expId, "count",
                         Expression.cast("void*", re),
                         Expression.cast("void**", le)) ) &
  (operator = "->indexOf"  =>
       result = createCBinOpCall(expId, "indexOf",
                         Expression.cast("void*", re),
                         Expression.cast("void**", le)) ) &
  (operator = "->union" or operator = "\\/"  =>
       result = createCBinOpCall(expId, "union" + left.elementType.name, le, re) ) &
  (operator = "^"  =>
       result = createCBinOpCall(expId, "concatenate" + left.elementType.name, le, re) ) &
  (operator = "->intersection" or operator = "/\\"  =>
       result = createCBinOpCall(expId, "intersection" + left.elementType.name, le, re) ) &
  (operator = "->isUnique"  =>
       result = createCBinOpCall(expId,
           "isUnique" + left.elementType.name, le, re) )
```

Unary expressions are mapped as follows:

```
UnaryExpression::
query mapUnaryExpression(arg : CExpression) : CExpression
pre:
  arg = CExpression[argument.expId]
post:
  (operator.size > 2 & Expression.isCfunction1(operator.tail.tail)  =>
      CBasicExpression->exists( c | c.cexpId = expId &
          c.data = Expression.cfunctionName(operator.tail.tail) &
          c.kind = "function" &
          c.parameters = Sequence{ arg } &
          c.type = CType[type.typeId] &
          c.elementType = CType[elementType.typeId] & result = c) ) &
  (operator = "->sort" => result = mapSortExpression(arg)) &
  (Expression.isReduceOp(operator)  =>
          result = mapReduceExpression(arg)) &
  (argument.type.name = "String" & Expression.isUnaryStringOp(operator)  =>
          result = mapStringExpression(arg)) &
  (operator = "->display"  =>
         result = createCUnaryOpCall(expId, "display" + argument.type, arg)) &
  ((argument.type.name = "Set" or argument.type.name = "Sequence") &
    Expression.isUnaryCollectionOp(operator)  =>
          result = mapCollectionExpression(arg)) &
  (true  =>
      CUnaryExpression->exists( c | c.cexpId = expId &
         c.operator = Expression.cop(operator) &
         c.argument = arg &
         c.type = CType[type.typeId] &
         c.elementType = CType[elementType.typeId] &
         result = c ) )
```

A reduce operator is one of $\rightarrow min$, $\rightarrow max$, $\rightarrow sum$, $\rightarrow prd$. These expressions are mapped by:

```
UnaryExpression::
query mapReduceExpression(arg : CExpression) : CExpression
post:
  (operator[1] = "-" & operator[2] = ">"   =>
         result = createCBinOpCall(expId,
            operator.tail.tail + argument.elementType.name, arg, argument.clength(arg)) ) &
  (true  =>
        result = createCBinOpCall(expId, operator + argument.elementType.name, arg, argument.clen
```

clength by default returns *length*((*void*∗∗) *arg*), but for $s \rightarrow collect$ on a collection, and for attribute applications *s.att* to a collection, it returns the size of s'.

Sort expressions are mapped by:

```
UnaryExpression::
query mapSortExpression(arg : CExpression) : CExpression
post:
  (elementType.name = "String"  =>
       result = Expression.cast("char**", createCBinOpCall(expId, "treesort",
           Expression.cast("void**", arg), CExpression.defineCOpReference("strcmp", "int"))) ) &
  (true  =>
      result = Expression.cast("struct " + elementType.name + "**",
        createCBinOpCall(expId, "treesort", Expression.cast("void**", arg), CExpression.defineCOp
```

Unary String expressions are mapped by:

```
UnaryExpression::
query mapStringExpression(arg : CExpression) : CExpression
post:
  (operator = "->size"  =>
         result = createCUnaryOpCall(expId, "strlen", arg) ) &
  (operator = "->first"  =>
        result = createCUnaryOpCall(expId, "firstString", arg) ) &
  (operator = "->last"  =>
        result = createCUnaryOpCall(expId, "lastString", arg) ) &
  (operator = "->front"  =>
        result = createCUnaryOpCall(expId, "frontString", arg) ) &
  (operator = "->tail"  =>
        result = createCUnaryOpCall(expId, "tailString", arg) ) &
  (operator = "->reverse"  =>
        result = createCUnaryOpCall(expId, "reverseString", arg) ) &
  (operator = "->display"  =>
        result = createCUnaryOpCall(expId, "displayString", arg) ) &
  (true =>
      result = createCUnaryOpCall(expId, Expression.cfunctionName(operator), arg) )
```

A (unary) collection operator is one of $\rightarrow$*size*, $\rightarrow$*any*, $\rightarrow$*reverse*, $\rightarrow$*front*, $\rightarrow$*last*, $\rightarrow$*tail*, $\rightarrow$*first*, $\rightarrow$*sort*, $\rightarrow$*asSet*, $\rightarrow$*asSequence*. These are mapped by:

```
UnaryExpression::
query mapCollectionExpression(arg : CExpression) : CExpression
post:
  (operator = "->size"  =>
         result = createCUnaryOpCall(expId, "length", Expression.cast("void**", arg)) ) &
  (operator = "->any"  =>
        result = createCUnaryOpCall(expId, "any", Expression.cast("void**", arg)) ) &
  (operator = "->first"  =>
        result = createCUnaryOpCall(expId, "first", Expression.cast("void**", arg)) ) &
  (operator = "->last"  =>
        result = createCUnaryOpCall(expId, "last", Expression.cast("void**", arg)) ) &
  (operator = "->front"  =>
        result = createCUnaryOpCall(expId, "front" + elementType.name, arg) ) &
  (operator = "->tail"  =>
        result = createCUnaryOpCall(expId, "tail" + elementType.name, arg) ) &
  (operator = "->reverse"  =>
        result = createCUnaryOpCall(expId, "reverse" + elementType.name, arg) ) &
  (operator = "->asSet"  =>
        result = createCUnaryOpCall(expId, "asSet" + elementType.name, arg) ) &
  (operator = "->asSequence"  =>
        result = arg ) &
  (true =>
      result = createCUnaryOpCall(expId, Expression.cfunctionName(operator), arg) )
```

For any, first, last, the result also needs to be cast to arg.elementType.

There are 44 operations and 3 transformation rules for this use case. Testing and inspection were used for validation and verification. The estimated effort for this iteration is shown in Table 13.

Stage	*Effort (person days)*
Req. Elicitation	10
Eval./Negotiation	2
Specification	30
Review/Validation	30
Implementation/ Testing	20
Total	92

Table 13: Development effort for Iteration 3

5 Iteration 4: Activities mapping

In this iteration, UML-RSDS activities are mapped to C statements by a subtransformation *statements2C*. Figure 7 shows the input activity language. UML-RSDS statements correspond closely to those of C. Figure 8 shows the metamodel for C statements. Table 14 shows the main cases of the mapping of UML activities to C statements.

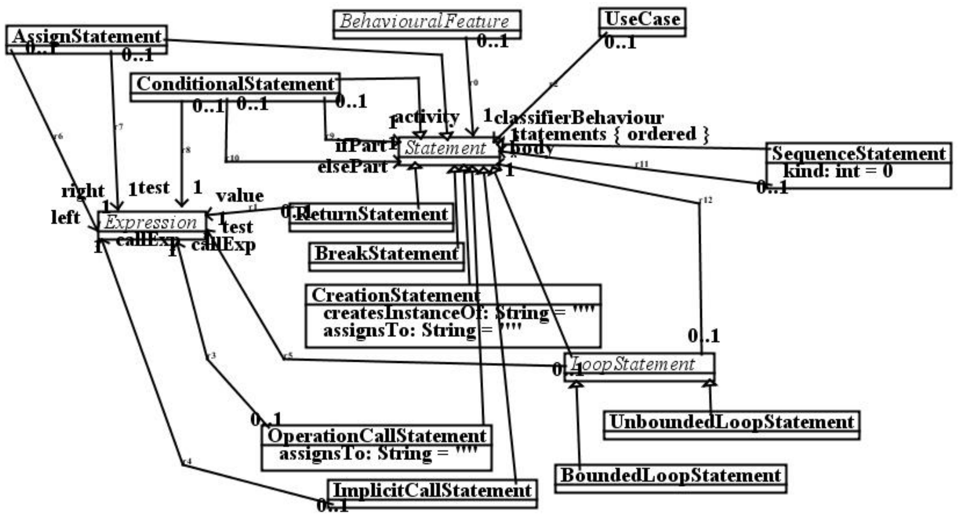

Figure 7: UML-RSDS activity metamodel

New identity attributes *statId* and *cstatId* are added to *Statement* and *CStatement*, respectively, to support the bx and traceability requirements. A similar recursive descent style as for *expressions2C* is used for the statement mapping specification. An operation

```
mapStatement() : CStatement
```

is defined in each Statement subclass.

For basic statements, this is defined as follows:

```
AssignStatement::
mapStatement() : CStatement
post:
    CAssignment->exists( ca | ca.cstatId = statId &
         ca.left = left.mapExpression() &
         ca.right = right.mapExpression() &
         result = ca )
```

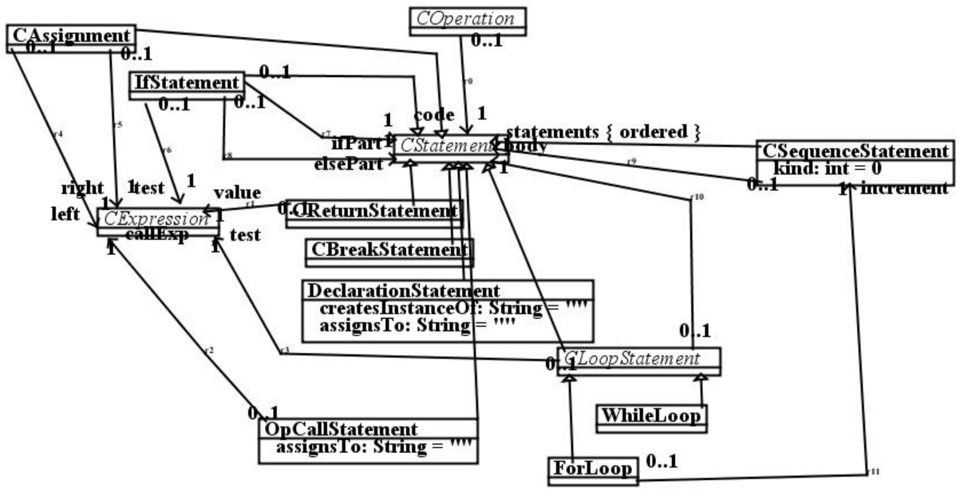

Figure 8: C statement metamodel

Requirement	*UML activity st*	*C statement st'*
F1.4.1	Creation statement `x : T` defaultT' is default value of T'	T' x = defaultT';
F1.4.2	Assign statement `v := e`	v' = e';
F1.4.3	Sequence statement `st1 ; ... ; stn`	st1' ... stn'
F1.4.4	Conditional statement `if e then st1 else st2`	if e' { st1' } else { st2' }

F1.4.5
F1.4.6
	Return statement `return e`
	Break statement `break`
		return e';
		break;

F1.4.7	Bounded loop `for (x : e) do st`
	on object collection e of entity
	element type E
		int i = 0;
		for (; i < length((void**) e'); i++)
		{ struct E* x = e'[i]; st' }
		New index variable i

F1.4.8	Unbounded loop `while e do st`	while (e') { st' }
F1.4.9	Operation call `ex.op(pars)`	op(ex',pars')

Table 14: Scenarios for mapping of statements to C

```
BreakStatement::
mapStatement() : CStatement
post:
    CBreakStatement->exists( ca | ca.cstatId = statId & result = ca )

OperationCallStatement::
mapStatement() : CStatement
post:
    OpCallStatement->exists( ca | ca.cstatId = statId &
                ca.callExp = callExp.mapExpression() &
                ca.assignsTo = assignsTo & result = ca )

ImplicitCallStatement::
mapStatement() : CStatement
post:
    OpCallStatement->exists( ca | ca.cstatId = statId &
                ca.callExp = callExp.mapExpression() &
                ca.assignsTo = assignsTo & result = ca )

CreationStatement::
mapStatement() : CStatement
post:
    DeclarationStatement->exists( ds | ds.cstatId = statId &
                 ds.createsInstanceOf = createsInstanceOf &
                 ds.assignsTo = assignsTo &
                 ds.type = CType[type.typeId] &
                 ds.elementType = CType[elementType.typeId] & result = ds)
```

For each category of composite statement, the subparts of the statement are mapped to C first, and then composed by a separate operation. For example:

```
SequenceStatement::
mapStatement() : CStatement
post:
  result = mapSequenceStatement(
                    statements.mapStatement())

mapSequenceStatement(css : Sequence(CStatement)) : CStatement
pre:
  css = CStatement[statements.statId]
post:
  CSequenceStatement->exists( cs | cs.cstatId = statId &
                 cs.kind = kind & cs.statements = css &
                 result = cs )

ConditionalStatement::
mapStatement() : CStatement
post:
  result = mapConditionalStatement(
                    ifPart.mapStatement(), elsePart.mapStatement())

mapConditionalStatement(ifP : CStatement, elseP : Sequence(CStatement)) : CStatement
pre:
  ifP = CStatement[ifPart.statId] &
  elseP = CStatement[elsePart.statId]
post:
  IfStatement->exists( istat | istat.cstatId = statId &
                        istat.ifPart = ifP &
                        istat.elsePart = elseP &
                        istat.test = test.mapExpression() &
                        result = istat )

UnboundedLoopStatement::
mapStatement() : CStatement
post:
  result = mapUnboundedLoopStatement(
                           body.mapStatement())

mapUnboundedLoopStatement(bdy : CStatement) : CStatement
pre:
  bdy = CStatement[body.statId]
post:
  WhileLoop->exists( lp | lp.cstatId = statId &
                    lp.body = bdy & lp.test = test.mapExpression() &
                    result = lp )

BoundedLoopStatement::
mapStatement() : CStatement
post:
  result = mapBoundedLoopStatement(
                           body.mapStatement())

mapBoundedLoopStatement(bdy : CStatement) : CStatement
pre:
  bdy = CStatement[body.statId]
post:
  ForLoop->exists( lp | lp.cstatId = statId &
                    lp.body = bdy & lp.test = test.mapExpression() &
                    lp.loopVar = loopVar.mapExpression() &
                    lp.loopRange = loopRange.mapExpression() &
                    result = lp )
```

The definitions of these mappings were revised and simplified during the specification stage, prior to implementation.

The mappings have inverses:

```
CSequenceStatement::
mapCStatement() : Statement
post:
  result = mapCSequenceStatement(
                     statements.mapCStatement())

mapCSequenceStatement(css : Sequence(Statement)) : Statement
pre:
  css = Statement[statements.cstatId]
post:
   SequenceStatement->exists( cs | cs.statId = cstatId &
                   cs.kind = kind & cs.statements = css &
                   result = cs )

IfStatement::
mapCStatement() : Statement
post:
  result = mapIfStatement(
                     ifPart.mapCStatement(), elsePart.mapCStatement())
```

```
mapIfStatement(ifP : Statement, elseP : Sequence(Statement)) : Statement
pre:
  ifP = Statement[ifPart.cstatId] &
  elseP = Statement[elsePart.cstatId]
post:
  ConditionalStatement->exists( istat | istat.statId = cstatId &
                 istat.ifPart = ifP &
                 istat.elsePart = elseP &
                 istat.test = test.mapCExpression() &
                 result = istat )

WhileStatement::
mapCStatement() : Statement
post:
  result = mapWhileStatement(
                          body.mapCStatement())

mapWhileStatement(bdy : Statement) : Statement
pre:
  bdy = Statement[body.cstatId]
post:
  UnboundedLoopStatement->exists( lp | lp.statId = cstatId &
         lp.body = bdy & lp.test = test.mapCExpression() &
         result = lp )
```

and likewise for other composite statement cases.

The operation *mapExpression* is invoked from *mapStatement* in the cases of basic and composite statements which involve expressions. For example:

```
ReturnStatement::
mapStatement() : CStatement
post:
  CReturnStatement->exists( r |
             r.cstatId = statId &
             r.returnValue = returnValue.mapExpression() &
             result = r)
```

This can be inverted using *mapCExpression* to convert the value expressions from C to UML.

Statements are printed by the *genCtext* use case, eg.:

```
DeclarationStatement::
query toString() : String
post:
  (createsInstanceOf = "String"  =>
         result = type + " " + assignsTo + " = \"\"") &
  (type : CPrimitiveType  =>
         result = createsInstanceOf + " " + assignsTo + " = 0;") &
  (createsInstanceOf : CStruct.name  =>
         result = type + " " + assignsTo + " = NULL;") &
  (createsInstanceOf.startsWith("Sequence")  =>
       result = type + " " + assignsTo + " = new" + elementType.name + "List();") &
  (createsInstanceOf.startsWith("Set")  =>
       result = type + " " + assignsTo + " = new" + elementType.name + "List();")

CSequenceStatement::
query toString() : String
post:
  result = statements->collect( s | s.toString() + "\n   " )->sum()

IfStatement::
query toString() : String
post:
  (elsePart.size = 0  =>
      result = "if (" + test + ")\n  { " +
                 ifPart + " }") &
  (elsePart.size > 0  =>
      result = "if (" + test + ")\n  { " +
                 ifPart + " }\n  else { " + elsePart.any + " }")

ForStatement::
query toString() : String
post:
  ind = "ind_" + cstatId &
  result = "  int " + ind + " = 0;\n" +
     "  for ( ; " + ind + " < length((void**) " + loopRange + "); " + ind + "++)\n" +
     "  { " + loopRange.elementType + " " + loopVar + " = (" + loopRange + "[" + ind + "]);\n" +
    "    " + body + "\n" +
    "  } "
```

It was identified that the *elsePart* association ends should be 0..1 multiplicities to include the cases of If statements without Else clauses. Likewise, the returnValue of a Return statement should be optional. There are 24 operations and 1 rule for this iteration.

The estimated effort for this iteration is shown in Table 15.

Stage	*Effort (person days)*
Req. Elicitation	2
Eval./Negotiation	1
Specification	6
Review/Validation	6
Implementation/ Testing	4
Total	19

Table 15: Development effort for Iteration 4

6 Iteration 5: Use case mapping

In this iteration, the mapping *usecases2C* of use cases is specified and implemented. A large part of this iteration was also taken up with integration testing of the complete transformation.

F1.5.1: A use case *uc* is mapped to a C operation with "application" scope, and with parameters corresponding to those of *uc*. Its code is given by the C translation of the activity *classifierBehaviour* of *uc*.

F1.5.2: Included use cases are also mapped to operations, and invoked from the including use case.

F1.5.3: Operation activities are mapped to C code for the corresponding COperation.

F1.5.1 and F1.5.2 are formalised as:

```
UseCase::
  COperation->exists( cop | cop.name = name & cop.scope = "application" &
        cop.isQuery = false &
```

```
      cop.code = classifierBehaviour.mapStatement() &
      parameters->forAll( x | CVariable->exists( y | y.name = x.name &
                  y.kind = "parameter" &
                  y.type = CType[x.type.typeId] &
                  y : cop.parameters ) ) &
      cop.returnType = CType[returnType.typeId] )
```

Similarly for the activities of UML operations (F1.5.3):

```
Operation::
  COperation->exists( cop | cop.opId = name + "_" + owner.name &
        cop.code = activity.mapStatement() )
```

The other features of the COperation in this case are set by the iteration 2 mapping.

All auxiliary scope operations are printed before all entity operations, and these are printed before all application operations:

```
COperation::
  scope = "auxiliary"  =>  self->display()

COperation::
  scope = "entity"  =>  self->display()

COperation::
  scope = "application"  =>  self->display()
```

The result of iterations 3, 4 and 5 is a transformation that operates on the 3 UML-RSDS metamodels (a subset of Figure 3, and Figures 5 and 7) as inputs, and on a subset of the C general metamodel (Figure ??) and on Figures 6, 8 as outputs. The completed prototype for iterations 3, 4, 5 is uml2Cb.jar. It reads cmodel.txt produced by uml2Ca.jar and writes the generated C program to the standard output.

There are 6 rules in this use case. Table 16 shows the resources used by this iteration.

Stage	*Effort (person days)*
Req. Elicitation	1
Eval./Negotiation	0
Specification	1
Review/Validation	1
Implementation/ Testing	5
Total	8

Table 16: Development effort for Iteration 5

7 Comparison with manual transformation development

Several code generators have previously been developed for UML-RSDS: for Java 4, Java 6, Java 7, C# and C++. Each of these was developed using an agile development process but with manual coding in Java. Table 17 shows the approximate effort in person-months expended for each of these to date. The generators for Java 6, 7 and C# used very similar strategies and extensively reused the code of the Java 4 version generator.

The best comparison with the C code generator is perhaps the C++ generator, which involved considerable background research into the semantics, language and libraries of C++, and

	Java 4	Java 6	Java 7	C#	C++
Req. Anal.	6	1	2	3	6
Coding	12	3	4	4	6
Testing	6	1	1	1	2
Maintenance	6	1	1	1	3
Total	30	6	8	9	17

Table 17: Development effort for previous code generators

significant revision of the existing Java-oriented code generator. Likewise, the C code generator involved substantial new work on the code generation stategy, in addition to the technical challenge of implementing this strategy.

Summarising Tables 2, 4, 13, 15, 16, we obtain an overall estimate for the C code generator in Table 18.

Stage	*Effort (person days)*
Req. Elicitation	17
Eval./Negotiation	5
Specification	56
Review/Validation	57
Implementation/ Testing	49
Total	184

Table 18: Overall development effort for C code generator

This amounts to 4.5 person months for requirements analysis/specification activities, compared to 6 months for the manually-developed C++ generator (which had no specification). 49 days were spent on implementation and testing, compared to 8 months for the C++ generator. A major factor in this difference is the simpler and more concise transformation specification of the C code generator (expressed in UML-RSDS) compared to the Java code of the C++ code generator.

8 Future work

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 [2]. For this subset, dynamic memory allocation is not permitted, so for each class, a maximum bound must be provided to the number of objects of the class. Classes can again be represented by C structs, but *e_instances* would be an array of structures, instead of an array of pointers to structures. Objects are represented as ints indexing into these arrays, and collections are also represented as fixed-size arrays. The code generator satisfies most of the code structuring restrictions of MISRA C, and union data structures and other cases of overlapping memory usage are already excluded. Recursive functions are not permitted, so an alternative means of sorting (not treesort or qsort) must be used.

Conclusions

This case study is the largest transformation which has been developed using UML-RSDS, in terms of the number of rules (of the order of 150 rules/operations in 5 subtransformations). By using a systematic requirements engineering and agile development approach, we were able to effectively modularise the transformation and to organise its structure and manage its requirements. Despite the complexity of the transformation, it was possible to use patterns to enforce bx and other properties, and to effectively prove these properties.

References

[1] K. Lano, S. Kolahdouz-Rahimi, *Model-transformation Design Patterns*, IEEE Transactions in Software Engineering, vol 40, 2014.

[2] MIRA Ltd., *MISRA-C:2004 Guidelines for the use of the C language in critical systems*, 2004.

[3] OMG, *Semantics of Business vocabulary rules (SBVR)*, Version 1.2 (2013), www.omg.org/spec/SBVR/1.2/PDF.