Transformation from UML to C: A large-scale example of MDD for model transformation development

K. Lano, S. Yassipour-Tehrani, H. Alfraihi
Dept. of Informatics, Strand, King's College London WC2R 2LS
Phone: 0207 848 2832. Fax: 0207 848 2851
S. Kolahdouz-Rahimi,
Dept. of Software Engineering, University of Isfahan, Iran
kevin.lano@kcl.ac.uk, s.yassipour_tehrani@kcl.ac.uk, hessa.alfraihi@kcl.ac.uk
sh.rahimi@eng.ac.ir

May 18, 2018

Abstract

In this paper we describe a substantial example of model-driven development (MDD) applied to model transformation (MT) development. A detailed requirements engineering process was followed, together with an agile MDD process. The transformation maps UML specifications to ANSI C code, and it is written using the UML-RSDS MDD specification language. We evaluate the effectiveness of the MDD approach, and show that it has benefits compared to MT construction using conventional development of code generators.

1 Introduction

Model transformations such as code generators are essential to model-driven engineering (MDE) and model-driven development (MDD). However, the construction of transformations has not generally involved rigorous development approaches, instead transformations have often been developed in an ad-hoc manner, dependent on the characteristics of (in many cases) still experimental MT languages [23, 6].

In this paper we describe the systematic software engineering of a large scale code generator for mapping UML to ANSI C. Specifically, an agile MDD development process is used to produce the code generator. This case study will provide evidence for the feasibility and benefits of such a systematic approach to MT development.

Although the C language is now over 40 years old, it is still in widespread use, especially for applications requiring high efficiency and small code size. Therefore it was considered useful to provide such a generator. Existing C code generators for UML do not map detailed functional specifications (expressed, for example, in OCL or by UML activities) into C, and we wished to provide a generator which would implement a large part of the OCL library and provide full functional implementations of UML operation specifications and activities. In this respect, the generator can be used as a 'virtual machine' for the execution of UML/OCL and for model transformations (MT) expressed in UML/OCL, as an alternative to the more usual Java/JVM implementation route for UML [26]. A technical overview of the OCL translator was published in [14], in this paper we focus on the agile MDD software engineering process used to develop the translator.

We use the UML-RSDS subset of UML as the input language for the translator [13]. UML-RSDS models systems, including model transformations, using UML 2 class diagrams, OCL 2.4 and use cases at the specification level. State machines and interactions may optionally be used. At the design level, UML activities using a pseudocode notation are used. Relative to the fUML

executable subset of UML [21], UML-RSDS has a more declarative orientation. As in the case of fUML, certain UML notations and modelling elements have been excluded from UML-RSDS in order to achieve a more coherent and precise modelling language (Table 1). Applications (such as the UML to C code generator itself) are specified using class diagrams, use cases and OCL constraints with a clear semantic meaning, and designs are then generated semi-automatically from these. From the design, code in Java, C# or C++ can be automatically generated (Figure 1). The UML to C translator, UML2C, will take as input text files model.txt which define the design level of an application as an instance model of the UML-RSDS class diagram, OCL and activities metamodels (Figures 6, 8, 10). It is assumed that these models are syntactically correct wrt the metamodels, and that type checking and design generation steps have been performed prior to export of the data.

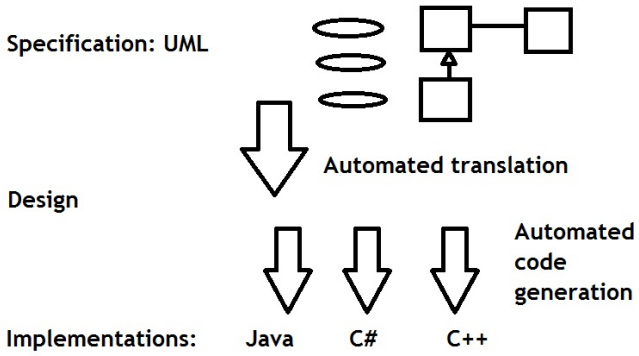

Figure 1: UML-RSDS software production process

UML2C is an example of the reflexive application of UML-RSDS to operate on UML-RSDS models. This technique enables the extension of the UML-RSDS tools with new capabilities such as customised XML production, document and code generators, analysis reports and new semantic mappings, etc. It also supports the use of UML-RSDS to define textual domain-specific languages (DSLs) and associated tools.

1.1 Requirements engineering

We have described a requirements engineering process for MT in [25], and explained how the RE techniques for the case study were selected. In this paper we give the details of the applications of the selected techniques (document mining, interviews, scenario analysis, goal decomposition, prototyping, UML specification, inspection, testing and refactoring).

The selected RE techniques were used in the case study as follows:

- *Document mining*: this involves comprehensive background research into relevant documents and software, specifically C standards, textbooks, compilers and forums, and review of existing code generators for C and the UML-RSDS code generators.
- *Interviews*: elicitation of requirements from stakeholders via structured interviews.
- *Scenario analysis*: detailed consideration of specific scenarios (model elements and structures of linked elements) which the translator should process. Both normal and abnormal (error) scenarios can be considered. Scenario analysis is a widely-used technique, however it can suffer from incompleteness, since in general it is not possible to identify all scenarios. In the case of model transformations this problem can be addressed by systematically considering each source language entity type and the relevant alternative structures of their instances which may occur in source models.

- *Goal decomposition*: this breaks down a requirement (eg., a user story) into subcases or subtasks which can be considered separately. As with scenarios, this decomposition was made on the basis of the source language structure.
- *Prototyping*: parts of the transformation are implemented in an initial form and tested using example models to identify if the intended mappings are correctly defined. This is an iterative process with successive refinement of the implementation based upon stakeholder feedback. In our case, the evolved prototype is also the final deliverable.
- *UML specification*: semi-formal specification of mappings, using concrete grammar of the source and target languages, is initially used to define the functional requirements. These specifications are then formalised as OCL constraints based upon the abstract syntax of the source and target languages.
- *Inspection*: systematic review of the specifications is performed to check their syntactic and semantic correctness, and their validity wrt requirements. As noted below in Section 7.2, we found that inspection of specifications was substantially more time-efficient than code inspection, corresponding to a 4-fold size reduction of the specification compared to executable code.
- *Testing*: test cases were manually constructed for each functional requirement. MDD provides the possibility of model-based test case generation, however we did not find any existing tools which would provide the necessary capability, and there were insufficient resources to develop our own.
- *Refactoring*: the specification structure was continuously reviewed and reorganised based on a number of MT design patterns and refactoring rules. This resulted in improvements in modularity, simplicity and a reduction in redundancy. Again, the use of specifications reduced the time and effort needed to perform refactoring/apply patterns compared to application of patterns or refactoring at code level.

We distinguish between *global* and *local* requirements for model transformations: a global requirement concerns properties of the source/target models or transformation considered as a whole (such as the syntactic correctness of the target model wrt its metamodel), whilst local requirements concern properties specific to particular types of model elements or particular kinds of structures in the models (such as a mapping requirement for the mapping of UML inheritance to C).

The initial requirements of the UML to C translator were identified using the elicitation techniques of interviews, document mining and scenarios. These produced the top-level functional requirement:

> **F1**: *Translate UML-RSDS designs (UML 2 class diagrams, OCL 2.4, 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.

As we have found in our empirical research [11], access to indirect stakeholders is often restricted in MT developments, because the transformation is often developed for MDD specialists, who are remote from end users of applications produced using the transformation. In this case direct access was only possible to stakeholders (i). Access to other stakeholders was substituted by research into the needs of such stakeholders, using document mining of sources such as C text books, the C standard, and specialised standards, particularly MISRA C [17]. The risk therefore remains that the C code produced may not be acceptable to category (ii) or (iii) 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. It was determined that the complete set of language restrictions of MISRA C would not be followed,

and instead the focus would be on supporting the implementation of UML in C for general users. Thus, we target the ANSI '89 standard version of C, as described in [7].

Using goal decomposition, the functional requirement was decomposed into five high-priority subgoals, each of which can be further decomposed into subgoals (Figure 3):

- 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.

This decomposition is based upon the main division of the UML-RSDS language into sublanguages, with types as a fundamental component which the other sublanguages depend upon, with OCL expressions defined using class diagram elements, and activities and use cases defined using OCL.

The mapping of OCL expressions depends upon the prior mapping of types and class structures, however we aimed for this mapping to be relatively independent of the strategy chosen for representing classes in C (eg., how inheritance and dynamic dispatch is expressed). To achieve this, all access to objects and their features in the C code is provided via an interface of creators, lookup operations and getters/setters which have a standard signature independent of their implementation details. This application API is defined in the header file app.h for each application. A library ocl.h of C functions for OCL operators is also defined, and the evaluation/execution of OCL expressions, operations and use cases is based upon app.h and ocl.h (Figure 2).

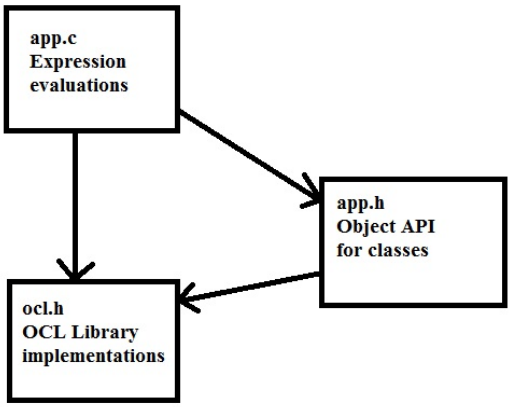

Figure 2: Application architecture of generated C code

The transformation structure is a sequential composition of separate transformations for the five sublanguage mappings, using the concept of *phasing* or *layering* organisation of transformations [9]. This structure of relatively independent transformation parts enables the development to be organised into five iterations, one for each translator part. The initial product backlog for the development consisted of F1.1 to F1.5, and the iteration backlog for the first iteration consisted of the subtasks of F1.1. 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.

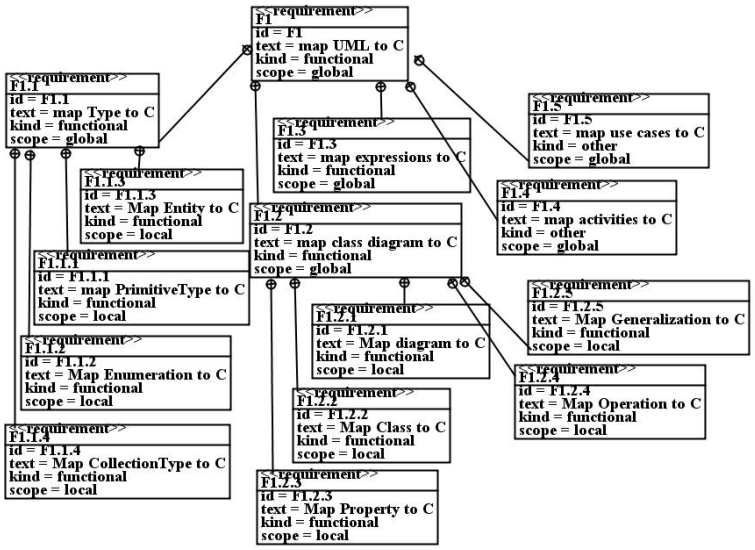

Figure 3: Functional requirements decomposition in SysML

- **F2**: Syntactic correctness: given a correct UML-RSDS design model as input, a valid C program will be produced.
- **F3**: Model-level semantic preservation: the semantics of the source (UML model) and target (C code or model) are equivalent.
- **F4**: Traceability: a record should be maintained of the correspondence between source and target elements.

Medium-level priority requirements were:

- **F5**: Bidirectionality (bx property) of the transformation between source and target models.
- **NF2**: Efficiency: input models with 100 classes and 100 attributes per class should be processed within 30 seconds.
- **NF3**: Modularity of the transformation.
- **NF6**: Produce efficient code, of similar or higher efficiency as equivalent hand-produced code.
- **NF7**: Produce compact code, of the same or smaller size as equivalent hand-produced code.

Low-priority requirements were:

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

There are potential conflicts between the requirements:

- NF2 conflicts with F4, F5 and NF3 because the additional structure needed for tracing and bx properties impairs efficiency, and the decomposition of the transformation into subtransformations composed sequentially also slows execution.
- NF4 conflicts with NF10 as the additional work required for NF4 would need substantial additional time resources.

- NF6 conflicts with F3 because in some cases semantic correctness will require inefficient coding, eg., because OCL collection operators produce modified copies of their arguments instead of updating them in-place.

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 transformation rules, which would otherwise need to combine language translation and text production. Figure 4 shows the resulting transformation architecture. This is an example of the architectural pattern Factor Code Generation into Model-to-model and Model-to-code [5, 16].

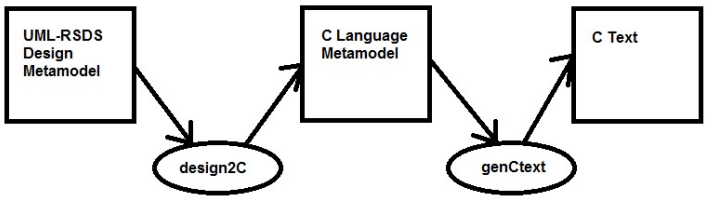

Figure 4: C code generator architecture

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 $design2C$.

Sections 2, 3, 4, 5 and 6 describe the five development iterations, Section 7 gives an evaluation, Sections 8, 9 describe related and future work, and Section 10 gives conclusions. The appendix summarises the C semantics which we use to justify semantic preservation.

1.2 Methodology

We adopted an agile MDD approach to develop the code generator.

The key principles of agile development include (agilemanifesto.org): (i) satisfy the customer through early and continuous software delivery; (ii) welcome changing requirements; (iii) deliver working software frequently (every 2 weeks to every 2 months); (iv) business people and developers to work together daily; (v) rely on face-to-face communication to convey information; (vi) continuous attention to software quality; (vii) simplicity is essential.

To achieve these principles, the following agile practices were used in the development: (i) short iterations (principles (i) and (iii)); (ii) refactoring (principle (vi)); (iii) emphasis on simplicity (principle (vii)); (iv) product and iteration backlogs (principles (i), (ii)); (v) Scrumboards (principles (i), (ii)); (vi) continuous reviews, integration and testing (principle (vi)). Due to the lack of direct end-user contacts, it was not possible to achieve principle (iv).

The following MDD practices were used: (i) metamodelling; (ii) transformations; (iii) executable modelling. An adapted Scrum process [22] was followed, with executable models being used in place of code. Figure 5 shows our agile MDD process – the UML to C translator development is an example of tool development (RHS of Figure 5).

Within each iteration, phases of requirements analysis, specification, implementation and testing were applied to each task, and the following process was generally followed:

1. Exploratory prototyping and research to assess the feasibility and semantic validity of possible C programming approaches to express UML and OCL elements. The Visual Studio (2012), gcc (2016) and lcc[1] (2016) C compilers were used.

[1] www.cs.virginia.edu/∼lcc-win32

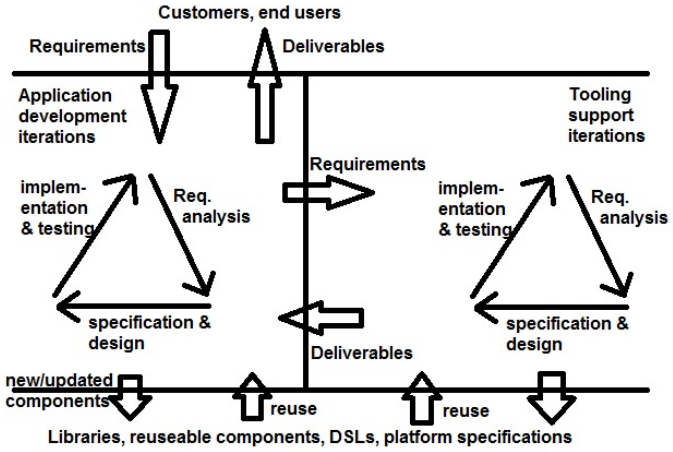

Figure 5: UML-RSDS agile MDD process: iterations

2. Informal specification in concrete grammar of the mappings from UML to C. Discussion of the 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. Test cases were manually constructed for each local functional requirement.

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). We refer to the complete translator as UML2C.

1.3 UML-RSDS restrictions

UML-RSDS omits constructs of UML and OCL which have complex semantics, such as the general *iterate* operator of OCL. It also adopts computational numeric types `int`, `long`, `double` in order to align the specification more closely to implementations (Table 1).

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

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

2 Iteration 1: Type mapping

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

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

Table 1: Differences between UML-RSDS and UML

The source language was identified as the *Type* class and its subclasses in the standard UML-RSDS class diagram metamodel (Figure 6, based upon UML 2.1 but with multiple inheritance between metaclasses removed). Not all elements of this metamodel are needed for the transformation, in particular *Constraint* is not needed, nor are the associations *linkedClass*, *ownedLiteral* or the classes *Enumeration*, *EnumerationLiteral* in this version of the transformation.

A model of Figure 6 is concretely represented as a text file with each line describing either (i) that an object exists in a particular concrete metaclass, eg:

```
c : Entity
p : Property
```

or (ii) that an attribute or 1-multiplicity role has a particular value:

```
c.name = "A"
p.type = Integer
```

or (iii) that an object is in a many-multiplicity role:

```
p : c.ownedAttribute
c : p.owner
```

The initial target language is a simplified version of the abstract syntax of C programs, sufficient to represent UML types (Figure 7). This language is open to further elaboration and extension during the development.

Using goal decomposition, the type mapping requirement F1.1 was decomposed into specific mapping requirements F1.1.1 to F1.1.4 for the translation of different categories of UML types (Table 2). The schematic concrete grammar is shown for the C target representation of each category of UML type. This mapping has the advantage of simplicity, and the T* operator directly interprets Collection(T) for sequences and (ordered) sets of objects, collections and strings. A representation for bags and an alternative (unordered) set representation have been implemented as binary search trees in ocl.h.

Requirements specification formalises these mappings as UML-RSDS rules, defining the post-conditions of a transformation $types2C$. Scenario analysis and evolutionary prototyping were used for this stage. The Auxiliary Correspondence Model pattern [12] was used to achieve the bidirectionality requirement, and the traceability requirement. A new identity attribute *typeId* : *String*

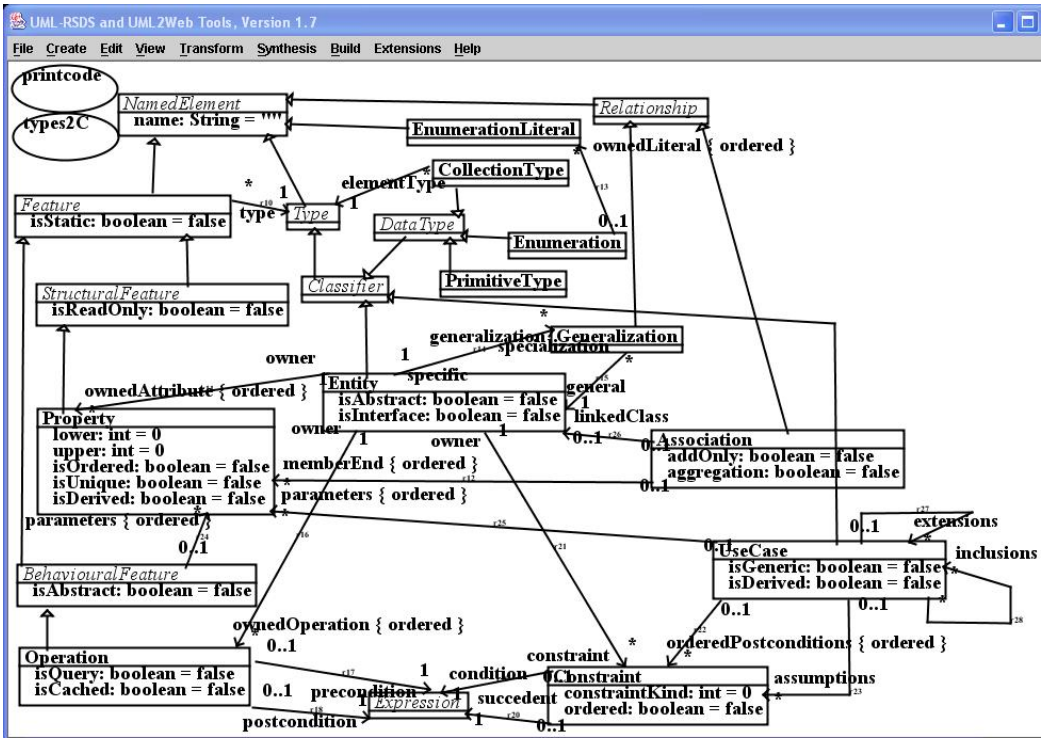

Figure 6: UML-RSDS class diagram metamodel

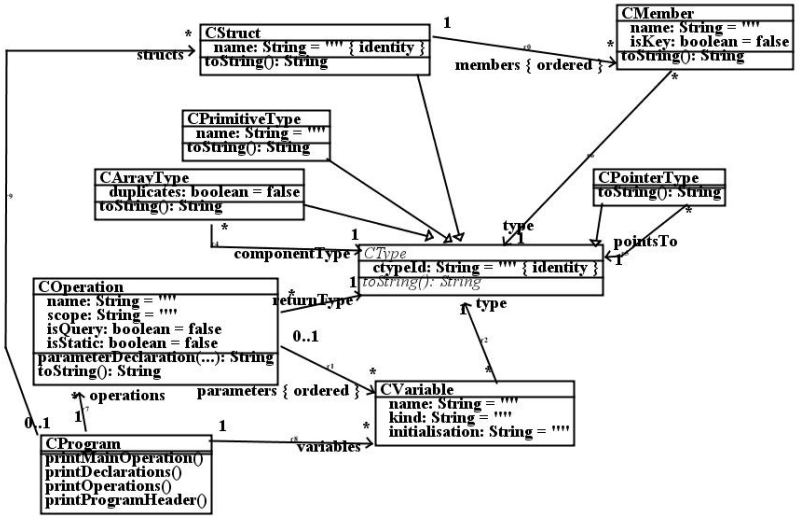

Figure 7: C language metamodel

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 2: Informal scenarios for types2C

was introduced into *Type*, and *ctypeId* into *CType*. This enables *Type* and *CType* instances to be looked-up by key value: $Type[id]$ is the type instance t with $t.typeId = id$. An instance $t : Type$ corresponds to an instance $c : CType$ if $t.typeId = c.ctypeId$.

An example of a scenario expressed in the SBVRSE notation for SysML [19] is (for F1.1.1.1):

> 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 expressed as UML-RSDS transformation rule specifications:

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

In this rule, the quantification is over objects *self* : *PrimitiveType*, and expresses that whenever the lhs of the rule is true, the rhs should be made true, ie., the relevant C types should be looked-up or created if they do not already exist.

Primitive types, entity types and collection types are successively mapped. It is assumed that collection types can only have entity types or primitive types as their element types:

> *CollectionType* ::
> $elementType : Entity$ *or* $elementType : PrimitiveType$

This ensures that, in the final two rules below, $CType[elementType.typeId]$ does exist at the point where it is looked-up: it has been created by earlier rules operating on the element type. Some example transformation rules of *types*2*C* are:

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

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] )
```

The mapping of strings and entity types is an example of the Entity Splitting MT pattern [9], in which one source instance is mapped to two or more linked target instances (t and p in the mappings of string type instances).

During requirements validation and verification of this subtransformation, the bx properties were proved. The above constraints can be inverted automatically to:

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

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. NF1 and F6 are achieved because all the transformation constraints are of type 1 (bounded loops). F2 was ensured by checking the generated syntax against the ANSI '89 C standard. F3 was ensured by constructing a semantic model $Sem_C(p)$ of the generated C programs p (in first-order set theory) and checking that this is equivalent to the corresponding mathematical model $Sem_{UML}(m)$ of UML/OCL models m [15], when p is generated from m. F4 is ensured by the use of the Auxiliary Correspondence Model pattern, as is the bx requirement F5. NF2 was checked for the uml2Ca release and found to be satisfied (Table 5).

In this iteration, the specification effort included construction of the C metamodels for use in subsequent iterations, and writing ocl.h, and the verification effort included verification of the ocl.h library. The transformation size was 10 rules and 5 operations. A further medium level non-functional requirement was added during this iteration:

- **NF5:** Usability of the transformation – it should be a simple process to invoke it from the UML-RSDS toolset interface.

The estimated effort for this iteration is shown in Table 3.

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

Table 3: Development effort for Iteration 1

3 Iteration 2: Class diagram mapping

This iteration also used a three-phase approach, to define a subtransformation *classdiagram*2*C*. The class diagram elements *Property*, *Operation*, *Entity*, *Generalization* from Figure 6 were identified as the input language. Scenario analysis and exploratory prototyping was used for requirements elicitation and evaluation.

The C representation in Figure 7 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 4.

Scenario	*UML element* e	*C representation* e'
F1.2.1	Class diagram D	C program with D's name
F1.2.2	Class/interface E	`struct E { ... };` Global variable `struct E** e_instances;` Global variable `int e_size;` `struct E* createE(void)` operation `struct E** appendE(struct E**, struct E*)` operation `struct E** insertE(struct E**, struct E*)` operation `struct E** newEList(void)` operation
F1.2.3.1	Instance-scope attribute $p : T$ (not principal identity attribute)	Member `T' p;` of the struct for p's owner, E, where `T'` represents T Operations T' $getE\_p(E'$ $self)$ and $setE\_p(E'$ $self, T'$ $px)$
F1.2.3.2	Principal identity attribute p : *String* of class E	Operations $getE\_p$, $setE\_p$, struct E* getEByPK(char* v) Key member `char* p;` of the struct for E
F1.2.4	Operation $op(p : P) : T$ of E (instance-scope)	C operation `T' op_E(E' self, P' p)` with scope = "entity"
F1.2.5	Inheritance of A by B	Member `struct A* super;` of struct B Operations $getB\_att(x)$ for inherited att invoke $getA\_att(x \rightarrow super)$. $op\_B(x, p)$ for inherited op invokes $op\_A(x \rightarrow super, p)$ unless B redefines op

Table 4: Informal scenarios for the mapping of UML class diagrams to C

In the UML-RSDS design, 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 mapped to C. The maintenance of the mutual consistency of the opposite ends ar, br of an $A - B$ association is carried out by operations $addA\_br(ax : A, bx : B)$, $removeA\_br(ax : A, bx : B)$, etc. These are synthesised in the UML-RSDS design stage and hence are translated to C by the mapping from *Operation* to *COperation* [14]. Likewise, object deletion operations *killA*, *killAbstractA* are included in the design.

Several alternative schemes were considered for representing inheritance in C: (i) struct inheritance with a superclass pointer in each subclass and explicit coding of method overriding using conditionals; (ii) embedded superclass struct instance in each subclass struct, and function pointers for each supported method; (iii) as (ii) but with vtables for function pointers; (iv) Objective-C style metaprogramming of classes and objects.

We chose option (i) as the simplest possible scheme, with the intention to move to (ii) or (iii) in future releases. Option (iv) was rejected because the resulting code is excessively complex and quite distant in style from conventional C coding.

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 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 *genCtext*:

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

Getters and setters are created by *CMember* operations such as *getterOp*, *setterOp*, *getAllOp*:

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


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


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 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"
```

Here, the Text Templates MT pattern [9] is used to combine fixed text portions with variable model-derived elements. The Auxiliary Metamodel pattern is used to introduce an auxiliary association *allMembers* from *CStruct* to *CMember*, which records all C members explicitly in or inherited by a struct.

The text generation operations are then used in the postconditions of the *genCtext* 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 & k.isStringType() ) &
key = members->select( isKey & isStringType())->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 & k.isStringType() ) &
key = members->select( isKey & isStringType() )->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. Selective generation of OCL operators is used, so that operations opE are only generated for OCL operators *op* if there is an occurrence of $\rightarrow op$ applied to a collection of E elements in the source model. This facilitates achieving NF7.

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

```
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 ) ) )
```

This constraint is in an invertible form according to [12]. Both single-valued and multi-valued attributes are mapped correctly by this scheme. However, static attributes would need to be represented as global scope C variables external to the struct definition of their entity.

F1.2.4 is specified by:

```
Operation::
    COperation->exists( op | op.name = name + "_" + owner.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.isStatic = isStatic &
        op.scope = "entity" &
        op.returnType = CType[type.typeId] )
```

For a static operation, the *self* parameter is omitted. 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* of *COperation* is introduced (the Entity Merging design pattern).

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] ) )
```

Only single inheritance is supported.

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

```
COperation::
  scope = "entity"  =>
    Operation->exists( op | op.name = name.before("_") &
             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.isStatic = isStatic &
                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] )

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 37 rules and 34 operations in this subtransformation, and 42 metamodel classes.

Testing, inspection and formal arguments were used for validation and verification. NF1 was achieved because all the transformation constraints are of type 1 (bounded loops), and recursions in helper operations are bounded by the maximum inheritance depth in the source model. F2 was ensured by checking the generated syntax against the C standard. F3 was ensured by checking that $Sem_C(p) \equiv Sem_{UML}(m)$ when p is generated from m. F4 is ensured by the use of the Auxiliary Correspondence Model pattern, as is the bx requirement F5. NF2 was checked for input UML models of different sizes and found to be satisfied (Table 5).

#classes	*#attributes per class*	*Execution time*
10	10	70ms
10	50	180ms
50	50	1.3s
50	100	4.3s
100	100	14.9s

Table 5: Execution times for uml2Ca on input models of different sizes

The estimated effort for this iteration is shown in Table 6.

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

Table 6: Development effort for Iteration 2

The result of iterations 1 and 2 is a transformation that operates on the 3 UML-RSDS metamodels (Figure 6, Figure 8 and Figure 10) as inputs, and on the C general metamodel (Figure 7) 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 MT pattern [9].

The completed prototype after iterations 1 and 2 has been implemented as a jar file *uml2Ca.jar* in the UML-RSDS version 1.7 at https://nms.kcl.ac.uk/kevin.lano/uml2web/. This reads an input

file model.txt, produced by the Save As Model option of UML-RSDS. The C code is written to a file app.h:

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

An intermediate model file cmodel.txt combining UML and C model elements is also produced, for use as the input file for uml2Cb.

4 Iteration 3: Expression mapping

In this iteration, the detailed requirements for mapping OCL expressions to C were identified, and this subtransformation, *expressions*2*C*, was specified and tested. Due to the large size of this transformation, it was not possible to complete it within 1 month.

Figure 8 shows the UML-RSDS OCL metamodel, which is the source language for the subtransformation. Figure 9 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 in the C library file `ocl.h`. An important semantic issue is that a NULL collection must be treated as equal to an empty collection in the generated code, to accord with the OCL semantics. 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 and let expression variables x in expressions $s \rightarrow forAll(x \mid P)$, and *let* $x : T = e$ *in* $e1$, etc. A further subclass, *ConditionalExpression*, is used to represent OCL conditional expressions. A $* - *$ association *context* from *Expression* to *Entity* is used to record the context(s) of use of the expression. Downcast expressions (where an object is used implicitly as an instance of a subclass of the context) are not supported by this translator.

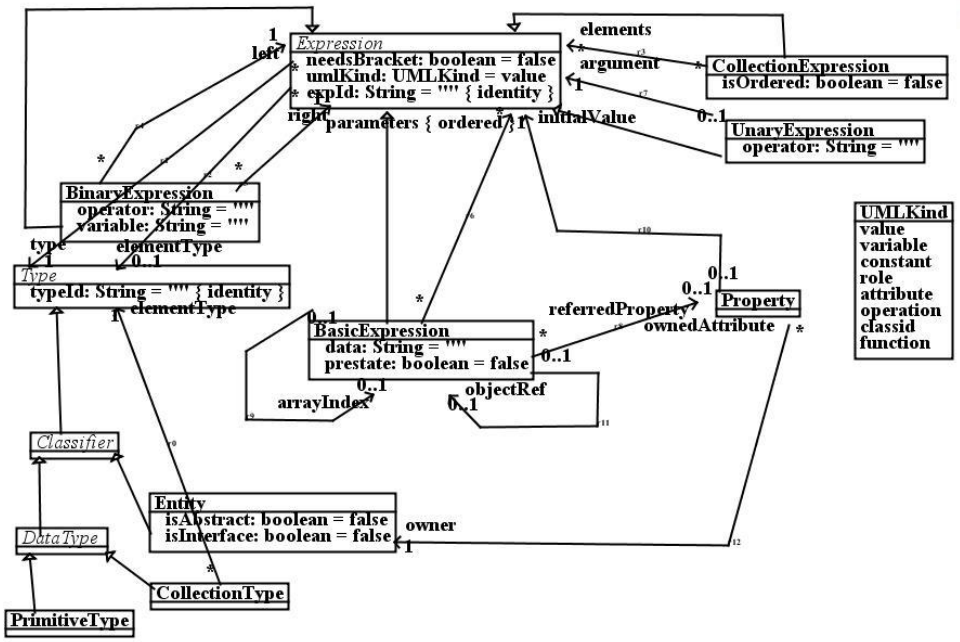

Figure 8: UML-RSDS OCL metamodel

The UML/OCL expressions were grouped into four categories: (i) mapping of basic expressions; (ii) mapping of logical expressions; (iii) mapping of comparitor, numeric and string expressions; (iv) mapping of collection expressions.

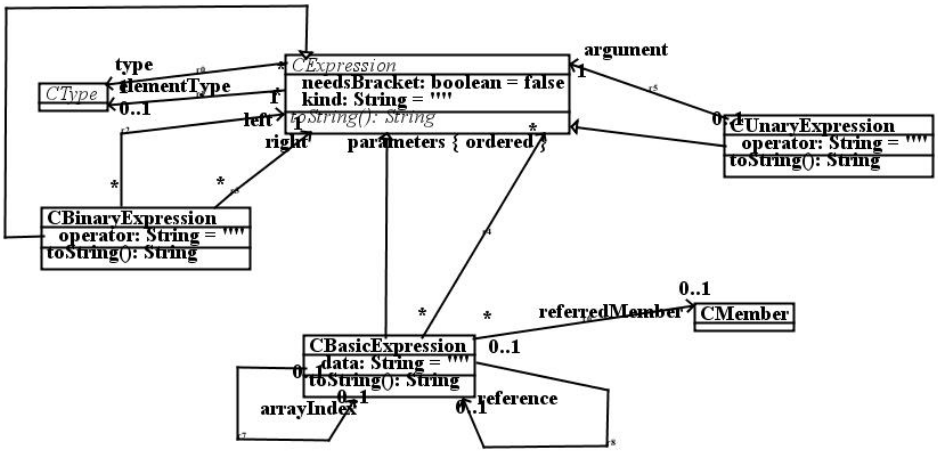

Figure 9: C expression metamodel

4.1 Basic expressions

OCL basic expressions of OCL map to C expressions (requirement F1.3.1) according to Table 7.

Not included are $objs.r$ for collection-valued r and $objs$, or operation applications $objs.op(pars)$ on collections: these can be expressed instead using *forAll* or *collect*.

4.2 Logical expressions

Table 8 shows the mapping of logical expressions and conditional/let expressions to C. These mappings are grouped together as requirement F1.3.2.

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

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

4.3 Comparator, numeric and string expressions

Table 9 lists the comparator operators and their mappings to C. These mappings are grouped as requirement F1.3.3.

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 10. Three OCL operators: ceil, round, floor, take a double value and return an int, in contrast to the corresponding C functions. The interpretations of other mathematical functions use standard C library functions from `<math.h>`. The mappings are grouped as requirement F1.3.4.

String operators are shown in Table 11. 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.

OCL expression e	*C representation e′*
self	*self* as an operation parameter
super	*self→super*
Variable v	v
Data feature f of context E with no objectRef	$getE\_f(self)$
Data feature f of E instance ex	$getE\_f(ex')$
Operation call $op(e1, ..., en)$ or $obj.op(e1, ..., en)$ of instance scope op of E	op_E((struct E*) self, e1', ..., en') op_E((struct E*) obj', e1', ..., en')
Call $op(e1, ..., en)$ of static entity scope op	op_E(e1', ..., en')
Call $op(e1, ..., en)$ of application scope op	op(e1', ..., en')
Instance-scope attribute f of E collection exs	$getAllE\_f(exs')$ (duplicate values preserved)
Single-valued role $r : F$ of E collection exs	$getAllE\_r(exs')$ defined by $(struct\ F**)\ collectE(exs', getE\_r)$
$E[v]$ v single-valued	getEByPK(v')
$E[vs]$ vs collection-valued	getEByPKs(vs')
$E.allInstances$ entity type E	$e\_instances$
$x.oclIsUndefined()$	$(x' == NULL)$
$x.oclAsType(T)$	$(T')\ x'$
value of enumerated type, numeric or string value	*value*
boolean true, false	TRUE, FALSE

Table 7: Mapping scenarios for basic expressions

OCL expression e	*C expression e′*
`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)`
`if e then e1 else e2 endif`	`(e')?(e1'):(e2')`
`let v : T = e in e1` E1 of type T1	`let_i(e')` where `T1' let_i(T' v) { return e1'; }` is defined as a new function

Table 8: Mapping scenarios for logical expressions

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 9: Mapping scenarios for comparator 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 10: Mapping scenarios for numeric expressions

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\n", e') for String-valued e, displayint(e') defined as printf("%d\n", e') for integer e, similarly for long, double, boolean
`s1 - s2`	subtractString(s1', s2')
`e->isInteger()`	`isInteger(e')`
`e->isReal()`	`isReal(e')`
`e->isLong()`	`isLong(e')`
`e->toInteger()`	`atoi(e')`
`e->toReal()`	`atof(e')`
`e->toLong()`	`atol(e')`

Table 11: Mapping scenarios for string expressions

4.4 Collection expressions

Table 12 shows the values and operators that apply to sets and sequences, and their C translations. 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 compareTo_String (based on *strcmp*) as the comparison function: treesort((void**) x', compareTo_String). Otherwise, a specifier must include a suitable $compareTo(other : E) : int$ operation in E. For each *sortedBy* expression, a new function *compare* is defined, if e is of a numeric or boolean type, $compare(const\ void*\ self,\ const\ void*\ other)$ returns e evaluated for $(struct\ E*)\ other$ subtracted from e evaluated for $(struct\ E*)\ self$. For String-valued e, *strcmp* is used, and for objects, the appropriate $compareTo\_F$ operation.

A common form of OCL expression is the evaluation of a reduce operation (min, max, sum, prd) applied to the result of a collect, eg.:

$$s \rightarrow collect(e) \rightarrow sum()$$

where e is double-valued. This is mapped to:

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

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

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 comparator function *comp*. The algorithm uses an underlying data structure of binary search trees, which can also be used to support the definition of maps (for caching and model input) and for Bag and unordered Set collection types. These are defined in ocl.h.

For sorting of collections of instances of entity type E, the entity E must have a compareTo(other : E) : int operation defined, this will become a function int compareTo_E(struct E* self, struct E* other) in C.

Table 13 shows the translation of select, collect and iterate operators, where s is a collection of E objects. These mappings are grouped as requirement F1.3.7.

Due to the self-recursive structure of OCL expressions and UML activities, the expression and activity mapping were specified using recursive operations to map an expression or activity based upon the mappings of its subordinate parts.

An operation

```
mapExpression() : CExpression
```

is defined in each Expression subclass. This is an example of the Rule Inheritance pattern [16].

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:
```

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->at(i)`	(x')[i'-1]
`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', compareTo_E) x of entity element type E (char**) treesort((void**) x', compareTo_String) 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)`	intSum(a',b',fe), longSum(a',b',fe), doubleSum(a',b',fe) fe computes e'(x')
`Integer.Prd(a,b,x,e)`	intPrd(a',b',fe), longPrd(a',b',fe), doublePrd(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 12: Scenarios for the translation of collection operators

UML expression e	*C translation e'*
`s->select(P)`	selectE(s', fP) where E is the entity element type of s, fP evaluates P': unsigned char fP(struct E* self) { return P'; }
`s->select( x | P )`	as above, fP is: unsigned char fP(struct E* x) { return P'; }
`s->reject(P)`	rejectE(s', fP), fP as for select
`s->reject( x | P )`	as above
`s->collect(e)` e of type et	(et'*) collectE(s', fe) fe evaluates e'
`s->collect( x | e )`	as above
`s->selectMaximals(e)`	–
`s->selectMinimals(e)`	–
`s->iterate(x; result : T = e | e1)` e1 of type et	iterateE_et(s',e',fe1) fe1 implements $e1$

Table 13: Scenarios for the mapping of selection, collection and iterate expressions

```
  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, for example:

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

```
      result = mapClassExpression(ob,aind,pars)) &
  (umlKind = function  =>
      result = mapFunctionExpression(ob,aind,pars))
```

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 [15]. The operation precondition asserts that the parameters correspond to the sub-parts of the basic expression. The *kind* attribute of *CExpression* records the origin of the C expression. This enables an inverse operation to be defined, eg.:

```
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 [9], 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.

Some specific cases for mapping different forms of basic expression are as follows:

```
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.reference = obs &
      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:
  (context.size = 0 & objectRef.size = 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "attribute" &
            c.data = data &
```

```
            c.isStatic = true &
            c.type = CType[type.typeId] &
            c.elementType = CType[elementType.typeId] &
            c.arrayIndex = aind &
            result = c ) )  &
  (context.size > 0 & objectRef.size = 0  =>
      CBasicExpression->exists( c | c.cexpId = expId & c.kind = "attribute" &
            c.data = "get" + context.any.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.cexpId = expId + "_self" &
                  s.type = CType[context.any.typeId] &
                  s.elementType = s.type &
            c.parameters = Sequence{ s } & 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] &
            c.parameters = obs &
            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] &
            c.parameters = obs &
            c.arrayIndex = aind &
            result = c ) )
```

Note that *context* should normally be set for attributes and roles, so that the context entity using the feature can be accessed (this entity may be a subclass of the owner of the feature). The mapping operations can be inverted clause-by-clause. This is possible since the target expression encodes all necessary information to derive the source expression. For example, an assignment $c.parameters = Sequence\{s\} \frown pars$ inverts to $pars = c.parameters.tail \ \& \ s = c.parameters.first$.

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 *objectRef* because this is of 0..1 multiplicity. *name* is needed for the copies of *Type* and *CType* in the uml2Cb metamodel. The generation of model.txt by the UML-RSDS tools needed to be adjusted in several cases to ensure that appropriate information was available for UML2C.

The correspondence of OCL and C operators is given in Table 14.

Various convenience operations were introduced during design, such as:

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

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

Table 14: Operator correspondence between UML and C

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

These are used to create C representations of binary and unary expressions as operation calls. They are an application of the MT pattern Factor out Duplicated Expression Evaluations [9].

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, "collect" + left.elementType.name, le,
              CExpression.defineCOpRefCast(CProgram.allInstances.any.defineCOp(re,
                    variable, le.elementType), "void* (*)(" + le.elementType + ")" ) )
```

These create new auxiliary 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. As noted above, there is only one parameter *variable* of the auxiliary operation.

The definition of binary expression mapping is:

```
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)) &
  (operator = "/="  =>
       result = mapInequalityExpression(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 = mapSortedByExpression(lexp,rexp)) &
  (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.needsBracket = needsBracket &
         c.type = CType[type.typeId] &
         c.elementType = CType[elementType.typeId] &
         result = c ) )
```

A string binary op is one of $\rightarrow indexOf$, $\rightarrow count$, $\rightarrow hasPrefix$, $\rightarrow hasSuffix$. An iterator operator is one of $\rightarrow forAll$, $\rightarrow exists$, $\rightarrow exists1$, $\rightarrow select$, $\rightarrow reject$.

A collection binary op is one of: $\rightarrow including$, $\rightarrow excluding$, $\rightarrow append$, $\rightarrow indexOf$, $\rightarrow count$, $\rightarrow union$, $\rightarrow intersection$, $\rightarrow at$, $\frown$, $\rightarrow isUnique$, $\rightarrow sortedBy$.

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 ) )
```

*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.

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.clength(arg)) )
```

clength(*arg*) 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'. For literal sequences the size can be directly evaluated, and for $Integer.subrange(a, b)$ it is $b' - a' + 1$.

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

During this iteration, a new functional requirement F1.2.6 to map static operations to C was added, which involved reworking the uml2Ca transformation. In particular, the Expression and CExpression and COperation classes all need an additional boolean attribute isStatic. This effort is included in the budget of iteration 3.

F1.2.6	Operation $op(p : P) : T$ of E (static)	C operation `T' op_E(P' p)` with scope = "entity"

Table 15: Additional class diagram mapping requirement F1.2.6

There are 92 operations and 33 transformation rules for this use case. Testing and inspection were used for validation and verification. NF1 was achieved because all the transformation constraints are of type 1 (bounded loops), and recursions in operations are bounded by the maximum depth of input model expressions as abstract syntax trees. F2 was ensured by checking the generated syntax against the C standard. F3 was ensured by checking $Sem_C(p) \equiv Sem_{UML}(m)$ when p is generated from m. F4 is ensured by the use of the Auxiliary Correspondence Model and Recursive Descent patterns, as is the bx requirement F5. NF2 was checked for the uml2Cb release and found to be satisfied (Table 19).

The estimated effort for this iteration is shown in Table 16.

Considerable effort was spent on debugging, with most of the problems traced to the input model data generated from UML models, or (less often) to errors in the preceding uml2Ca transformation. Errors in the uml2Cb transformation itself were usually quickly traced and corrected, and were for the most part minor, eg., incorrectly treating a 0..1-multiplicity role as if it were 1-multiplicity, etc.

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

Table 16: Development effort for Iteration 3

5 Iteration 4: Activities mapping

In this iteration, the mapping of UML-RSDS activities to C statements was specified by a subtransformation *statements*2*C*. Figure 10 shows the input activity language. Additionally, *CreationStatement*s have a type and elementType. UML-RSDS activities correspond closely to the statements of programming languages, including C. Figure 11 shows the metamodel for C statements. *DeclarationStatement*s have a type and elementType, and for *IfStatement*, *elsePart* is optional, as for UML-RSDS *ConditionalStatement* activities. Table 17 shows the main cases of the mapping of UML activities to C statements. Calls *objs*.*op*(*pars*) of instance operations using collections *objs* of objects are converted to bounded loop statements during design generation so they do not need to be considered here. OCL collections and strings are values, and hence should be copied when assigned (scenario F1.4.2).

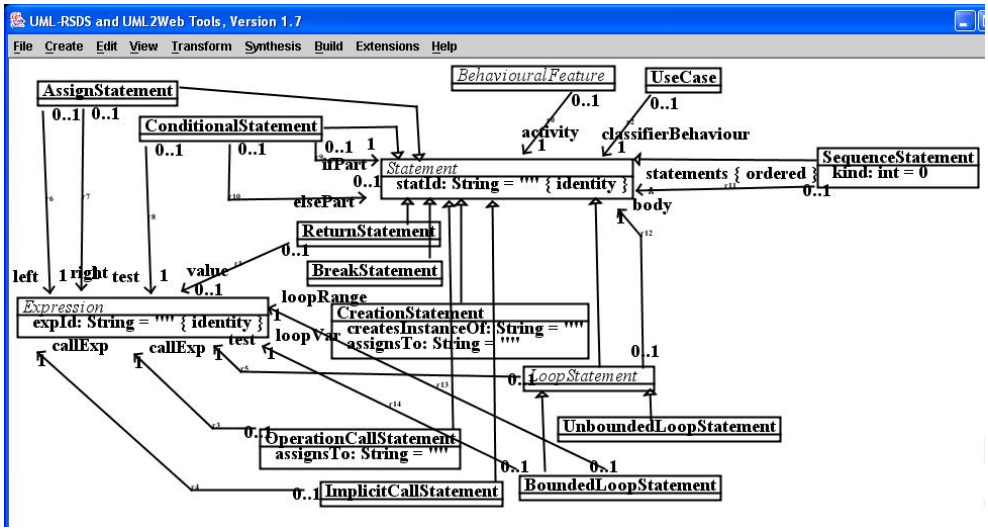

Figure 10: UML-RSDS activity metamodel

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 *expressions*2*C* 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::
```

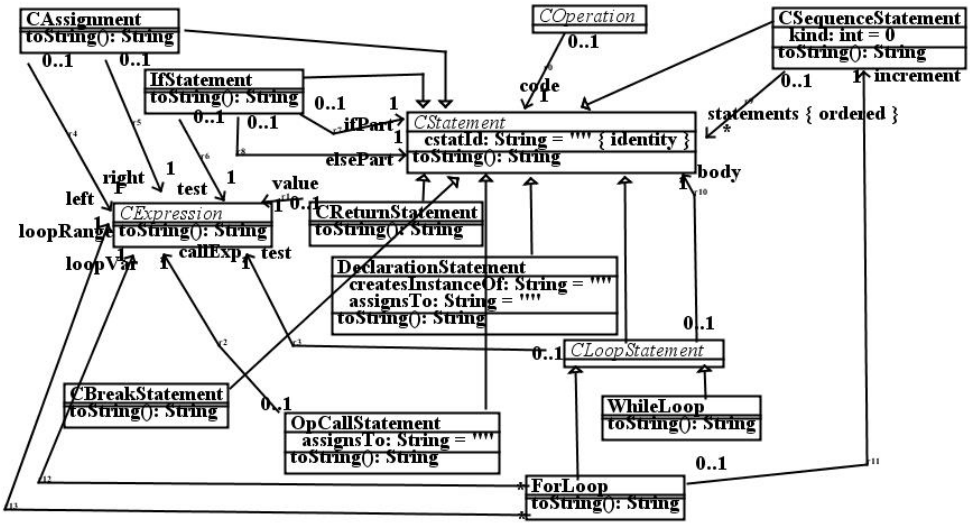

Figure 11: C statement metamodel

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` Assign statement `v : T := e` Assign statement `obj.f := e` obj of class E	v' = e'; strcpy(v', e'); for String variable v T' v' = e'; setE_f(obj', e')
F1.4.3	Sequence statement `st1 ; ... ; stn`	st1' ... stn'

F1.4.4	Conditional statement
	`if e then st1 else st2`
	Conditional statement `if e then st1`
		
		if (e') { st1' } else { st2' }
		if (e') { st1' }

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

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

F1.4.8	Unbounded loop `while e do st`	while (e') { st' }
F1.4.9	Operation call `ex.op(pars)` on single object *ex* of *E*	op_E((struct E*) ex',pars')

Table 17: Scenarios for mapping of statements to C

```
mapStatement() : CStatement
post:
  assign = left.mapAssignment(self, right.mapExpression()) &
  (type.size = 0  => result = assign)  &
  (type.size > 0 =>
    assign.type = CType[type.typeId] & result = assign)

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 )


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 : Set(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.

As with the mappings of expressions, these mappings can be directly inverted, for example:

```
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 : Set(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 )
```

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 *toString* operations, eg.:

```
DeclarationStatement::
query toString() : String
post:
  (createsInstanceOf = "String"  =>
        result = type + " " + assignsTo + " = \"\"") &
  (type : CPrimitiveType  =>
        result = type + " " + assignsTo + " = 0;") &
  (true  =>
        result = type + " " + assignsTo + " = NULL;")


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 + " }")
```

It was identified that the *elsePart* association ends should have 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 28 operations and 4 rules for this iteration.

The estimated effort for this iteration is shown in Table 18.

6 Iteration 5: Use case mapping

In this iteration, the mapping of UML use cases was specified by a transformation $usecases2C$. It was agreed during evaluation and negotiation that use case attributes and operations would be excluded from the mapping, since these are non-essential.

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

Table 18: Development effort for Iteration 4

F1.5.1: A use case *uc* is mapped to a static 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. Use case extension is not represented, because this is considered as a purely specification-level composition mechanism.

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.isStatic = true &
        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()
```

NF1 was achieved because all the transformation constraints are of type 1 (bounded loops). F2 was ensured by checking the generated syntax against the C standard. F3 was ensured by checking the equivalence of the semantics of the UML and C models. F4 is ensured by the use of the Auxiliary Correspondence Model and Recursive Descent patterns, as is the bx requirement F5. NF2 was checked for the uml2Cb release and found to be satisfied (Table 19).

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

Table 19: Execution times for uml2Cb

The result of iterations 3, 4 and 5 is a transformation that operates on the 3 UML-RSDS metamodels (a subset of Figure 6, and Figures 8 and 10) as inputs, and on a subset of the C general metamodel (Figure 7) and on Figures 9, 11 as outputs. Uml2Cb operates on 49 metamodel classes. The completed prototype for iterations 3, 4, 5 is uml2Cb.jar. It reads an intermediate model file cmodel.txt produced by uml2Ca.jar and writes the generated C program to a file app.c. NF5 is achieved because uml2Ca and uml2Cb can be invoked from the main UML-RSDS interface.

There are 6 rules in this iteration. Table 20 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 20: Development effort for Iteration 5

7 Evaluation

In this section we evaluate the outcomes of the development, the effectiveness of UML-RSDS for the development, and the agile MDD approach that we have used. We compare the approach to the agile traditional coding approach used for other UML-RSDS code generators.

7.1 Comparison with requirements

Table 21 compares the functional and non-functional requirements and the actual achieved results. In some cases it is possible to prove by the construction of the transformation that some properties hold (termination and confluence). For syntactic and semantic correctness we can give rigorous arguments based on considering each mapping rule and checking that it produces valid C with the same semantics as its input. For some aspects, such as numeric computations, semantic correctness is only relative to the same definitions of numeric types being used in the input UML and output C: the specifier needs to use in their specification the same data type sizes (eg., 16 bit int type) as the target code platform. For dynamic memory allocation, we assume that *malloc* and *calloc* always succeed. Select and other iterator expressions are restricted to depend on only one variable. Only collections containing strings or entity instances can be explicitly constructed.

In order to test NF6 and NF7 we carried out a number of tests on different UML specifications. All tests were carried out on a standard Windows 7 laptop with Intel i3 2.53GHz processor using 25% of processing capacity. The first test was a small-scale application involving a fixed-point computation of the maximum-value node in a graph of nodes. This application has one entity type A, with an attribute $x : int$ and a self-association $neighbours : A \rightarrow Sequence(A)$. There is a use case *maxnode* with the postcondition

Requirement	*Priority*	*Achievement*
NF1: Termination	High	Proved
NF10: Development time	High	Achieved
F2: Syntactic correctness	High	Rigorous argument and testing
F3: Semantic preservation	High	Rigorous argument and testing
F4: Traceability	High	Achieved
F5: Bidirectionality	Medium	Partly achieved
NF2: Transformation efficiency	Medium	Achieved
NF3: Transformation modularity	Medium	Achieved
NF5: Usability	Medium	Achieved
NF6: Efficient code	Medium	Partly achieved
NF7: Compact code	Medium	Partly achieved
F6: Confluence	Low	Proved
NF4: Flexibility	Low	Not achieved

Table 21: Achievement of requirements

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

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

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

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

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

unsigned char maxnode1search(void)
{ int ind_boundedloopstatement_80 = 0;
  int size_boundedloopstatement_80 = oclSize((void**) a_instances);
  for ( ; ind_boundedloopstatement_80 < size_boundedloopstatement_80;
            ind_boundedloopstatement_80++)
  { struct A* ax = (a_instances)[ind_boundedloopstatement_80];
    int ind_boundedloopstatement_85 = 0;
    int size_boundedloopstatement_85 = oclSize((void**) getA_neighbours(ax));

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

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

```
  while (maxnode1_running)
  {  maxnode1_running = maxnode1search(); }
}
```

Table 22 compares the code size (for the complete applications, including OCL library code) and the efficiency of the C code with the Java code produced by the UML-RSDS Java code generator. The execution time figures are for the lcc compiler, which are somewhat lower than the figures for Visual Studio. These show that code size is halved by using C, and that efficiency is improved.

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

Table 22: Generated C code versus Java code, case 1

In a second case, the efficiency test from [8] was used. This computes prime numbers in a given range using a double iteration. Table 23 compares the generated code in Java, C, C# and C++ on this case.

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

Table 23: Generated C code versus Java, C#, C++ code, case 2

The main causes of inefficiency in the C code are (i) repeated linear traversals of collections to calculate the sizes of collections; (ii) the cost of allocating and reallocating large contiguous blocks of memory for array-based collections. An alternative array collection representation could use the first element of an array to store the collection length. This also has the advantage that C and OCL indexing of collections would coincide. However it would hinder the compatibility of the generated code with conventional C code. For (unordered) sets and bags non-contiguous memory blocks can be used, and this reduces the memory allocation costs. In cases where a computed collection value e is immediately assigned to a collection-valued feature: $obj.r = e$, there is no need to allocate new memory for e, instead $obj.r$ can be updated in-place. In cases of *sum*, *prd* applied to the result of a *collect*, an *Integer.Sum* or *Integer.Prd* expression could be used instead: $objs{\rightarrow}collect(e){\rightarrow}sum()$ for sequence *objs* is the same as

$$Integer.Sum(1, objs.size, i, e(objs[i]))$$

This avoids the allocation of memory space for the *collect* result.

7.2 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 24 shows the approximate effort in person-months expended for each of

	Java 4	Java 6	Java 7	C#	C++
Req. Analysis	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 24: Development effort for previous code generators (person months)

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

Summarising Tables 3, 6, 16, 18, 20, we obtain an overall estimate for the C code generator in Table 25.

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

Table 25: Overall development effort for C code generator

Stage	*C++ generator*	*UML2C*
Req. Analysis/Specification	6	4.5
Design/Coding	6	1
Testing	2	0.5
Maintenance	3	0
Total	17	6

Table 26: Development effort of C++ and C code generators

This amounts to 4.5 person months for requirements analysis/specification activities, compared to 6 months for the manually-developed C++ generator. 49 days were spent on implementation and testing, compared to 8 months for the C++ generator (Table 26). 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. Not only is the UML-RSDS specification 4 times shorter than the Java code, but the latter is scattered over multiple source files (eg., Attribute.java, Association.java, Entity.java, etc), making debugging and maintenance more complex compared to the C translator, which is defined in 2 specification files. The C++ generator does not construct a C++ language model, instead language mapping and text production are mixed together, resulting in complex and duplicated processing. In total, the core code of the UML-RSDS tools is 90,500 lines of Java code, of which approximately 20% (18,100 lines) is the C++ code generator. In contrast the UML2C specification is 2,200 (uml2Ca) and 2,700 (uml2Cb) lines, in total 4,900 lines. The OCL specification of UML2C is highly declarative

and corresponds directly to the informal requirements, hence it is easier to understand and modify compared to a programming language implementation. In iterations 3 and 4 the specification style is less purely declarative than in iterations 1, 2 and 5, but instead is in a functional programming style. It was found that this was also more concise and easier to understand and change than the imperative Java coding of the C++ code generator transformation.

Whilst UML2C is explicitly divided into 5 main stages, each subdivided into model to model and model to text modules, the C++ generator has a monolithic structure. Only two design patterns (Iterator and Visitor) are used in the C++ generator, whilst 13 are used to organise UML2C (Section 7.4).

Table 27 summarises the differences in software quality measures between the C++ generator and UML2C.

Measure	*C++ generator*	*UML2C*
Size (LOC)	18,100	4,900
Abstraction level	Low (code)	High (specification)
Software architecture	Partial	Detailed
Modularity	Low (1 module)	High (10 modules)
Cohesion	Low	High
Coupling	Low	Low
Design patterns	2	13

Table 27: Software quality measures of C++ and C code generators

We can also compare the level of design flaws or *technical debt* in the C++ translator and in UML2C. For the C++ translator the data has been calculated using the PMD tool (https://pmd.github.io). For UML2C we have used the following measures of technical debt:

ETS: Excessive transformation size (total complexity > 1000, where complexity is the sum of the number of operator and identifier occurrences)

ENR: Excessive number of rules ($nrules > 10$)

ENO: Excessive number of helpers/operations ($nops > 10$)

ERS: Excessive rule size (complexity > 100)

EHS: Excessive helper size (complexity > 100)

EPL: Excessive parameter list (for transformation, rules, and helpers): > 10 parameters including auxiliary rule variables

CC: Cyclomatic complexity (of rule logic or of procedural code) (> 10).

Measures of fan-out, duplicate code and coupling between operations are not measured by PMD, so are omitted. There are substantial numbers of code clones and inter-operation dependencies in the C++ code generator Java code, however. *ETS* is taken to be the same as Excessive Class Size in PMD. The threshold values used in PMD are: *ETS* 1000 LOC; *EHS* 100 LOC; *EPL* 10 parameters; *CC* 10; *ENO* 10 per class. Table 28 compares the TD figures for UML2C and the C++ generator. For the latter, the figures for EHS/ERS, CC and EPL are produced by dividing the total numbers of faulty operations in UML-RSDS by 5. It can be seen that the figures for UML2C are generally lower than for the C++ translator.

7.3 Effectiveness of agile MDD

The benefits of effort reduction described above are mainly due to the use of specifications instead of code, and to the use of executable modelling. The use of a systematic requirements engineering process also helped to capture and make explicit all requirements, avoiding ambiguity over the

Transformation	*ETS*	*EHS + ERS*	*CC*	*ENO + ENR*	*EPL*
C++ generator	5 (classes)	16.6	110.6	28 (classes)	0.4
UML2C	2 (transformations)	13	62	4 (transformations)	0

Table 28: C++ translator versus UML2C technical debt

development tasks. The decomposition of the transformation into semi-independent phases formed a natural basis for the definition of the top-level work items in the product backlog, and this in turn led to the definition of iterations. The use of short iterations and task backlogs enabled incremental development of the translator and helped organise the development. Partial specifications were defined for each separate iteration of the system, and incrementally refined. Refactoring was extensively used to improve the specification, in particular the removal of duplicated functionality. The agile emphasis on simplicity helped to reduce the specification complexity: in particular only class diagrams and OCL constraints are used to define the translator, without activities or other UML models. Overall, we consider the combination of agile and MDD to be effective in improving system quality and reducing development time. Further techniques, such as model-based testing and pair modelling, will be investigated in future work. The work on this transformation has helped to identify improvements in the existing UML-RSDS code generator transformations.

7.4 Transformation design patterns

We found that several MT design patterns from [9], [16] were useful in structuring and simplifying the code generator:

- *Phased Construction*: Non-collection types are mapped before any of their containing collection types, in the types mapping.
- *Auxiliary Correspondence Model*: New identity attributes are added to source and target entities, to ensure bx and traceability properties, for the type, expression and activity mappings.
- *Auxiliary Metamodel*: A new auxiliary association, *allMembers*, is added to store the inherited and direct members of a C struct.
- *Object Indexing*: Previously-created target model elements are looked up by their ids, to facilitate use of the Phased Construction, Entity Merging and Transformation Chain patterns.
- *Entity Splitting*: UML expressions and other elements are sometimes mapped to several linked C elements. To ensure traceability and bx properties, the ids of the different target C elements are each derived by 1-1 mappings from the source element id.
- *Entity Merging*: The representation of UML entity operations as C operations is achieved by merging information from the class model (iteration 2) and the activities (iteration 5), using an introduced primary key for COperation.
- *Recursive Descent*: The mapping of expressions and activities are naturally expressed using this pattern, because of their self-recursive structures.
- *Rule Inheritance*: In some cases (such as the definition of mapExpression, mapStatement, clength, etc), a generalised mapping rule is expressed in a superclass definition, and specialisations defined in certain subclasses.
- *Transformation Chain*: The implementation is organised as the sequential composition of uml2Ca and uml2Cb. This allows each to operate on only subsets of the complete UML and C metamodels.

- *Implicit Copy*: The executable implementations of UML-RSDS transformations automatically copy any unmodified input data to the output model, so that there is no need to write copying rules for entities unaffected by the transformation. Uml2Ca copies UML expression and activity data unchanged, for example.
- *Factor Code Generation into Model-to-model and Model-to-code*: The UML model is mapped to a C language model and then code text is printed from this.
- *Text Templates*: The printing of code uses fixed text templates with variable elements specified by C model data.
- *Factor out Duplicated Expression Evaluations*: Helper operations are introduced, such as createCBinOpCall, to factor out repeated processing. Let variables are also used in certain operations (eg., getPKOp).

The patterns are mainly rule modularisation or optimisation patterns. Transformation Chain and Factor Code Generation are architectural patterns. The invertible form of Recursive Descent used here is a novel MT design pattern discovered because of this case study.

We found that all of these patterns had positive contributions in the development, as shown in Table 29.

Pattern	*Benefits*
Phased Constr.	Helps achieve NF3
Aux. Corr. Model	Helps achieve F3, F4, F5
Aux. Metamodel	Helps achieve NF3
Object Indexing	Helps achieve NF2
Entity Splitting	Helps achieve NF3, F4
Entity Merging	Helps achieve NF3
Recursive Descent	Helps achieve F5, NF3
Rule Inheritance	Helps achieve NF3, simplicity
Transformation Chain	Helps achieve NF3, potentially NF4
Implicit Copy	Simplicity, NF2
Factor Code Gen.	Helps achieve F2, F3, NF3, potentially NF4
Text Templates	Helps achieve simplicity, F2, NF3, potentially NF4
Factor out Dup. Expressions	Helps NF2, NF3, simplicity

Table 29: Pattern Benefits

However there were some disadvantages: in particular the use of Transformation Chain required additional effort to ensure that all information needed by uml2Cb was correctly produced by uml2Ca. The detection and correction of errors in uml2Cb caused by incorrect UML model data was more difficult because this data was accessed via uml2Ca. The organisation and division of metamodels between these transformations also required effort, and deployment was made more complex. However, we consider that overall the benefits of modularity and cohesion gained by this architecture outweigh the costs.

8 Related work

Systematic software engineering of model transformations is only practised in a minority of MT developments, according to surveys of MT development [2, 11]. The emphasis in MT developments has been on implementation, with less attention paid to requirements engineering or the overall software quality of the transformation. One example of a detailed development process is the migration case study of [24], which describes the techniques used in this industrial project. Details of the development process for an industrial transformation project are also provided in [18]. In this

paper we have given a detailed description of the development process and engineering techniques used, together with evaluations of their effectiveness.

Code generation from UML to ANSI C is also an unusual topic, with only one recent publication describing such a translator [5]. This code generator is described in a high-level manner, and it is not clear how OCL expressions or UML activities are mapped to C using the transformation. In contrast, we have provided explicit mappings for all elements of a substantial subset of UML, including a large subset of OCL. Formal specification approaches for MT are described in [27] and [4]. These do not appear to have been applied to large scale transformations.

Agile MDD has been applied in the areas of Web application development, finance, vehicle control systems and telecom systems [1]. In general, benefits have been obtained from this combination, particularly in improving development agility compared to a pure MDD approach. An obstacle to the use of MDD with agile is the cost of model management and the maintenance of consistency between linked models. We have addressed this problem by using a single specification model which combines a class diagram, OCL and use cases. We found that this reduced the amount of work needed to change a model in response to changing requirements/refactoring or error corrections.

9 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 [17]. For this subset, dynamic memory allocation is not permitted, so for each class, a maximum bound must be provided for the number of objects of the class. 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 would be represented as ints indexing into these arrays, and collections represented as fixed-size arrays. Our code generator already satisfies most of the code structuring restrictions of MISRA C, and union data structures and other cases of overlapping memory usage are already excluded. Bitfields are not used, nor are static elements. Recursive functions are not permitted by MISRA C (rule 16.2), so an alternative means of sorting (not treesort or qsort) must be used. Declarations of all operations must be provided in separate header files (rule 8.1). The following rules of MISRA-C are satisfied by our code generator: 1.1; 1.2; 1.3; 2.1; 2.2; 2.3; 2.4; 3.5; 4.1; 4.2; 5.1*; 5.2*; 5.3; 5.4; 5.5; 6.1; 6.4; 6.5; 7.1; 8.2; 8.3; 8.4; 8.6; 8.7; 8.8; 8.9; 9.1; 9.2; 9.3; 10.3; 10.4; 10.5; 11.2; 11.3; 11.5; 12.1*; 12.2*; 12.3; 12.4*; 12.5*; 12.6; 12.7; 12.8; 12.9; 12.12; 12.13; 13.1; 13.2; 13.3*; 13.4; 13.5; 13.6; 14.2; 14.3; 14.4; 14.5; 14.6*; 14.7*; 14.8; 14.9; 15.0; 15.1; 15.2; 15.3; 15.4; 15.5; 16.1; 16.3; 16.4; 16.6; 16.8; 17.1; 17.2; 17.3; 17.5; 17.6; 18.1; 18.2; 18.4; 19.1; 19.2; 19.3; 19.4; 19.5; 19.6; 19.7; 19.8; 19.9; 19.10; 19.11; 19.12; 19.13; 19.14; 19.16; 19.17; 20.1; 20.2; 20.5; 20.6; 20.7; 20.8; 20.11; 20.12. The rules marked with an asterisk are only satisfied if corresponding restrictions are observed in the UML model.

The transformation can also serve as a basis for generators into other languages, currently a VB.net code generator is being developed from UML2C. It can also be used as a template for re-engineering the UML to Java code generators as UML-RSDS specifications. The transformation is an example of a UML-RSDS plugin, that is, an additional tool which can process UML-RSDS model data in text form to produce specialised code or documentation. Developers may add such plugins f.jar by placing them in a /f subdirectory of the UML-RSDS tool directory. More generally, UML-RSDS can be used to define domain-specific languages (DSLs) and supporting tools in a similar manner: the abstract syntax of a DSL is specified as a UML class diagram (such as Figure 10 as an abstract syntax for activities or pseudocode statements), the concrete text syntax is given by text lines of the three forms $object : Entity$, $object.feature = value$ and $object1 : object2.feature$ as described in Section 2. This is also the serialisation syntax of the DSL. Plugins can then operate on files of such text to perform analysis, to generate other representations of DSL models (such as graphical syntax, or semantic representations) and to generate executable code or configuration files, etc. Using UML-RSDS, such plugins can be themselves written as transformations that use the DSL metamodel as their source language.

UML2C is an example of a multiple-staged code-generation strategy, which maps UML to code in a series of steps, with successive refinement of UML elements to program elements. This approach facilitates optimisation of generated code, and facilitates reuse of code generators (only one stage may need to be changed to generate in a new/variant language). The steps of staged code generation are (Figure 12):

1. *UML specification model to UML design model*: operation and use case postconditions (OCL) are refined to explicit activities. Auxiliary operations are introduced for object creation, deletion and for association management. It is possible also to introduce auxiliary data and operations to implement iterators and caching.

2. *UML design model to generic or language-specific 3GL model*: UML types are mapped to 3GL types, operations to 3GL operations, activities to statements, etc.

3. *3GL model to code*: text is printed from the 3GL model.

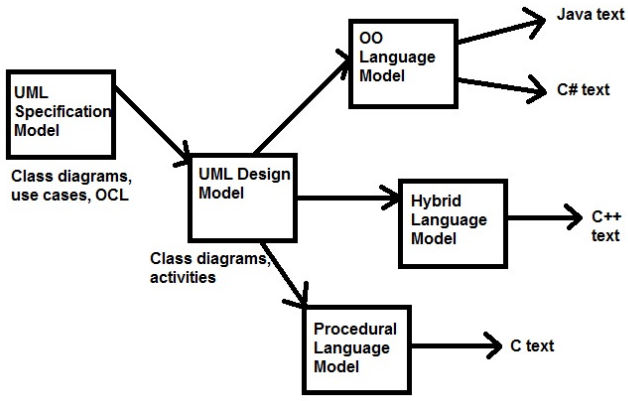

Figure 12: Staged code generation process

This provides flexibility, since the stage 1 translator does not need to change if a generator for a new target programming language is needed. A new version of stage 2 is only needed in the case of a significantly different programming language family. A new stage 3 mapping is needed for each new programming language/variant within a family. For example, the language family of statically-typed pure OO languages including all versions of Java and C# can be represented by a single program metamodel. Only textual differences arise between these languages (eg., what keywords and type names are used), and hence only different stage 3 mappings are needed.

Hybrid OO languages such as C++ have significantly different program structures and language rules, and hence a separate language model is needed, together with stage 2 and stage 3 mappings for this.

Procedural languages such as C and VB again require a distinct language model, as will dynamically-typed languages (such as Python, Ruby, etc) and specialised languages (Matlab, R).

There is a tradeoff between the degree of generality of the code generation process and the efficiency of the resulting code. Placing more code generation processing in stage 1 reduces the processing needed in later stages – but this would result in inefficient general solutions for some UML/OCL elements (eg., implementing a merge-sort algorithm in the UML design model, for *sortedBy*). Likewise many OCL library operators (*union*, *intersection*, etc) could be defined in general terms in the UML design, but it is more efficient to define language-specific libraries for OCL, based on program libraries (eg., Java Collections). Object creation/deletion may also need language-specific code (eg., *malloc*, *free* in C).

10 Conclusions

We have shown that a systematic MDD process including requirements engineering and agile practices can be beneficial for practical MT development. In addition, we have shown that it is feasible to use a declarative and semantically simple specification approach (using OCL without null or undefined values) to define a substantial system.

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 250 rules/operations in 5 subtransformations). By using a systematic requirements engineering and agile development process, 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. The translator has been incorporated into the UML-RSDS tools version 1.7.

References

[1] H. Alfraihi, K. Lano, *The integration of Agile development and Model driven development*, Modelsward 2017.

[2] E. Batot, H. Sahraoui, E. Syriani, P. Molins, W. Sboui, *Systematic mapping study of model transformations for concrete problems*, Modelsward 2016, pp. 176–183.

[3] J. Cuadrado, F. Jouault, J. Molina, J. Bezivin, *Deriving OCL optimisation patterns from benchmarks*, OCL 2008.

[4] A. Dieumegard, A. Toon, M. Pantel, *Model-based formal specification of a DSL library for a qualified code generator*, OCL 2012.

[5] M. Funk, A. Nysen, H. Lichter, *From UML to ANSI-C: an Eclipse-based code generation framework*, RWTH, 2007.

[6] E. Guerra, J. de Lara, D. Kolovos, R. Paige, O. Marchi dos Santos, *Engineering Model Transformations with transML*, SoSyM, vol. 12, no. 3, 2013.

[7] B. Kernighan, D. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[8] M. Kuhlmann, L. Hamann, M. Gogolla, F. Buttner, *A benchmark for OCL engine accuracy, determinateness and efficiency*, SoSyM vol. 11, 2012.

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

[10] K. Lano, H. Alfraihi, S. Yassipour-Tehrani, H. Haughton, *Improving the Application of Agile Model-based Development: Experiences from Case Studies*, ICSEA 2015.

[11] S. Yassipour-Tehrani, K. Lano, S. Zschaler, *Requirements engineering in MT development*, ICMT 2016.

[12] K. Lano, S. Yassipour-Tehrani, *Verified bidirectional transformations by construction*, VOLT '16, MODELS 2016.

[13] K. Lano, *Agile Model-based Development using UML-RSDS*, Taylor and Francis, 2016.

[14] K. Lano, S. Yassipour-Tehrani, H. Alfraihi, S. Kolahdouz-Rahimi, *Translating UML-RSDS OCL to ANSI C*, OCL 2017.

[15] K. Lano, *The UML-RSDS Manual*, https://nms.kcl.ac.uk/kevin.lano/umlrsds.pdf, 2017.

[16] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, M. Sharbaf, *A survey of model transformation pattern usage*, ICMT 2017.

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

[18] M. Nakicenovic, *Agile driven architecture modernization to a MDD solution*, IJAS, vol. 5, 2012.

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

[20] OMG, *OCL Version 2.4*, 2014.

[21] OMG, *Semantics of a Foundational Subset for Executable UML Models (FUML)*, v1.1, 2015.

[22] K. Schwaber, M. Beedble, *Agile software development with Scrum*, Pearson, 2012.

[23] B. Selic, *What will it take? A view on adoption of model-based methods in practice*, Software systems modeling, 11: 513–526, 2012.

[24] G. Selim, S. Wang, J. Cordy, J. Dingel, *Model transformations for migrating legacy deployment models in the automotive industry*, SoSyM (2015), 14: 365–381.

[25] S. Yassipour-Tehrani, K. Lano, *Requirements engineering in model transformation development: a technique suitability framework*, International Journal on Advances in Software, vol. 9, nos. 3, 4, IARIA, 2016.

[26] E. Willink, *An extensible OCL virtual machine and code generator*, OCL '12, 2012.

[27] S. Zschaler, I. Poernomo, J. Terrell, *Towards using constructive type theory for verifable modular transformations*, FREECO' 11.

A Semantic mapping from C

In order to establish semantic preservation of the UML source specification in the generated C code, we use the UML-RSDS semantics of [15] (based upon the standard OCL mathematical semantics [20]) for UML, and assign a corresponding semantics to C as follows (Table 30).

C element e	*Semantic denotation e'*
int	Range $INT\_MIN..INT\_MAX$ for particular implementation, denoted INT
long	$LONG\_MIN..LONG\_MAX$, denoted LONG
double	IEEE 754 double-precision floating point
struct E*	Domain OBJ of object identifiers
void*	OBJ
char*	seq(INT)
struct E**	seq(OBJ)
e_instances	Set es of existing E instances, $es \subseteq OBJ$
Member T f; of struct E	Map $f\_E : es \to T'$

Table 30: Semantic denotation of C programs

The denotation of members f of E can be abstracted to mathematical functions f_E because each ex in *e_instances* has a value which is the memory location l of (the start of) a struct E data group. $ex \to f$ is the contents of a location $l + ft$ offset from l by a fixed number of memory locations calculated from the declaration of E. Thus, given a valid instance ex of E, a value of T' can be obtained.

Note that *null* can never be an existing object of any class (it cannot be a member of the set es), however it is considered to belong to the type OBJ corresponding to the class, as in [20].

We assume that malloc and calloc always succeed and allocate areas of memory which are not already used. Semantically, these operations add elements to certain es sets, and OBJ must be of sufficient size that this is always possible.

To prove semantic preservation, we need to show that $Sem_{UML}(e) = Sem_C(c(e))$ for each OCL expression e in a specification, where Sem_{UML} is the semantic denotation of OCL given in [15], $c(e)$ is the translation of e by UML2C, and Sem_C is the semantic representation of C expressions. We assume that e is well-defined, ie., that $Def(e)$ and $Det(e)$ hold [15]. We can consider expressions case by case, only selected examples will be given here. $c(e)$ is $CExpression[e.expId]$ for $e : Expression$.

For numeric expressions, the main issues are datatype sizes and agreement of the specified and implemented numeric operators. Both int and INT should be the domain of 32-bit signed integers, and both long and LONG should be 64-bit signed integers. In special cases alternative datatypes could be used, but then the specifier must adopt the same types as used in the specific implementation, and adapt definitions of definedness appropriately. The operators +, *, - at specification and implementation levels agree using the conventional definitions, provided the result is in the appropriate numeric domain. The operator / on integers should use the 'round towards zero' convention, ie., -5/3 is -1, not -2. Likewise, %

should have the same sign as its first argument. Real numbers should be interpreted by IEEE standard 754 double-precision floating point numbers at specification and implementation levels.

For string values, $Sem_C(str)$ is the sequence of characters up to but not including the first 0 value. Thus for literal strings, $Sem_{UML}(str) = Sem_C(c(str))$ since the mapping to C inserts a 0 character at the end of the OCL string, which cannot itself contain 0. We need also to show that for each unary operator *op* on strings, that

$$Sem_{UML}(str) = Sem_C(c(str)) \Rightarrow Sem_{UML}(str{\rightarrow}op()) = Sem_C(c(str{\rightarrow}op()))$$

and likewise for binary and ternary operators.

The extent $E.allInstances$ of a class E is represented in C by the union of $f\_instances$ where F is a leaf subclass of E (or F is E). Only leaf classes can be concrete in UML-RSDS. Therefore if B is a subclass of A, $B.allInstances$ is a subset of $A.allInstances$.

If *arr* is of type struct E**, then $Sem_C(arr)$ is the representation of this as a mathematical set or sequence, and is the collection of elements up to and not including the first NULL element. Likewise for char* arrays. Thus, if e is $Set\{\}$, $c(e)$ is an array *arr* with $arr[0] = NULL$, and $Sem_C(arr) = \{\}$, which is the same as $Sem_{UML}(e)$.

The appendE(col,x) operation in C operates to add x to the end of the collection col, and returns this collection:

$$Sem_C(append(col, x)) = Sem_C(col) \frown [Sem_C(x)]$$

Thus by structural induction we can infer that

$$Sem_C(c(Sequence\{x1, ..., xn\})) = Sem_{UML}(Sequence\{x1, ..., xn\})$$

for any finite sequence of elements. Likewise for *insert* and other implementations of OCL collection operators. The ocl.h library provides implementations of OCL operators that agree with this semantic interpretation. Note that for *subString*, *indexOfString*, *indexOf*, *insertAtString* and *at*, the OCL numbering convention (indexes start at 1) is used, not the C convention. For any unary collection operator *op* we need to show that:

$$Sem_{UML}(col) = Sem_C(c(col)) \Rightarrow Sem_{UML}(col{\rightarrow}op()) = Sem_C(c(col{\rightarrow}op()))$$

and likewise for binary collection operators.