KMEHR to FHIR case solution with UML-RSDS

Kevin Lano[1], Alireza Rouhi[2]

[1]King's College London, Strand, London, UK
[2]Azabaijan Shahid Madani University, Tabriz, Iran

Abstract
The KMEHR to FMIR case is a large-scale transformation in the medical domain, translating content from the Belgium KMEHR electronic health record (EHR) format to the international standard FHIR format. We analyse the existing ATL solution with regard to measures of quality, and propose an improved solution using UML-RSDS. We show that this solution has effective performance on the test models provided with the case. We also investigate the derivation of an inverse transformation from the UML-RSDS solution.

Keywords
Model transformation, UML-RSDS, ATL

1. Introduction

The existing ATL solution for this case [5] is a large-scale transformation, consisting of 20 matched rules, 32 lazy rules, 36 helper operations and 6 helper attributes. The library package is 346 LOC, and the main transformation module is 973 LOC. Some individual rules are also large (over size 100 using the c-measure of syntactic complexity from [1]).

Apart from the size of the transformation, there are also other quality aspects which could hinder the understanding and maintenance of the transformation:

- Maximum OCL expression length (MEL): token count of the largest subexpression within a rule, considered to be a flaw if greater than 10 [6]
- Excessive fan-out (EFO): more than a threshold number (5) of different operations are called from the rule [1]
- Excessive parameter length (EPL): more than a threshold number (5) of rule input, output or local variables [1]
- Excessive rule size (ERS): rule size $c > 100$ [1]
- Magic numbers (MGN): literal constants (other than 0, 1, true, false, null, enumeration literals and the empty string) are used within a rule
- Duplicated code (DC): exactly cloned sections of a rule occurring in two or more rules.

High MEL, ERS, EFO and EPL in a rule can make comprehension of the rule difficult, and also increase the cost of testing it. MEL can also imply high memory use. MGN and DC can increase the work needed for maintenance.

Table 1 summarises the quality issues of the matched rules of the main transformation module of the ATL case solution.

TTC Workshop, STAF 2023, Leicester, UK
kevin.lano@kcl.ac.uk (K. Lano); rouhi1_80ir@yahoo.com (A. Rouhi)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

Table 1
Quality issues in ATL transformation

Rule	Issues
$Folder$	c = 133, MEL (size = 110), EFO = 8
$SumEHRTransaction$	c = 124, MEL (size = 25), EFO = 12, MGN = 6
$SumEHRTransactionWithAuthor$	MGN = 3
$SumEHRTransactionWithCustodian$	MGN = 4, DC: the $refPrefix$ out-pattern of these rules is a parameterised clone.
$Patient$	c = 99, EFO = 6, MEL (size = 16)
$PatientContact$	DC: cloned out-pattern for $humanName$ with $Patient$ rule
$Organization$	MGN = 1, DC: cloned out-patterns for oid, $nihii$, $riziv$ with $Practitioner$ rule
$Practitioner$	MGN = 1
$Medication$	MEL (size = 15), this expression is also cloned in $Vaccine$ (DC)
$Posology$	c = 160, EPL = 11, MGN = 2
$PosologyWithUnitAndTakes$	EPL = 6, MGN = 5
$AllergyOrIntolerance$	c = 111, MGN = 12
$AllergyOrIntoleranceWithCode$	MEL (size = 17), this expression is cloned in $ProblemWithCode$ (DC)
$Problem$	c = 108, EFO = 6, MGN = 7
$Vaccine$	c = 118, EPL = 7, MGN = 2, MEL (size = 15, DC)

Overall, there are 85 quality flaws in the main module rules, including 23 rules with MGN, 5 with ERS, 7 MEL cases and 4 exact clones. There are some incompleteness issues, eg., the mandatory feature $title$ of an FHIR $Composition$ is not set by the $SumEHRTransaction$ rule or its extensions. The $msgSender$ function is left undefined.

Some of these issues can be addressed by revising the ATL specification, for example, the large number of magic numbers can be removed by defining the constant values as helper attributes in a library. Rule inheritance can be used to reduce rule size and EPL. Clones can be refactored into helper operations. However, other issues are difficult to resolve within ATL, in particular the MEL case in the $Folder$ rule, which involves the union of several collections of FHIR Bundle entries derived by specific mapping rules from different kinds of KMEHR transactions. This arises because ATL out pattern assignments cannot be split into different steps which successively accumulate subsets of a complex result, instead the entire result collection has to be assembled in one expression and assigned in one step. Similarly with the MEL case in $SumEHRTransaction$.

For these reasons, we decided to explore the possible improvement of the transformation by expressing it in UML-RSDS, which has more powerful facilities for decomposing complex transformation processing into smaller and more comprehensible parts (Figure 1).

2. Migrating from ATL to UML-RSDS

The AgileUML toolset for UML-RSDS already contains a parser and analyser for ATL, which translates ATL rules and helpers into UML-RSDS use case postconditions and operations,

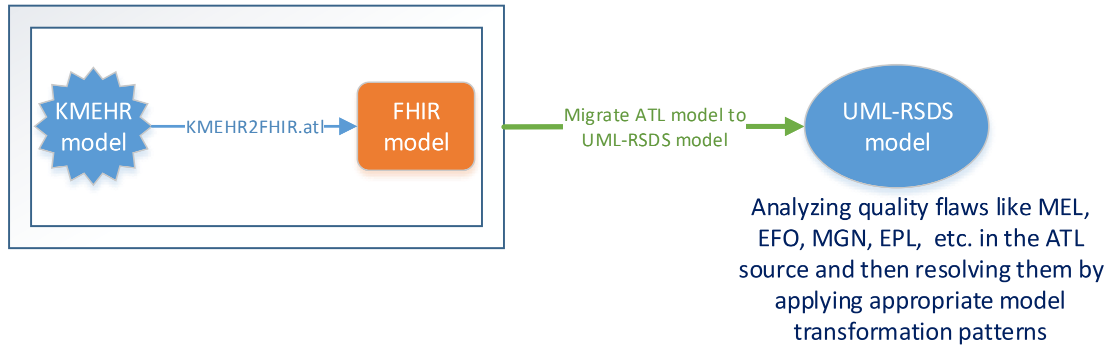

Figure 1: Solution framework

respectively[1]. Quality flaw analysis is performed on the ATL source.

Corresponding to ATL rule inheritance, UML-RSDS has *rule conjunction*, which enables separation of different target mappings into separate rules/postconditions.

For example, the ATL rules:

```
rule A2B {
  from a : A
  to b : B
     ( y <- a.x )
}

rule A2C extends A2B {
  from a : A
  to b : B ( cs <- Set{c} ),
     c : C ( z <- a.x->size() )
}
```

are translated to:

```
A::
  B->exists( b | b.$id = self.$id & b.y = self.x )

A::
  B->exists( b | b.$id = self.$id &
    C->exists( c | c.$id = "c_" + $id &
      c.z = self.x->size() & b.cs = Set{c} ) )
```

Because the $id attributes are identity attributes, the same B instance is updated by both constraints, for a given A instance, so that effectively the conjunction of the two inner predicates $b.y = self.x$ and $c.z = self.x{\rightarrow}size()$ & $b.cs = Set\{c\}$ of the postconditions is achieved.

[1] github.com/eclipse/agileuml

Secondary output variables such as $c : C$ can also be shared and updated by both rules, because of the key assignments $c.\$id = \text{"}c_\text{"} + \$id$. It would be preferable to use meaningful names for these output variables where possible, indicating the role that they play relative to the main output variable. The `mapsTo` keyword of ATL indicates which input and output elements are linked by the same identity:

```
to
  t : T mapsTo s ( ... )
```

for the first out variable t means that $t.\$id = s.\$id$ instead of $t.\$id = self.\$id$.

Unlike with ATL rule inheritance, where only one child rule in a rule family may be executed for a given input element, in UML-RSDS any number of rules may execute for a given element. This enables more flexible splitting of rules, in particular the Entity splitting pattern [3] can be applied to define separate rules to instantiate individual target elements derived from a single source element, thus reducing or eliminating EPL flaws. Entity splitting can also help to reduce ERS and EFO flaws [4].

ATL also supports inheritance between lazy rules, which correspond to entity or use case operations in UML-RSDS. This form of inheritance can be expressed in UML-RSDS by conjunction of the operation postconditions of the inherited and inheriting operations, this applies in the same manner as the conjunction of use case postconditions (UML-RSDS transformation rules) given above.

To address MEL flaws, fine-grained decomposition of postconditions can be defined in UML-RSDS rules or operations. In particular, assigning a union of two or more collections to a target feature g:

```
  t.g = s.f1->union(s.f2)
```

can be split into multiple separate and simpler conjuncts:

```
  s.f1 <: t.g & s.f2 <: t.g
```

within one rule postcondition, or further split into separate postconditions of successive rules. `<:` denotes the subset operator $\subseteq$. Interpreted operationally it adds all elements of the left argument to the right.

UML-RSDS provides (since 2014) the "Restrict input ranges" pattern [3] to optimise rules which involve multiple input linked elements. This corresponds to the new ATL Version 4.8 optimisation feature for this situation [5].

3. Improved solution in UML-RSDS

The transformation was re-expressed in UML-RSDS using a similar set of main rules and auxiliary operations, however with a greater degree of factoring to reduce the number of clones and to exploit the similarities between the processing involved in different ATL rules.

For example, several ATL rules call the $CodeableConcept$ and $CodingWithDisplay$ operations together to wrap a $Coding$ instance in a $Codeable$ instance. This double call can

be replaced by a single operation call of an operation $CodeableConceptForCoding$ which performs both the instance creation and wrapping.

The UML-RSDS transformation is organised using the Phased construction and Object indexing patterns [3]. Phased construction is used to build complex target objects successively from their parts. Object indexing is used for target object lookup, and to ensure that basic types such as $FhirBoolean$ and $FhirString$ behave like *value types*: there are not multiple instances of the types which have the same values. The transformation is represented as a UML use case called $mainModule$.

The KMEHR metamodel, and the relevant subset of the FHIR metamodel, are represented in KM3 format in the file $kmehr.km3$. The forward transformation, together with auxiliary operations, is defined in $ttc23.km3$. To produce an implementation for the transformation, these files are successively loaded using the AgileUML File menu option "Load metamodel $\rightarrow$ KM3". The specification should be type-checked (Analysis menu), and a design generated (Synthesis menu), then code generated using the "Java6" option on the Build menu. The generated Java code is placed in the $kmehr$ subdirectory and can be compiled using $javac\ kmehr/Controller.java$ followed by $javac\ kmehr/GUI.java$. The GUI can be run as $java\ kmehr/GUI$, and provides a visual interface, the $loadXMLModel$ option loads $in.xmi$, $mainModule$ runs the transformation, and $saveModel$ saves the target model to $out.txt$.

3.1. Quality improvement

The transformation clarity has been improved in several aspects: (i) it is more concise, in particular the matched rules part is reduced to 45% of the original length in terms of LOC; (ii) a 'bottom-up' process is used to construct complex target objects successively from simpler objects, whereas the ATL transformation uses a top-down process; (iii) explicit conversion of source elements to target elements is used, whilst the ATL transformation uses implicit conversion. For example, the expression $Address[personxx.address.\$id]$ returns the collection of FHIR $Address$ instances corresponding to the KMEHR $AddressType$ instances in $personxx.address$; (iv) clones and other similar processing steps have been replaced by calls of operations that factor out the duplicated code; (v) the frequency of magic numbers and other flaws has been reduced. Table 2 compares the ATL and UML-RSDS solutions for the matched rules. Rule length is measured in LOC.

The total number of 'magic numbers' in the matched rules have been reduced from 47 to 13. The MEL values of matched rules have been reduced, in particular the MEL for $Folder$ has been reduced from size = 110 to 6, and the MEL for $SumEHRTransaction$ from size = 25 to 12. The MEL over all matched rules has been reduced to 13 (for $DocumentRoot$). The maximum c value of matched rules has been reduced from 160 to 144. The 4 exact clones and 1 parameterised clone of the ATL version have been factored out. The $Organization$ and $Practitioner$ rules are now closely similar and could be further factored. Likewise for $Patient$ and $PatientContact$.

On the other hand, although the rule and operation inheritance organisation in the ATL version has been retained in UML-RSDS, the explicit inheritance relations in ATL are only implicitly represented in the UML-RSDS version[2].

[2]The AgileUML tools do however issue warnings when two rules may update the same target object, as in the

Table 2
ATL and UML-RSDS versions

Rule	ATL length	UML-RSDS length
DocumentRoot	8	5
Folder	25	12
SumEHRTransaction	48	24
SumEHRTransactionWithAuthor	24	7
SumEHRTransactionWithCustodian	27	10
Patient	30	15
Address	11	7
Telecom	16	11
PatientContact	23	11
Organization	23	7
Practitioner	27	8
Medication	23	7
Posology	54	13
PosologyWithUnitAndTakes	35	15
AllergyOrIntolerance	45	22
AllergyOrIntoleranceWithCode	20	7
Problem	39	24
ProblemWithCode	19	7
Vaccine	44	14
Total	541	226

3.2. Performance

The UML-RSDS transformation was tested on the provided examples. The protocol defined in [5] was followed. Table 3 shows the average execution time of the UML-RSDS transformation on each input model, using three independent executions. These times are for Java version 1.8 with 25% processor allocation on a 4-core Windows 10 laptop with Intel i5-7440HQ CPU at 2.8GHz, 8GB RAM. Standard Java settings are used, except that for the largest model the stack size was increased to 8MB: `java -Xss8m` (needed for XML parsing, rather than the transformation itself).

Table 3
Performance of UML-RSDS version

Input model	Execution time (ms)	Output model size (KB)
1	31.3	77
10	56.3	440
100	194.3	4171
1000	3857	42560

The produced output models are provided in the solution repository on Github.

case of matched rule inheritance.

4. Inverse transformation

Apart from facilitating formal analysis, the logical expression of transformation rules in UML-RSDS also supports the synthesis of inverse transformations in many cases [2]. For certain forms of predicates $Succ$ which may appear in a rule postcondition, an inverse $Succ^{\sim}$ can be defined. For example, an assignment

$$t.g = Set\{s.f\}$$

inverts to $s.f = t.g{\rightarrow}any()$. The inverse of a rule

```
A::
  PCond(a) =>
    B->exists( b | b.$id = $id & SCond(b) & Succ(a,b) )
```

is then

```
B::
  SCond(b) =>
    A->exists( a | a.$id = $id & PCond(a) & Succ~(a,b) )
```

in the case that $Succ$ is invertible. The inverse rule expresses an invariant of the forward transformation.

The task of the inverse transformation in the KMEHR to FHIR case is to reconstruct the KMEHR source information from a FHIR model which has been built using the forward transformation. A common situation in the forward transformation is the assignment of some function of a source attribute to a target attribute, of the form

$$t.g = expr(s.f)$$

If an inverse function $expr^{\sim}$ exists, then this assignment inverts to $s.f = expr^{\sim}(t.g)$. Likewise,

$$t.g = Set\{expr(s.f)\}$$

inverts to $s.f = expr^{\sim}(t.g{\rightarrow}any())$. Thus in the UML-RSDS version of the $Address$ rule, the assignment

```
addrx.postalCode = Set{FhirString.newFhirString(self.zip)}
```

inverts to:

```
self.zip = addrx.postalCode.any.value
```

Specific inverse functions may need to be introduced, eg., to invert the $addressLine()$ operation of an $AddressType$ instance to recover the individual $street$, $housenumber$ and $postboxnumber$ values from the tab-separated concatenation of these values. In some cases the basic mappings of values are not injective (eg., the mapping of gender designations from KMEHR to FHIR, where both $\#changed$ and $\#undefined$ values in KMEHR map to $\#other$ in FHIR). For these cases the inverse will néed to be custom-coded for the specific problem.

The inverse of a $\rightarrow collect(\ x \mid expr(x)\ )$ assembly is a $\rightarrow collect$ of $expr^{\sim}$ values. For example:

```
t.given = s.firstname->collect( fn | FhirString.newFhirString(fn) )
```

inverts to

```
s.firstname = t.given->collect( gn | gn.value )
```

In general, the FHIR metamodel and representation is more elaborate than the KMEHR representation, so that one KMEHR object may map to a group of linked FHIR objects (eg., a $TelecomType$ instance maps to linked $ContactPoint$, $ContactPointSystem$ and $ContactPointUse$ instances). Therefore the inverse transformation needs to combine information from multiple FHIR objects to populate the corresponding KMEHR object.

For example, the $ContactPoint$ rule:

```
TelecomType::
    ContactPoint->exists( contactx |
        contactx.$id = self.$id &
        ContactPointSystem->exists( cpsys |
          cpsys.$id = "cpsys_" + $id &
          cpsys.value = self.system() &
          ContactPointUse->exists( cpuse |
            cpuse.$id = "cpuse_" + $id &
            cpuse.value = self.contactPointUse() &
            cpsys : contactx.system &
            cpuse : contactx.use &
            FhirString.newFhirString(
                telecomnumber->trim()) : contactx.value)))
```

inverts to the rule:

```
ContactPoint::
  cpsys : self.system &
  cpuse : self.use =>
      TelecomType->exists( telex |
          telex.$id = self.$id &
          CDTELECOM.newCDTELECOM(cpsys.value) : telex.cd &
          telex.telecomnumber = self.value.any.value )
```

We have defined inverse rules for the $Patient$ rule and all rules that contribute to the $PersonType$ to $Patient$ mapping, so that it is possible to recover KMEHR $PersonType$ information from an FHIR $Patient$. The inverse transformation is defined by the $fhir2kmehr$ use case.

In some cases, source information is not mapped to the target, eg., the $text$ of an allergy or intolerance. In such cases there is no way to reconstruct the complete source information from the target.

The inverse transformation is implemented in the same way as the forward transformation. The inverse transformation, together with auxiliary operations, is defined in $fhir2kmehr.km3$.

$kmehr.km3$ and $fhir2kmehr.km3$ are successively loaded using the AgileUML File menu option "Load metamodel $\rightarrow$ KM3". Rename the specification to $fhir2kmehr$ (File menu, first option). The specification should be type-checked (Analysis menu), and a design generated (Synthesis menu), then code generated using the "Java6" option on the Build menu. The generated Java code is placed in the $fhir2kmehr$ subdirectory and can be compiled using $javac\ fhir2kmehr/Controller.java$ followed by $javac\ fhir2kmehr/GUI.java$. The GUI can be run as $java\ fhir2kmehr/GUI$, the $loadModel$ option loads $in.txt$ (this should be the same file $out.txt$ produced by the forward transformation), $mainModule$ runs the transformation. In the existing code we have added println statements to display the generated KMEHR elements.

Conclusions

We have described an alternative solution to the KMEHR to FHIR case, using UML-RSDS to provide a more concise version of the transformation, with improved quality measures compared to the original. The efficiency of this solution is satisfactory and it can also be used as the basis of an inverse transformation from FHIR to KMEHR.

References

[1] S. Kolahdouz Rahimi, K. Lano et al, *A comparison of quality flaws and technical debt in model transformation specifications*, JSS, 2020.

[2] K. Lano, *Agile Model-based Development using UML-RSDS*, CRC Press, 2016.

[3] K. Lano et al., *A survey of MT design patterns in practice*, JSS, 140, pp. 48–73, 2018.

[4] A. Rouhi, K. Lano, *Towards a pattern-based model transformation framework*, Software: Practice and Experience, 2023.

[5] D. Wagelaar, *The TTC 2023 KMEHR to FHIR Case*, TTC 2023, STAF 2023.

[6] M. Wimmer, S. Martinez, F. Jouault, J. Cabot, *A Catalogue of Refactorings for model-to-model transformations*, Journal of Object Technology, vol. 11, no. 2, 2012, pp. 1–40.