

The Agile UML Manual

K. Lano

September 30, 2019

Contents

1	Introduction	2
2	Class diagrams	3
2.1	Drawing a class diagram	5
2.2	Generating Java	7
2.3	Bi-directional associations	9
2.4	Object deletion	10
2.5	Qualified associations	10
2.6	Association classes	10
3	Constraints	11
3.1	Collections	18
3.2	Type checking	18
3.3	Differences to OCL	20
4	Operations	22
5	Use cases	24
6	Model transformations	27
6.1	Bidirectional transformations	30
6.2	Importing ATL modules	33
6.3	Importing QVT-R transformations	38
6.4	QVT-R in-place mode	42
6.5	Importing ETL modules	44
6.6	Importing Flock modules	45
7	Refactoring	48
8	Design patterns	49
9	State machines	50
10	Synthesis of EIS applications	51
11	Mapping to B	52
12	Mapping to Z3	59
13	Mapping to SMV	61
14	Interactions	62

15 Interoperation with Eclipse	63
15.1 Exchanging metamodels	63
15.2 Exchanging models	64
16 Extensions and importing libraries	64
16.1 External classes and operations	65
16.2 Adding an import	65
16.3 Adding a new operator	65
16.4 Libraries	66
16.5 Plugins and DSLs	66
17 The software development process using UML-RSDS	66
18 Current state of the tools	68
18.1 Planned extensions	68
18.2 Release version history	68
18.3 Known bugs	69
A Reserved words	71
B Expression syntax of UML-RSDS	72
C Activity syntax of UML-RSDS	73
D Mappings from UML-RSDS to Java, C#, C, C++ and Python	74
E Read and write frames, definedness and determinacy	75
F Confluence checks	77
G OCL expression semantics	79
H Alternative Z3 mapping	82
I Format of mm.txt file	83
J Transformation synthesis from metamodels	84

1 Introduction

This manual gives a guide to versions from 1.5 (2015) to 1.9 (2019) of the UML-RSDS tools for model-driven development. From Version 1.8 onwards the tools have been rebranded as “Agile UML” on Eclipse (<https://projects.eclipse.org/projects/modeling.agileuml>).

UML-RSDS/Agile UML provides rapid automated construction of software systems (in Java, C, JSP/Servlets, Python, C# or C++) from high-level UML specifications (class diagrams, use cases, state machines and activities). The tools are currently free to download and use. In general, standard UML 2 and OCL 2 is used for specification and design.

The tool can be used as part of software development to reduce coding costs, time and errors, and to ensure a greater consistency between system components written in multiple languages. It can ensure the correct and consistent implementation of business data and logic in multiple system components.

We have simplified OCL notation for ease of use. In particular, two-valued logic is used, and all collections are either sets or sequences. This also facilitates verification, in cases of high-integrity systems requiring a high degree of assurance.

The software is released for use under the Eclipse EPL-2.0 license (<http://www.eclipse.org/legal/epl-2.0>). Copyright in the software remains with Dr Kevin Lano.

The tools prior to release 1.9 can be downloaded from: <https://nms.kcl.ac.uk/kevin.lano/uml2web>. Download the `umlrsds.zip` file, uncompress, and execute the tool as

```
java -jar umlrsds.jar
```

in the main directory, which contains a writable subdirectory called `output`. Java (v1.4.1 or later) is required to run the tools. This should bring up the initial screen similar to that shown in Figure 1.

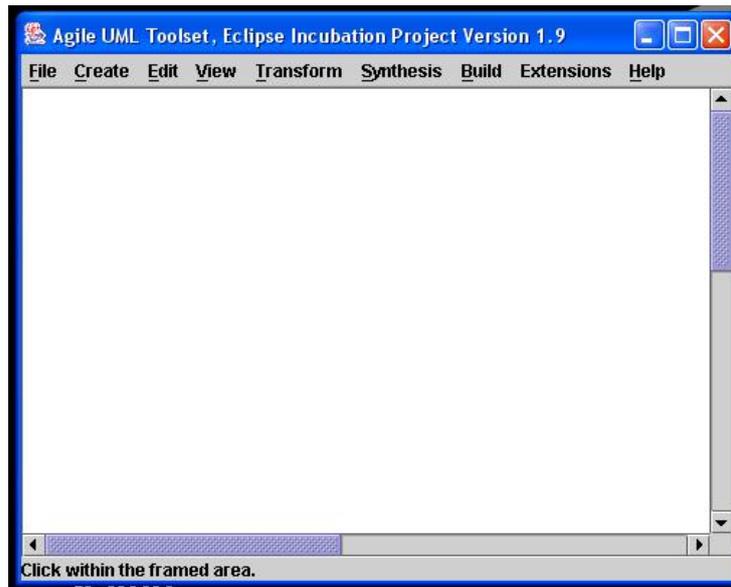


Figure 1: Start screen of UML-RSDS/Agile UML

To explain the simple use of UML-RSDS to rapidly generate applications from UML specifications, the following example system will be used:

- There is data on the courses, students and lecturers within each department of a college.
- Students are registered for a set of courses, and each lecturer teaches a set of courses.
- The system provides operations to add and remove students from courses, and to modify the set of courses taught by a lecturer.

2 Class diagrams

This is the central model of UML-RSDS and of UML. The menu options are, in brief:

File menu: options *Set name* to set the package name of the class diagram model, *Recent* to load a recent class diagram (in the file `output/mm.txt`), *Load metamodel* to load a class diagram in either text format, KM3 format, Ecore XML format, or in Excel CSV format. A menu *Load transformation* has options *Load standard ATL* to convert an ATL module (in `output/mm.atl`) to UML-RSDS (the option *Load refining ATL* is for refining mode ATL), *Load Flock* to convert a Flock module (in `output/flock.txt`) to UML-RSDS, *Load ETL* to convert an ETL module (in `output/mm.etl`) to UML-RSDS, *Load QVT* to convert a QVT-R

transformation (in *output/mm.qvt*) to UML-RSDS. On the main file menu there are further options: *Save* to save the current model in text format (into *output/mm.txt*), *Save as*, to save the class diagram to a file in a number of formats, including *Save data* to save in text format, *Save Ecore* to save as an Ecore metamodel definition (in *output/My.ecore*) and *Save model* to save as an instance model of the UML-RSDS design metamodel. *Print* and *Exit* options are also on the *File* menu. A *Convert* menu allows XMI data in XML files (by default, *output/xsi.txt*) to be converted to model data format (in *output/model.txt*).

Create menu: options

Class – create an entity type (class), including interfaces and abstract classes

Type – create an enumerated type

Association – create an association between two classes or between a class and itself

Inheritance – draw an inheritance arrow from the subclass to the superclass (the superclass must be an abstract class or an interface)

Association Class – create an association class

Use Case – create a use case for an enterprise information system, or a general use case

Entity Statechart – create a state machine to define the behaviour of an entity (its life history)

Operation Statechart – create a state machine to define the algorithm of a specific operation

Entity Activity – define behaviour of object by an activity (pseudocode)

Operation Activity – define the behaviour of an operation by an activity

Use Case Activity – define the behaviour of a use case by an activity (if a use case includes one or more others)

Interaction – describe an example of system behaviour by a sequence diagram

GUI – describe a GUI by a text file

Invariant – introduce a constraint for a class or for the system.

Requirements – open a SysML editor to write requirements for the system (versions from 1.6 onwards only).

Edit menu – options to *Move*, *Delete* and *Modify* visual elements, also to modify use cases and to delete invariants and use cases.

View menu – options to view the invariants, types, operations, activities and use cases of the system.

Transform menu – options to apply refactoring, refinement or design pattern transformations to a class diagram or system.

Synthesis menu – options to analyse the constraints of the system and to generate a design, SMV, B AMN and Z3 code.

Build menu – options to generate Java 4, Java 6, Java 7, C#, C++, C, Python code and EIS/Web system code.

Extensions menu – options to add (language-specific) imported packages, to define language extensions, and to execute plugins.

Help menu – brings up help panel on the right of the screen.

It is possible to load a class diagram in parts: if a class (with the same name) is defined both in the current displayed model and in a loaded file, then the union of the features and stereotypes of the two definitions is taken.

The default action for a mouse press/click on a class, use case or association is to edit that element.

The following key shortcuts can be used:

- Alt-F, Alt-R for Recent
- Alt-E, Alt-U for Edit Use Case
- Alt-F, Alt-S for Save
- Alt-S, Alt-D for Generate Design
- Alt-S, Alt-J for Generate Java 4
- Alt-S, Alt-6 for Generate Java 6
- Alt-S, Alt-7 for Generate Java 7
- Alt-C for Create menu

The following command-line options can be used:

```
java -jar umlrsds.jar -gj
```

loads the model in *output/mm.txt*, generates the design and Java (4) code of the model.

The command line option:

```
java -jar umlrsds.jar -gcs
```

loads the model in *output/mm.txt*, and generates the design and C# code of the model.

The command

```
java -jar umlrsds.jar -jsp
```

loads the model in *output/mm.txt*, and generates JSP-style web system code for this model.

2.1 Drawing a class diagram

To create our example system, we create the four classes *Student*, *Course*, *Lecturer*, *Department*. Classes are created by selecting the *Create Class* option, entering the name, and clicking on the location for the class. Figure 2 shows the class definition dialog. Options in the *stereotypes* field include *abstract* for abstract classes, and *interface* for interfaces. *leaf* denotes that the class cannot have subclasses. *persistent* denotes that the class represents persistent data (eg., data repository data in an enterprise information system). In this example the default *none* option is used.

Class names (by convention) should begin with a capital letter, they can then consist of any further letters or digits. Note that `_` is not a permitted character, but `$` is.

The classes are then edited by clicking on the class, or by using the *Modify* option to add attributes *name* : *String* to each entity, and associations are created, using the dialog of Figure 5: a 1-* association from *Department* to *Lecturer*, with role 1 blank and role 2 as *staff*; a 1-* association from *Department* to *Course*, with role 1 blank and role 2 as *courses*; a 1-* association from *Department* to *Student*, with role 1 blank and role 2 as *students*; a *-1 association from *Course* to *Lecturer*, with role 1 blank and role 2 *lecturer*; a *-* association from *Student* to *Course*, with role 1 blank and role 2 *taking*. Associations are created by selecting *Create Association* and dragging the mouse from the source class to the target class. Waypoints can be created by pressing the space key at the point (provided the class diagram editor panel has focus).

Association stereotypes include *implicit* (for associations calculated on demand using a predicate, rather than being stored), *source* (for associations belonging to the source metamodel of a model transformation), etc. The default *none* can be used in most cases.

The construction process should produce the class diagram of Figure 3.

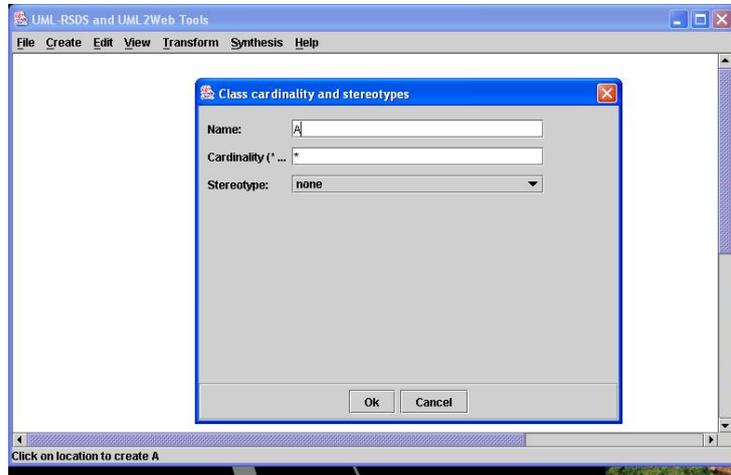


Figure 2: Class definition dialog

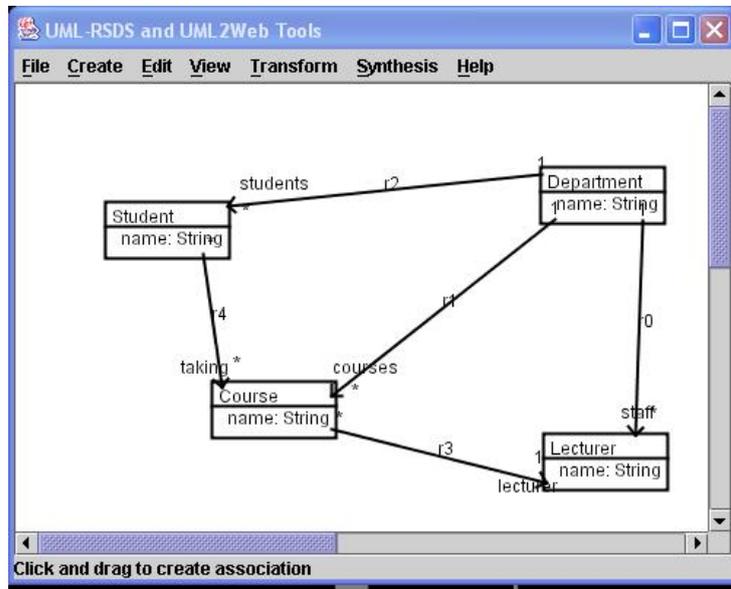


Figure 3: Class diagram

2.2 Generating Java

Java code can be generated from this model by the option Generate Java on the Build menu. The options include Java 4, Java 6 and Java 7. The main difference is that the Java 6 code uses HashSet to represent unordered association ends, and ArrayList to represent ordered association ends, whilst the Java 4 code uses Vector for both. The Java 6 version can be more efficient for systems involving uni-directional unordered associations. Java 7 generation has been added from version 1.6. This uses templated classes such as *ArrayList < A >* to represent sequences of A objects.

The code consists of Java classes for each class of Figure 3, in a file *Controller.java*, together with the main class, *Controller*, of the system. A file *SystemTypes.java* contains an interface with utility operations (for computing OCL expressions), and *ControllerInterface.java* contains an interface for *Controller*. These files are in the same Java package *pack* if this is specified as the system name. A file *GUI.java* defines a simple Swing interface for the system, with buttons for each use case. The file *Controller.java* is also displayed to the user (Figure 4).

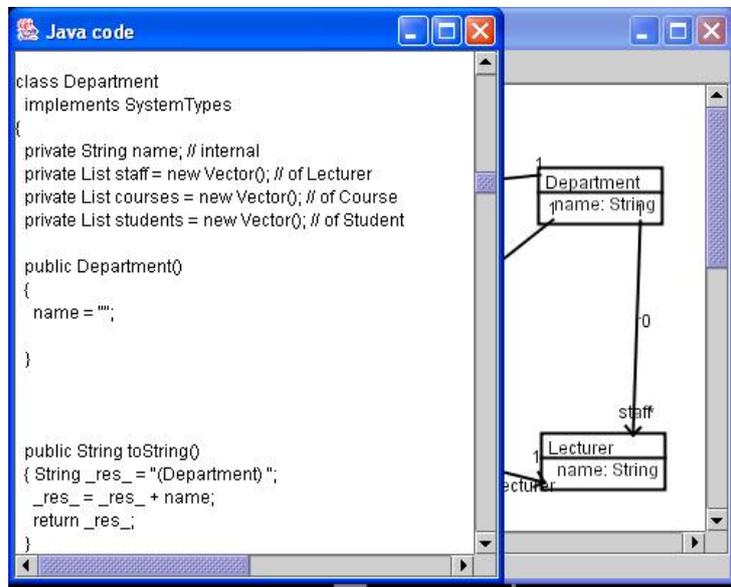


Figure 4: Display of Java

The Java code can be used directly in other applications, a standard interface of operations is provided for each class *C*. In the case of the Java 4 generated code, this is:

- A no-argument constructor $C()$ which sets attributes and roles to the default values of their types.
- A constructor $C(T\ attx, \dots, D\ rolex)$ which takes input values for each *final* or *frozen* instance attribute and for each 1-multiplicity role.
- An operation $setatt(T\ attx)$ for each modifiable instance attribute $att : T$
- An operation $setrole(D\ rolex)$ for each modifiable instance role $role : D$.
- An operation $setAllf(List\ objs, T\ val)$ to set $ob.f$ to val for each $ob : objs$ for each modifiable feature $f : T$.

For sequence-valued roles f there is an operation $setAllf(List\ objs, int\ ind, D\ fx)$ to set $ob.f[ind]$ to fx for $ob : objs$.

- Operations $setrole(List\ rolex)$, $addrole(D\ rolexx)$, $removerole(D\ rolexx)$, $addAllrole(List\ objs, D\ rolexx)$, $removeAllrole(List\ objs, D\ rolexx)$ for each modifiable many-valued role $role : Set(D)$ or $role : Sequence(D)$. The $removerole$ and $removeAllrole$ operations are omitted if the role is $addOnly$. An operation $setrole(int\ i, D\ rolex)$ is included for sequence-valued roles, to set the i -th element of $role$ to $rolex$.
- An operation $getrole() : D$ for a 1-multiplicity role $role : D$.
- An operation $getatt() : T$ for an attribute $att : T$.
- An operation $getrole() : List$ for a many-valued role $role$.
- A static operation $getAllf(List\ objs) : List$ which returns the set (ie., with duplicate values removed) of all $ob.f$ values for $ob : objs$, for any data feature f of the class.
- A static operation $getAllOrderedf(List\ objs) : List$ which returns the sequence/bag of all $ob.f$ values for $ob : objs$, for any data feature f of the class (with duplicate values preserved). This corresponds to OCL $objs \rightarrow collectNested(f)$.
- For interfaces I , the above static operations are instead placed in an inner class $IOps$ of I .
- There are also specialised operations for manipulating elements of qualified associations using qualifier index values.

Both sets and sequences are represented by the Java List type, and are implemented as Vectors. The Java 6 and 7 code instead uses Collection as a general input type, and HashSet and ArrayList for sets and sequences, respectively.

One limitation compared to standard UML is that multiple inheritance is not fully supported (except for implementation in C++). Only multiple interfaces can be inherited by a single class, not multiple classes. However, unlike Java, such interfaces can contain instance features, which can be inherited by multiple classes. If this is used to simulate multiple inheritance, then before synthesising Java, the Refactoring transformation *Push down abstract features* should be used to copy such features down to all subclasses, thus achieving the effect of multiple class inheritance.

A language rule of UML-RSDS is that any class with at least 1 subclass must be abstract, and classes with no subclass must be concrete (in a completed specification).

Controller.java contains a public set of operations to modify the objects of a system:

- An operation $setatt(E\ ex, T\ attx)$ for each modifiable instance attribute $att : T$ of class E .
- An operation $setrole(E\ ex, D\ rolexx)$ for each modifiable instance role $role : D$ of E .
- Operations $op(E\ ex, pardec)$ and $AllEop(List\ exs, pardec)$ for each update operation $op(pardec)$ of class E .
- Operations $List\ AllEop(List\ exs, pardec)$ for each query operation $rT\ op(pardec)$ of class E .
- Operations $addrole(E\ ex, D\ rolexx)$, $removerole(E\ ex, D\ rolexx)$ for each modifiable many-valued role $role : Set(D)$ or $role : Sequence(D)$ of class E . $removerole$ is omitted if the role is $addOnly$.
- An operation $public\ rT\ uc(pardec)$ for each general use case uc of the system.

Note that $objs.op(e)$ is interpreted by the Controller operation $AllEop(objs, e)$ and always returns a sequence-typed result (representing a bag of values) even if $objs$ is set-valued.

Generation of C# follows the same structure as for Java, with classes E.cs for each UML class E, Controller.cs and SystemTypes.cs being produced. C# ArrayList is used in place of Java List. If a system name is specified, this name is used as the C# namespace name of the generated code. In Visual Studio, all code can be included in a single Project.cs file. To add timing information, the C# class Stopwatch can be used, with operations Start, Stop and ElapsedMilliseconds.

For C++, class declarations are placed in a file `Controller.h`, code is placed in `Controller.cpp`. Note that for Visual Studio, the `Controller.cpp` code becomes the text of the main `Application.cpp` file, and all necessary libraries are *#included* in this, as is `Controller.h`. The C++ `time()` function from `<ctime>` can be used to analyse the execution time of generated code.

For C, a header file `app.h` is generated separately to a code file `app.c`, the latter also depends upon the `ocl.h` library (in the `/libraries` subdirectory).

For Python, all code is generated in one `app.py` file.

Tables 21, 22 in the appendix show in detail the mapping of UML and OCL to Java 4, Java 6, Java 7, C#, C++, C and Python.

2.3 Bi-directional associations

Associations with an inverse can be specified by filling in the ‘Role 1’ field in the association creation dialog (Figure 5).

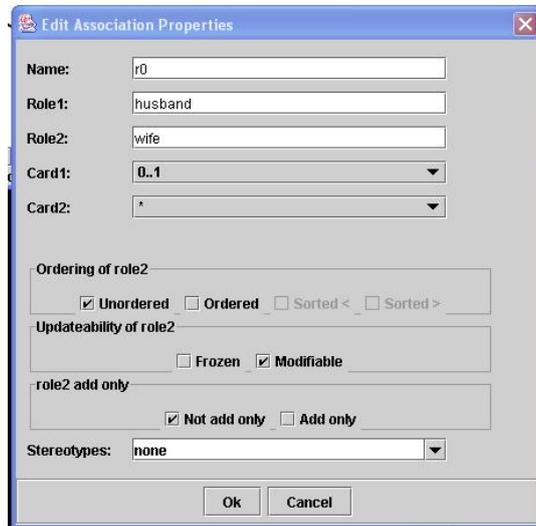


Figure 5: Create association dialog

If the association is drawn from class E1 to class E2, then an inverse association from E2 to E1 will be created also. The inverse association is treated as a feature of E2. The generated Java code will automatically maintain the two associations as mutual inverses: updates to the forward association will generally cause updates to the inverse association also, and vice-versa. This maintenance code is in the Controller operations such as `addrole2(E1 e1x, E2 e2x)` and `killE2(E2 e2x)`.

In some cases, the existence of an inverse association may enable more efficient code to be produced. However, it also closely semantically links the classes E1 and E2, so that these must be contained in the same code module. In general, bidirectional associations should only be introduced if they are necessary for the system being developed, ie., navigation in both directions along the association is required for the functionality of the system.

If one end of a bidirectional association is `{addOnly}`, then the other end is also implicitly `{addOnly}` (for a many-valued opposite end), or `{frozen}`, for a single-valued opposite end (because a change to the opposite end of object *x* requires removal of *x* from any existing forward role sets).

Bidirectional association management is not supported by the C or Python translators, for these the specifier must perform their own data management.

2.4 Object deletion

An object or set of objects x of class E is deleted by a statement $x \rightarrow isDeleted()$. This deletion means that the objects are no longer valid elements of E , and leads to several further possible effects:

1. x is removed from any association which has E as an end class.
2. x is removed from any superclass of E .
3. If an association $r : F \rightarrow E$ has multiplicity 1 at the E end, then any F object linked to x by r is also deleted (since each existing F object must have exactly one valid associated E object).
4. If an association $r : E \rightarrow F$ or $r : E \rightarrow Collection(F)$ is an aggregation (composition association) at the E end, then any F objects linked to x by r will also be deleted: this is the ‘deletion propagation’ semantics of aggregation.

The generated Java/C#/C++ code carries out this succession of deletion actions via operations *killE*, *killF*, etc, of the *Controller* class. If cascaded deletions in cases 3 and 4 are not intended, then the F objects should be unlinked from x before the deletion of x .

If an association end is *{addOnly}*, then objects of the class at this end can only be deleted if they do not occur in any role set of this association.

Object deletion is not currently supported by the C or Python translators.

2.5 Qualified associations

A qualified association br from class A to class B represents a String-indexed map from each A object to individual B objects or to sets of B objects (depending on if the multiplicity at the B end is 1 or not). The multiplicity at the A end is assumed always to be *, and this end is un-named. Qualified associations cannot be static.

Given such an association, the Java code generated for A will represent br as a Java Map, and include operations to modify and read br , based on the index values. For many-valued br , the operations are:

- An operation $setbr(String\ ind, List\ rolex)$ to set $br[ind]$ to a set $rolex$ of B objects.
- Operations $addbr(String\ ind, B\ rolex)$, $removebr(String\ ind, B\ rolex)$, to add/remove $rolex$ from $br[ind]$.
- An operation $getbr(String\ ind) : B$ to return $br[ind]$.

Corresponding operations are also defined in the Controller class. Analogous representations are used in C# (the Hashtable class) and C++ (the map template class).

Qualified associations cannot be translated to C, Python, B AMN, SMV or Z3. Their values cannot be stored in/read from model data text files.

2.6 Association classes

An association class is a class C which is also an association between classes A and B (normally a many-many association). In translation to Java, C#, C++ or B AMN, association classes are represented by two implicit many-one associations $a : C \rightarrow A$ and $b : C \rightarrow B$. UML does not permit C to be the same class as A or B , we additionally require A and B to be distinct, and unrelated to each other or to C by inheritance.

Instances of C should not be created directly, but only as a result of linking an A object to a B object by the association. Likewise, deleting C objects is a consequence of unlinking A and B objects.

The set of C objects linked to a particular $ax : A$ is $C \rightarrow select(a = ax)$, likewise for the C 's linked to some B object.

The values of association classes cannot be stored in/read from model data text files. Association classes are not mapped to C or Python.

3 Constraints

OCL and abbreviated OCL constraints can be used in many places in a UML-RSDS specification:

1. Class invariants
2. Global constraints of a class diagram
3. Operation pre and postconditions
4. Use case pre and postconditions and invariants
5. State machine state invariants and transition guards.

Figure 6 shows the dialog for entering constraints.

A constraint $P \Rightarrow Q$ on entity E is entered by writing E in the first text area, P in the second and Q in the third.

This has the logical meaning

$$E \rightarrow forAll(P \Rightarrow Q)$$

“for all instances of E , if P holds, then Q holds”. Operationally, it means “for all instances of E , if P holds, then make Q hold true, if possible”.

The default options (system requirement, non-critical, update code) can be chosen for the other options in most cases.

E is called the *context* or *scope* of the constraint. Local invariants of class C are specified by naming C in the first field of this dialog. Global constraints forAll-quantified over C also need C to be entered here. Use case constraints based on class C should have the class name entered as context (for options Add Postcondition, Add Precondition or Add Invariant on the edit use case dialog).

If the constraint is a local invariant of a class C , or it is a use case constraint based on C , then features of C can be written simply by their names in P or Q . Navigation expressions starting from features of C can also be written.

For example:

`c : courses`

as an antecedent P and

`c.lecturer : staff`

as a succedent Q could be an invariant of class *Department* in our example, expressing that all courses of the department have their lecturer also within the department. c is an implicit variable, whose type is inferred as being *Course*.

The same logical constraint can be alternatively expressed without c by writing

`true`

as the antecedent P and

`courses.lecturer <: staff`

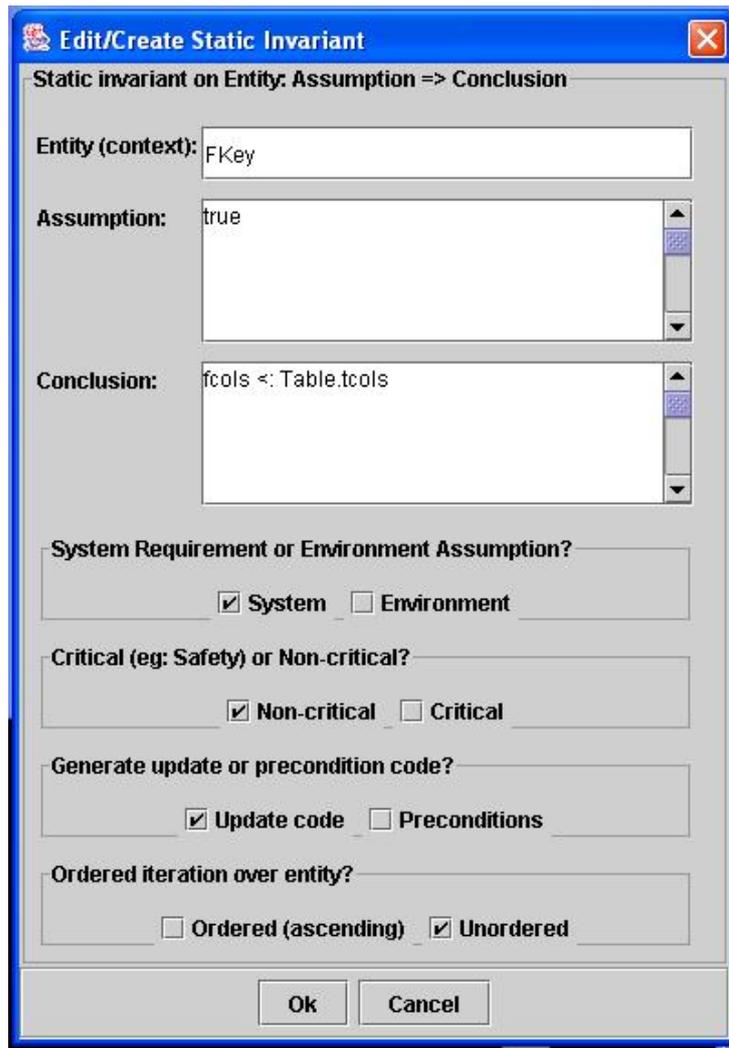


Figure 6: Constraint dialog

as Q . “The lecturers of all courses are members of staff”.

A common simple form of constraint is one that derives the value of an attribute of a class from other features of the class. For example, if we add an integer attribute *numberOfStudents* to *Department* in Figure 3, we can define its value by the invariant constraint

$$\text{numberOfStudents} = \text{students.size}$$

or

$$\text{numberOfStudents} = \text{students} \rightarrow \text{size}()$$

on *Department*.

Table 1 defines the basic expressions which may be used in constraints, and their meanings.

A constraint has scope or context the entities to which it implicitly applies (features of such entities can be written without object references in the constraint, or with the reference *self*). The most local context is termed the *primary scope* of the expression, and feature references are interpreted as features of this entity by default. Other entities in the scope are termed the *secondary scope*.

A local invariant of a class E has primary scope E , as do pre and postconditions of E 's instance operations, and state invariants and guards in the entity state machine of E or in its instance operation state machines. A use case constraint with owner entity E also has primary scope E .

A predicate P in the body of a select, reject or collect expression $e \rightarrow \text{select}(P)$ has primary scope the element type of e , if this is an entity type, and secondary scope the scope of the selection expression.

<i>Expression</i>	<i>Meaning as query or update</i>
<i>self</i>	The current object of class E : for constraints with a single scope E
f $op(e1, \dots, en)$	Attribute/role f of class E , for constraints with scope E Invocation of operation op of E , for constraints with scope E
$f[ind]$	<i>ind</i> element of ordered or qualified role f of current scope E
$E[v]$ v single value	The instance of E with (principal) primary key value v
$E[v]$ v collection	The instances of E with (principal) primary key value in v
$obj.f$ $objs.f$	Data feature f of single object obj Collection (set or sequence) of values $o.f$ for $o \in objs$, collection $objs$. This is a set if $objs$ is a set or if f is set-valued.
$obj.op(e1, \dots, en)$ $objs.op(e1, \dots, en)$	Apply op to single object obj Apply op to each $o \in objs$, collection $objs$ Result value is sequence/bag of individual result values $o.op(e1, \dots, en)$ if op has result
<i>value</i>	Value of enumerated type, numeric value, string value “string”, or boolean true, false

Table 1: Basic OCL expressions in Agile UML

The attribute *key* : *String* used in the lookup $E[v]$ is the principal primary key of E – the most locally-defined identity attribute of E , if any such attribute exists, or otherwise the most locally-defined attribute for which $E \rightarrow \text{isUnique}(key)$ is an invariant of E .

Constraints not only have a declarative meaning at the specification level, defining what conditions should be maintained by a system, but also they may have a procedural meaning, defining code fragments which enforce the conditions. For example, the constraint defining

numberOfStudents will produce code that modifies this attribute in response to any operation *setstudents*, *addstudents* or *removestudents* which affects *students*.

Table 2 shows the logical operators that can be used in constraints, and their declarative and procedural meanings. The procedural interpretation of a formula P is formally defined as a statement in the activity language (Appendix C) denoted by $stat(P)$.

<i>Expression e</i>	<i>Meaning as query e'</i>	<i>Meaning as update stat(e)</i>
A => B	A' implies B'	if A' then stat(B)
A & B	A' and B'	stat(A); stat(B)
A or B	A' or B'	
E->exists(P) entity E	An existing instance of E satisfies P'	
e->exists(P) expression e	An existing element in e satisfies P'	
E->forall(P) entity E	All existing instances of E satisfy P'	
e->forall(P) expression e	All elements of e' satisfy P'	
E->exists1(P) entity E	Exactly one existing instance of E satisfies P'	
e->exists1(P) expression e	Exactly one element of e' satisfies P'	
E->exists(x P) concrete entity E	An existing instance x of E satisfies P'	Create x : E and make P true for x
e->exists(x P) other expression e	An existing element x in e satisfies P'	Select an element of e and apply P to it
E->forall(x P) entity E	All existing instances x of E satisfy P'	for x : E do stat(P)
e->forall(x P) expression e	All elements x of e satisfy P'	for x : e do stat(P)
E->exists1(x P) concrete entity E	Exactly one existing instance x of E satisfies P'	If no instance of E satisfies P, create x : E and make P true for x
e->exists1(x P) other expression e	Exactly one element x of e satisfies P'	If no x : e satisfies P, select some x : e and apply P to it

Table 2: Logical operators and interpretations

There are some special cases in the operational use of the *exists* and *exists1* quantifiers:

- In the case of $E \rightarrow exists1(x | x.id = v \ \& \ P)$ or $E \rightarrow exists(x | x.id = v \ \& \ P)$ where *id* is the principal primary key of concrete entity *E*, a lookup for $E[v]$ is performed first, if this object exists then (i) the update effect of *P* is applied to it (unless *P* already holds for it). Otherwise, (ii) a new instance of *E* is created and the formula $x.id = v \ \& \ P$ is applied to it.
- If *E* is abstract, or of the form $F@pre$ for an entity *F*, then only the lookup and part (i) of this functionality is performed. Likewise for general expressions *e* in place of *E*.

Although checking for existence of an x with $x.id = v$ does affect the efficiency of the implementation, it is necessary to ensure correctness: it would be invalid to create two or more objects with the same id value.

From Version 1.8 we have introduced the concept of “least change” object creation, via the operator $\rightarrow existsLC$. This avoids the creation of a new object to satisfy the RHS, if any object can be found which partly satisfies the RHS (at least one conjunct is already true).

Table 3 lists the comparator operators and their meanings.

<i>Expression e</i>	<i>Meaning as query e'</i>	<i>Meaning as update stat(e)</i>
$x : E$ <i>E</i> entity type	x is an existing instance of E	Create x as a new instance of E (for concrete E)
$x : s$ s collection	x is an element of s	Add x to s
s->includes(x) s collection	Same as $x : s$	Same as $x : s$
$x / : E$ <i>E</i> entity type $x / : s$ s collection	x is not an existing instance of E x is not an element of s	Remove x from s
s->excludes(x) s collection	Same as $x / : s$	Same as $x / : s$
$x = y$	x 's value equals y 's value	Assign y 's value to x
$x < y$ x, y both numbers, or both strings	x 's value is less than y 's. Likewise for $>$, $<=$, $>=$, $/=$ (not equal)	
$s <: t$ s, t collections	All elements of s are also elements of t	Add every element of s to t
t->includesAll(s)	Same as $s <: t$	Same as $s <: t$
$s / <: t$ collections s, t	s is not a subset of t	Remove all elements of s from t
t->excludesAll(s) collections s, t	No element of s is in t	Remove all elements of s from t

Table 3: Comparitor operators

$x : E$ for an entity type (classifier) E corresponds to $x.oclIsKindOf(E)$ in OCL [19] and to $x instanceof E$ in Java and to $x is E$ in C#.

Numeric operators for integers and real numbers are shown in Table 4. Instead of OCL Integer and Real, we use types `int` (32 bit signed integers) and `double` (signed double precision floating point numbers) which correspond to the types with these names in Java, C++ and C#. `long` (64-bit integers) is currently supported in provisional form. All operators take double values as arguments except as noted. Three operators: `ceil`, `round`, `floor`, take a double value and return an `int`.

<i>Expression</i>	<i>Meaning as query</i>
<code>-x</code>	unary subtraction
<code>x + y</code>	numeric addition, or string concatenation (if one of x, y is a string)
<code>x - y</code>	numeric subtraction
<code>x * y</code>	numeric multiplication
<code>x / y</code>	integer division (div) if both x, y are integers, otherwise arithmetic division
<code>x mod y</code>	integer x modulo integer y
<code>x.sqr</code>	Square of x
<code>x.sqrt</code>	Positive square root of x
<code>x.floor</code>	Floor integer of x
<code>x.round</code>	Rounded integer of x
<code>x.ceil</code>	Ceiling integer of x
<code>x.abs</code>	Absolute value of x
<code>x.exp</code>	e to power x
<code>x.log</code>	e -logarithm of x
<code>x.pow(y)</code>	y -th power of x
<code>x.sin</code>	sine of x (given in radians)
<code>x.cos</code>	cosine of x (given in radians)
<code>x.tan</code>	tangent of x (given in radians)
<code>Integer.subrange(st,en)</code>	Sequence of integers starting at st and ending at en , in order

Table 4: Numeric operators

Other math operators in common between Java, C, C# and C++ are: `log10`, `cbt`, `tanh`, `cosh`, `sinh`, `asin`, `acos`, `atan`. These are double-valued functions of double-valued arguments. Some functions may not be available in old versions of Java (`log10`, `cbt`, `tanh`, `cosh`, `sinh`) or C (`cbt`).

String operators are shown in Table 5. Notice that since strings are sequences of characters (single-character substrings), all sequence operations should be expected to be applicable also to strings.

String positions in OCL are numbered starting at 1, as are sequence positions.

A pre-state expression $f@pre$ for feature f can be used in operation postconditions and use case postcondition constraints, but not elsewhere. In an operation postcondition, $f@pre$ refers to the value of f at the start of the operation, ie., the same value as denoted by f in the operation precondition. Occurrences of f without $@pre$ in the postcondition refer to the value of f at the end of the operation. So, for example:

```
op()
pre: b > a
post: b = b@pre - a@pre
```

decrements b by a . Since a itself is not updated by the operation, there is no need to use pre with a , and the operation should be written as:

<i>Expression</i>	<i>Meaning as query</i>
<code>x + y</code>	string concatenation
<code>x->size()</code>	Length of x
<code>x->first()</code>	First character of x
<code>x->front()</code>	Substring of x omitting last character
<code>x->last()</code>	Last character of x
<code>x->tail()</code>	Substring of x omitting first character
<code>x.subrange(i, j)</code>	substring of x starting at i-th position, ending at j-th
<code>x->toLowerCase()</code>	copy of x with all characters in lower case
<code>x->toUpperCase()</code>	copy of x with all characters in upper case
<code>s->indexOf(x)</code>	index (starting from 1) of s at which the first subsequence of characters of s equal to string x occurs. Default is 0
<code>s->hasPrefix(x)</code>	string x is a prefix of s
<code>s->hasSuffix(x)</code>	string x is a suffix of s
<code>s->characters()</code>	sequence of all single character substrings of s in same order as in s
<code>s.insertAt(i, s1)</code>	copy of s with string s1 inserted at position i
<code>s->count(s1)</code>	number of distinct occurrences of s1 as a substring of s
<code>s->reverse()</code>	Reversed form of s
<code>e->display()</code>	displays expression e as string on standard output
<code>s1 - s2</code>	string s1 with all occurrences of characters in s2 removed
<code>e->isInteger()</code>	true if e represents an <code>int</code> value
<code>e->isReal()</code>	true if e represents a <code>double</code> value
<code>e->toInteger()</code>	returns the value if e represents an <code>int</code> value
<code>e->toLong()</code>	returns the value if e represents a <code>long</code> value
<code>e->toReal()</code>	returns the value if e represents a <code>double</code> value

Table 5: String operators

```

op()
pre: b > a
post: b = b@pre - a

```

3.1 Collections

There are two kinds of collection in UML-RSDS: sets and sequences. Sets are unordered collections with unique membership (duplicate elements are not permitted), sequences are ordered collections with possibly duplicated elements. If br is an unordered * role of class A , then $ax.br$ is a set for each $ax : A$. If br is an ordered * role of class A , then $ax.br$ is a sequence for each $ax : A$. The further two OCL collection types, ordered sets and bags, can be defined in terms of sequences.

Table 6 shows the values and operators that apply to sets and sequences. We intend to add a $\rightarrow concatenateAll$ operation, $\rightarrow oclIsTypeOf$ and binary $\rightarrow any$ operator in version 2.0.

For sequences, $s \hat{\ } t$ should be used instead of $s \setminus / t$, which produces the set union of two sets, and $s \rightarrow append(x)$ should be used instead of $s \rightarrow including(x)$.

$s \rightarrow at(i)$ for a sequence s can also be written as $s[i]$ in our notation, if s is an identifier or navigation expression using '!'.

$st = sq$ for set st and sequence sq is always false, even if both st and sq are empty. Two sets are equal iff they contain exactly the same elements, two sequences are equal iff they contain exactly the same elements in the same order.

Of particular significance are collection operators which construct new collections consisting of all elements of a given collection which satisfy a certain property (Table 7).

Any order in the original collection is preserved by *select* and *reject*, ie., if they are applied to a sequence the result is also a sequence. In the operators *select* and *reject* without variables, an element of the first argument, s , can be referred to as *self*.

From Version 1.9 there is prototype support for a $Map(T1, T2)$ type and operators $mp \rightarrow including(x, y)$, $mp \rightarrow excludingKey(x)$, $mp \rightarrow excludingValue(y)$ on these. Map values cannot be saved or loaded from models, and can only be used for internal processing within operations.

3.2 Type checking

For each expression in a specification, its type and element type (for a collection-valued expression) are determined, according to Tables 8, 9 and 10. The least common supertype (LCS) of the types of $x1, x2$ is the most specific type which includes both types. The least common numeric supertype LCNS is *double* if either is double, otherwise *long* if either is *long*, otherwise *int*.

The OCL standard does not define $-$ or \cap when the first argument is a sequence. For convenience we define these as:

$$\begin{aligned}
sq - col &= sq \rightarrow reject(x \mid col \rightarrow includes(x)) \\
sq \cap col &= sq \rightarrow select(x \mid col \rightarrow includes(x))
\end{aligned}$$

so that they are sequence-valued.

A subtle aspect of the semantics of *collect* is that the result of $st \rightarrow collect(e)$ can contain duplicate values for sets st , and the cardinality of the result equals that of st . This is in accordance with the OCL behaviour of *collectNested*, which should return a Bag of the evaluation results.

Errors may be reported during initial type-checking in the cases:

- An operation is used prior to its definition.
- An implicitly-typed let variable is used in a constraint or operation.

These errors should not arise in subsequent type-checking of the same specification. If errors are encountered after re-running the type check, then there is probably an error in your specification.

<i>Expression</i>	<i>Meaning as query</i>
<i>Set</i> {}	empty set
<i>Sequence</i> {}	empty sequence
<i>Set</i> { x_1, x_2, \dots, x_n }	set with elements x_1 to x_n
<i>Sequence</i> { x_1, x_2, \dots, x_n }	sequence with elements x_1 to x_n
<code>s->including(x)</code>	s with element x added
<code>s->excluding(x)</code>	s with all occurrences of x removed
<code>s - t</code>	s with all occurrences of elements of t removed
<code>s->prepend(x)</code>	sequence s with x prepended as first element
<code>s->append(x)</code>	sequence s with x appended as last element
<code>s->count(x)</code>	number of occurrences of x in sequence s
<code>s->indexOf(x)</code>	position of first occurrence of x in s
$x \setminus y$	set union of x and y
x / y	set intersection of x and y
$x \hat{~} y$	sequence concatenation of x and y
<code>x->union(y)</code>	Same as $x \setminus y$
<code>x->intersection(y)</code>	same as x / y
<code>x->unionAll(e)</code>	union of y.e for $y : x$
<code>x->intersectAll(e)</code>	intersection of y.e for $y : x$
<code>x->symmetricDifference(y)</code>	symmetric difference of x, y
<code>x->any()</code>	Arbitrary element of non-empty collection x
<code>x->subcollections()</code>	set of subcollections of collection x
<code>x->reverse()</code>	reversed form of sequence x
<code>x->front()</code>	front subsequence of sequence x
<code>x->tail()</code>	tail subsequence of sequence x
<code>x->first()</code>	first element of sequence x
<code>x->last()</code>	last element of sequence x
<code>x->sort()</code>	sorted (ascending) form of sequence x
<code>x->sortedBy(e)</code>	x sorted in ascending order of y.e values (numerics or strings) for $y : x$
<code>x->flatten()</code>	Expand out all nested collections in x
<code>x->sum()</code>	sum of elements of collection x
<code>x->prd()</code>	product of elements of collection x
<code>x->max()</code>	maximum element of collection x
<code>x->min()</code>	minimum element of collection x
<code>x->asSet()</code>	set of distinct elements in collection x
<code>x->asSequence()</code>	collection x as a sequence
<code>s->isUnique(e)</code>	true if the collection of values of x.e for $x : s$ has no duplicates
<code>x->isDeleted()</code>	destroys all elements of collection x. Can also be used on individual objects x

Table 6: Collection operators

<i>Expression</i>	<i>Meaning as query</i>
<code>s->select(P)</code>	collection of s elements satisfying P
<code>s->select(x P)</code>	collection of s elements x satisfying P
<code>s->reject(P)</code>	collection of s elements not satisfying P
<code>s->reject(x P)</code>	collection of s elements x not satisfying P
<code>s->collect(e)</code>	collection of elements x.e for $x : s$
<code>s->collect(x e)</code>	collection of elements x.e for $x : s$
<code>s->selectMaximals(e)</code>	collection of s elements x which have maximal x.e values
<code>s->selectMinimals(e)</code>	collection of s elements x which have minimal x.e values

Table 7: Selection operators

<i>Expression</i>	<i>Type</i>	<i>Element type</i>
$x + y$	String (if one of x, y is a string), otherwise LCNS	–
$x - y$	LCNS of the types	–
$x * y$	LCNS of the types	–
x / y	LCNS of the types	–
$x \bmod y$	int	–
$x.\text{sqr}$	double	–
$x.\text{sqrt}$	double	–
$x.\text{floor}$	int	–
$x.\text{round}$	int	–
$x.\text{ceil}$	int	–
$x.\text{abs}$	double	–
$x.\text{exp}$	double	–
$x.\text{log}$	double	–
$x.\text{pow}(y)$	double	–
$x.\text{sin}$	double	–
$x.\text{cos}$	double	–
$x.\text{tan}$	double	–
$\text{Integer.subrange}(st,en)$	Sequence	int

Table 8: Numeric operators types

3.3 Differences to OCL

UML-RSDS supports most features of the OCL 2.4 standard library [19]. Omitted are:

- `OclAny`, `OclVoid`, `OclInvalid`: omitted due to absence of suitable semantics. The implementation of OCL in UML-RSDS (and for B, Z3 and code translations) has a two-valued logic. Association ends of multiplicity 0..1 are treated as sets (or sequences, if ordered) of maximum size 1. There are no explicit null or invalid elements. Null elements cannot be defined, and are not permitted to be members of association ends or other collections. Thus a test for $e == \text{null}$ is $\text{Set}\{e\} \rightarrow \text{size}() = 0$. The `oclIsUndefined()` operator has been introduced from version 1.7.
- `OclMessage`: replaced by invocation instance expressions on sequence diagrams.
- `UnlimitedNatural`: not needed for internal use within models.
- `Integer`, `Real`: replaced by `int`, `long`, `double`. The reason for this is to ensure that the specification semantics of these types corresponds to their semantics in the generated executable (in Java and C#) and in B. A `long` (64-bit) integer type has also been added.
- `max`, `min` between two values: expressed instead by generalised `max`, `min` operations on collections. These apply also to collections of strings.
- `oclIsTypeOf(T)`, `flatten()` for arbitrary nested collections, and `equalsIgnoreCase()` (to be included).
- `closure()` is only included in the form $objs \rightarrow \text{closure}(r)$ or $objs.r \rightarrow \text{closure}()$ where r is an association role name defined for elements of $objs$. r must be many-valued.
- xor logical operator: this can be expressed using the other logical operators. The operators *and*, *implies* are written as `&`, `=>`. The if-then-else-endif operator is included from version 1.8.
- Set, sequence and string subtraction uses the `-` operator, extending OCL in the case of sequences and strings.

<i>Expression</i>	<i>Type</i>	<i>Element type</i>
<i>Set</i> {}	Set	-
<i>Sequence</i> {}	Sequence	-
<i>Set</i> { x_1, x_2, \dots, x_n }	Set	LCS of types of x_1 to x_n
<i>Sequence</i> { x_1, x_2, \dots, x_n }	Sequence	LCS of types of x_1 to x_n
<i>s</i> -> <i>including</i> (<i>x</i>)	Type of <i>s</i>	LCS of type of <i>x</i> and element type of <i>s</i>
<i>s</i> -> <i>excluding</i> (<i>x</i>)	Type of <i>s</i>	Element type of <i>s</i>
<i>s</i> - <i>t</i>	Type of <i>s</i>	Element type of <i>s</i>
<i>s</i> -> <i>prepend</i> (<i>x</i>)	Sequence	LCS of type of <i>x</i> , element type of <i>s</i>
<i>s</i> -> <i>append</i> (<i>x</i>)	Sequence	LCS of type of <i>x</i> , element type of <i>s</i>
<i>s</i> -> <i>count</i> (<i>x</i>)	Integer (int)	-
<i>s</i> -> <i>indexOf</i> (<i>x</i>)	Integer (int)	-
<i>x</i> - <i>y</i>	Type of <i>x</i>	Element type of <i>x</i>
$x \setminus y$	Sequence if <i>x</i> , <i>y</i> ordered, otherwise Set	LCS of element types of <i>x</i> , <i>y</i>
$x \setminus y$	Type of <i>x</i>	Element type of <i>x</i>
$x \hat{=} y$	Sequence	LCS of element types of <i>x</i> and <i>y</i>
<i>x</i> -> <i>union</i> (<i>y</i>)	Same as $x \setminus y$	Same as $x \setminus y$
<i>x</i> -> <i>intersection</i> (<i>y</i>)	same as $x \setminus y$	same as $x \setminus y$
<i>x</i> -> <i>symmetricDifference</i> (<i>y</i>)	Set	LCS of element types of <i>x</i> , <i>y</i>
<i>x</i> -> <i>any</i> ()	Element type of <i>x</i>	-
<i>x</i> -> <i>subcollections</i> ()	Set	Type of <i>x</i>
<i>x</i> -> <i>reverse</i> ()	Sequence	Element type of <i>x</i>
<i>x</i> -> <i>front</i> ()	Sequence	Element type of <i>x</i>
<i>x</i> -> <i>tail</i> ()	Sequence	Element type of <i>x</i>
<i>x</i> -> <i>first</i> ()	Element type of <i>x</i>	-
<i>x</i> -> <i>last</i> ()	Element type of <i>x</i>	-
<i>x</i> -> <i>sort</i> ()	Sequence	Element type of <i>x</i>
<i>x</i> -> <i>sortedBy</i> (<i>e</i>)	Sequence	Element type of <i>x</i>
<i>x</i> -> <i>flatten</i> ()	Type of <i>x</i>	Element type of elements of <i>x</i>
<i>x</i> -> <i>sum</i> ()	String if any element of <i>x</i> is a String, otherwise least common numeric supertype of <i>x</i> element types	-
<i>x</i> -> <i>prd</i> ()	Least common numeric supertype of elements of <i>x</i>	-
<i>x</i> -> <i>max</i> ()	Element type of <i>x</i>	-
<i>x</i> -> <i>min</i> ()	Element type of <i>x</i>	-
<i>x</i> -> <i>asSet</i> ()	Set	Element type of <i>x</i>
<i>x</i> -> <i>asSequence</i> ()	Sequence	Element type of <i>x</i>
<i>s</i> -> <i>isUnique</i> (<i>e</i>)	Boolean	-
<i>x</i> -> <i>isDeleted</i> ()	Boolean	-

Table 9: Collection operators types

<i>Expression</i>	<i>Type</i>	<i>Element type</i>
<code>s->select(P)</code>	Type of s	Element type of s
<code>s->select(x P)</code>	Type of s	Element type of s
<code>s->reject(P)</code>	Type of s	Element type of s
<code>s->reject(x P)</code>	Type of s	Element type of s
<code>s->collect(e)</code>	Sequence	Type of e
<code>s->collect(x e)</code>	Sequence	Type of e
<code>s->selectMaximals(e)</code>	Type of s	Element type of s
<code>s->selectMinimals(e)</code>	Type of s	Element type of s
<code>s->unionAll(e)</code>	Sequence if s type and e type Sequence, otherwise Set	Element type of e
<code>s->intersectAll(e)</code>	Set	Element type of e
<code>s->flatten()</code>	Type of s	Element type of s elements

Table 10: Selection operators types

- OrderedSet and Bag are omitted: both can be represented by sequences.
- Tuples and the cartesian product of two collections: very rarely used in modelling. The same effect can be achieved by introducing a new class with many-one associations directed to the tuple component entities, and similarly for ternary or higher-arity associations.
- *collectNested* in OCL is the same as *collect* in UML-RSDS, ie., no flattening or merging of duplicated elements takes place.
- The *one* operator is expressed by *exists1*.
- The *any* operator is available only in the form *col*→*any*().

substring and *subsequence* are combined into a single *subrange* operation.

Sets can be written with the notation $\{x_1, \dots, x_n\}$ instead of *Set*{ x_1, \dots, x_n }. Intersection and union can use the mathematical operators \cap and \cup written as \wedge and \vee . Likewise, subtraction on collections is written using $-$.

Additional to standard OCL are the operators *s*→*isDeleted*(), *s*→*unionAll*(*e*), *s*→*intersectAll*(*e*), *s*→*selectMinimals*(*e*), *s*→*selectMaximals*(*e*) and *s*→*subcollections*(), and the *Integer.subrange*(*i*,*j*) constructor.

Qualified or composition associations and association classes are not yet fully supported: Java, C#, C++ and B can be defined for these associations, but their values cannot be stored in or retrieved from models. Only binary associations are supported.

4 Operations

Operations can be added to classes using the *Modify* option, and the edit class dialog (Figure 7) and the operation definition dialog (Figure 8).

In the operation dialog both the precondition and postcondition predicate should be entered, "true" can be entered to denote no constraint.

Query operations must have a result type, and some equation *result* = *e* as the last conjunct in each conditional case in their postcondition (as in Figure 8 and the factorial example below). Update operations normally have no return type, but this is possible. A typical form of query operation has a series of if-then cases

(Cond1 => P1) & ... & (Cond_n => P_n)

These are mapped to a design of the form

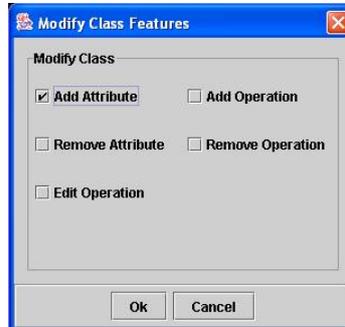


Figure 7: Edit class dialog

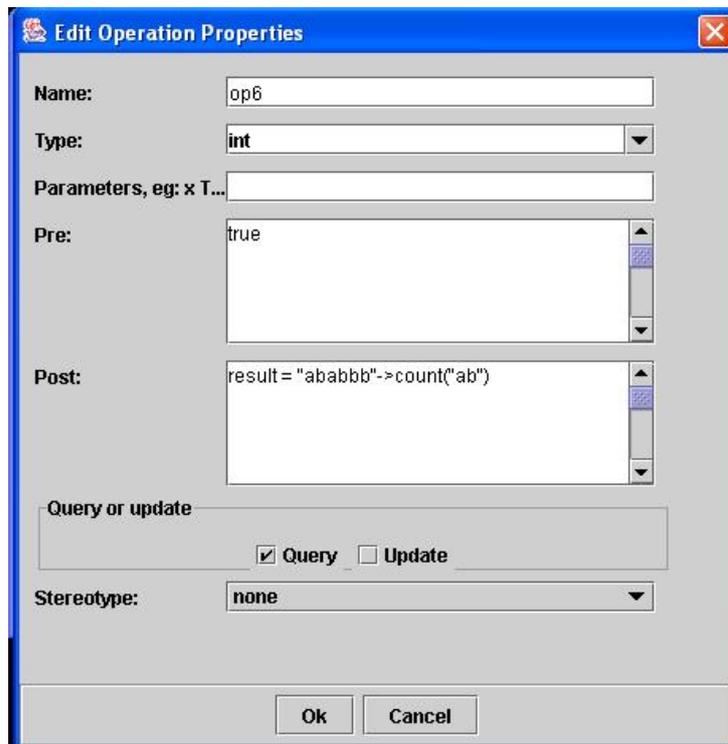


Figure 8: Operation definition dialog

```

if Cond1' then P1*
else if ...
else if Condn' then Pn*

```

In contrast, if an operation does not have the *query* stereotype, then these cases are sequenced:

```

if Cond1' then P1* ;
... ;
if Condn' then Pn*

```

Operations can be defined recursively, for example:

```

factorial(i : int) : int
pre: i >= 0
post:
  (i <= 1 => result = 1) &
  (i > 1 => result = i*factorial(i-1))

```

Recursive update operations cannot be expressed in B, but recursive query operations can be.

Operations can be called by the notation *obj.op(params)* as usual. Such calls are expressions, in the case of query operations, and statements in the case of update operations. If *op* is a query operation then this expression can be used within other expressions as a value of its declared return type. Update operations should not be used as values or in contexts (such as conditional tests or constraint antecedents) where a pure value expression is expected.

Query operations with at least one argument, or instance scope query operations with no arguments can be stereotyped as *cached*, which means that caching is used for their implementation: the results of operation applications are stored in order to avoid recomputation of results. This can make a substantial difference to efficiency in the case of recursive operations. Only operations owned by a class (not an interface or use case) can be cached.

For example, for the function defined as:

```

f(x : Integer) : Integer
pre: x >= 0
post:
  ( x = 0 => result = 1 ) &
  ( x = 1 => result = 2 ) &
  ( x > 1 => result = f(x-1) + f(x-2) )

```

there are 331,160,281 calls to *f* when evaluating *f*(40) without caching, and an execution time of 7.5s. With caching there are only 41 calls, and an execution time of 10ms. However, the user should ensure that cached behaviour is really what is intended – if the operation result could legitimately change if the operation is invoked at different times with the same argument values then this stereotype should not be used.

For C++, caching is restricted to operations with at most one parameter. Caching is not yet supported by the C code generator.

5 Use cases

Use cases of a system represent the externally available services that it provides to clients, either directly to users, or to other parts of a larger system.

Two forms of use case can be defined in UML-RSDS:

- EIS use cases – simple operations such as creating an instance of an entity, modifying an instance, etc, intended to be implemented using an EIS platform such as Java Enterprise Edition. These are described in Section 10.

- General use cases – units of functionality of arbitrary complexity, specified by constraints (pre and postconditions of the use case). These are particularly used to specify model transformations (Section 6). They are visually represented as ovals in the class diagram. They may have their own attributes and operations, since use cases are classifiers in UML. Use case invariants can also be defined for general use cases.

Figure 9 shows the UML-RSDS dialog for creating use cases.

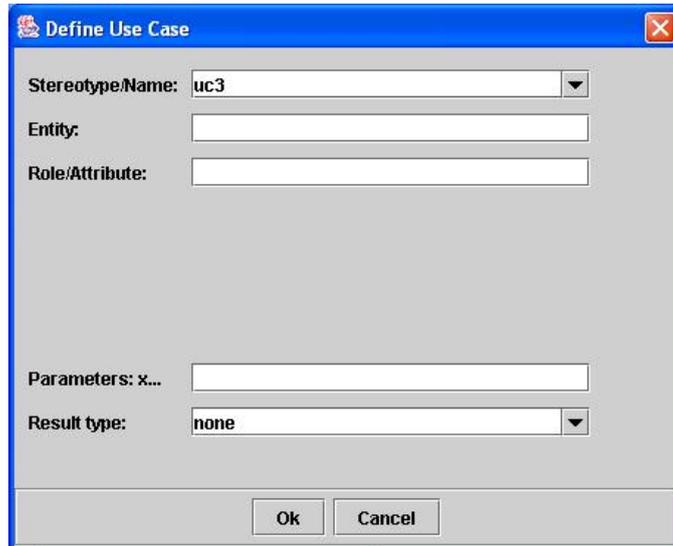


Figure 9: Use case dialog

Figure 10 shows the UML-RSDS dialog for modifying general use cases.

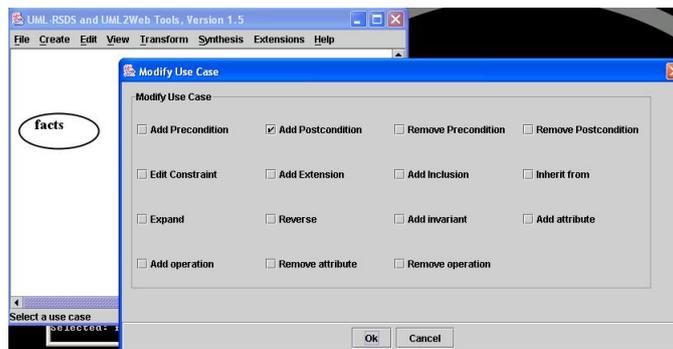


Figure 10: Use case edit dialog

In addition, an interactive editor has been introduced (from version 1.6) for modifying use cases (Figure 11). In this editor, postconditions are written as:

```
E::
Pre => Post
```

for a constraint with context entity E, assumption Pre and conclusion Post. The syntax:

```
::
Pre => Post
```

is used if there is no context entity (type 0 constraint). To check the syntax of a constraint, position the cursor after its end, and select option Check. Constraint text can be copy/pasted within the use case, and between use cases (with multiple editors open).

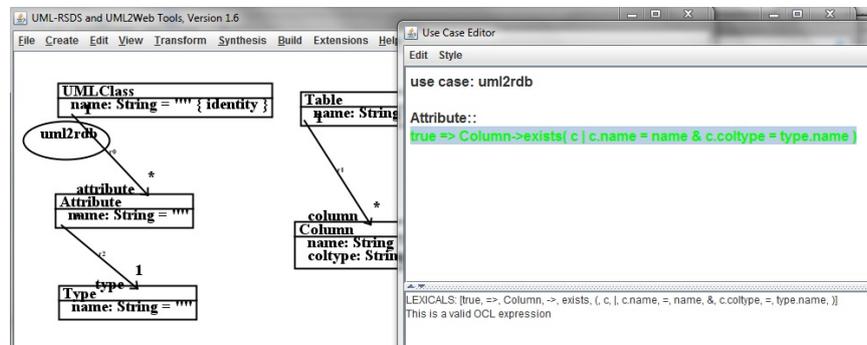


Figure 11: Use case editor

General use cases can be structured and composed using several different mechanisms:

- *extend*: one use case can be declared as an extension of another, to provide additional functionality to the base use case.
- *include*: one use case can be included in one or several other use cases, to define a sub-operation or process of the base use cases. This can be used to factor out common sub-processes.
- Parameterisation: a use case can be given typed parameters to make it generic and reusable.
- Inheritance: a use case can inherit another, to specialise its effect. This is not yet supported by the tools.

Basic use cases (use cases without activities or included use cases) can be composed together using *extend*. This has the effect of conjoining the postconditions of the extension and extended use cases to form the postconditions of the composed use case, and likewise for preconditions and invariants. It acts like a conjunction combinator of use cases (assumed to be mutually consistent). A new use case with the conjoined postconditions is added to the system model.

More complex use cases/services can be composed from these basic use cases using *include* and use case activities. If use cases uc_1, \dots, uc_n are included in use case uc , in that order, then the composed effect of uc includes executing uc_1, \dots, uc_n after uc 's own effect. Alternatively, an activity for uc can invoke the included use cases in complex algorithms.

The elementary example of a “Hello world” program can be defined as a single postcondition

```
"Hello world!"->display()
```

of a use case on an empty metamodel.

A simple example of a use case for our teaching system could have the single postcondition (with no entity context)

```
Lecturer->exists( k | k.name = "Mike" )
```

which creates a lecturer object with name “Mike”.

The option *Use Case Dependencies* on the *View* menu allows the dependencies between use cases and classes to be seen: a red dashed line is drawn from a use case to each class whose data it may modify, and a green dashed line is drawn from each class read by the use case, to the use case.

6 Model transformations

Model transformations in UML-RSDS are defined by UML use cases, the postconditions of these use cases define the transformation effect. Both the source and target metamodels are represented as parts (possibly disjoint and unconnected parts) of the same class diagram. Each use case postcondition constraint C_n usually has context some source metamodel entity E , and defines how elements of E are mapped to target model elements.

Figure 12 shows an example where the source metamodel consists of a single entity A with attribute $x : int$, and the target metamodel consists of entity B with attribute $y : int$, and there is a use case $a2b$ with a single postcondition constraint:

```
A::  
  x > 0 => B->exists( b | b.y = x*x )
```

on context A . The constraint specifies that a new B object b is created for each $ax : A$ with $ax.x > 0$, and $b.y$ is set to the square of $ax.x$.

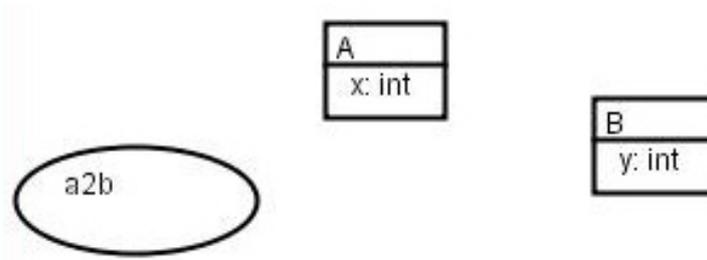


Figure 12: A to B transformation

Assumptions of a transformation can also be defined, as precondition constraints of its use case. In this example we assume that the target model is initially empty, ie:

```
B.size = 0
```

A valid transformation invariant constraint could be:

```
B::  
  A->exists( a | a.x > 0 & y = a.x*a.x )
```

on B . This expresses that every B must have been derived from a corresponding A .

The following steps are taken to define a transformation:

- Define the metamodels, as a single class diagram or separate class diagrams.
- Create a general use case, for the transformation.
- Edit the use case, adding pre and postcondition and invariant constraints as required. The order of the postcondition constraints determines the order of execution of the corresponding code segments.
- On the synthesis menu, select Generate Design to analyse the use case and generate a design-level description as operations and activities.
- Select Build Java (Java 4, 6 or 7) to produce an executable version of the transformation, which will be contained in the *output* directory files *Controller.java*, *GUI.java*, *ControllerInterface.java* and *SystemTypes.java*.

- The transformation implementation can be compiled using `javac Controller.java, javac GUI.java` and then executed by the command `java GUI`. This opens up a simple dialog to load the input model and execute individual transformations.

Input models are specified in a file `in.txt`, output models are created in `out.txt`, together with XMI versions in `xsi.txt`.

- Executable implementation in the other supported languages is also possible.

The text format for models in `in.txt` and `out.txt` is very simple, consisting of lines of text of three possible forms:

`e : T`

declaring that *e* is an instance of concrete entity *T*,

`e.f = v`

declaring that the attribute or 1-multiplicity role *f* of *e* has value *v*, and

`obj : e.role`

declaring that *obj* is a member of the many-valued role *role* of *e*. For sequence-valued roles, the order in which these membership assertions occur in the file is the order of elements in *role*. There must be spaces around the = and : operators (and no spaces around .).

Instances of entity types with stereotypes *auxiliary*, *external* or *externalApp* are not saved to `out.txt`.

For example, we could have the example model

```
a1 : A
a1.x = 5
a2 : A
a2.x = 3
b1 : B
b1.y = 2
```

for our elementary transformation example. Running the transformation on this input model produces the following output model:

```
a1 : A
a1.x = 5
a2 : A
a2.x = 3
b1 : B
b1.y = 2
b2 : B
b2.y = 25
b3 : B
b3.y = 9
```

Qualified associations and association classes cannot be represented in models, however these could be used for internal processing of a transformation, as auxiliary metamodel elements.

Example transformations are given on the UML-RSDS website, in the *zoo* subdirectory, eg., *mmuml2Ca*, *mmcra17* and *mmpn2sc*. The UML to C translator is a large transformation example, its specifications are included in the *mmuml2Ca* and *mmuml2Cb* metamodel files.

Four categories of postconditions *Cn* are distinguished:

1. Type 0: *Cn* has no context classifier, for example

```

::
  "Hello world!"->display()

```

The constraint dialog field for the context entity is left empty when defining such constraints.

2. Type 1: a context classifier E exists, and $wr(Cn)$, the set of entity types and features written by Cn , is disjoint from $rd(Cn)$, the set that it reads (write and read frames are defined in Table 25). Such Cn are implemented by a for-loop over E . For example, the constraint

```

A::
  x > 0 => B->exists( b | b.y = x*x )

```

given above is type 1. E (in this case A) is specified in the context entity field of the constraint dialog (Figure 6).

3. Type 2: $wr(Cn)$ has common elements with $rd(Cn)$, but the context classifier E itself is not in $wr(Cn)$, nor are other entity types/features in the assumptions of Cn . These are implemented by a simplified fixed-point iteration over E .
4. Type 3: as type 2, but $wr(Cn)$ includes the context classifier or other assumption entity types/features. For example, to delete all A objects with negative x values, we could write:

```

A::
  x < 0 => self->isDeleted()

```

on context A . For such constraints a general fixed-point implementation is used, however it is often the case that this can be optimised by omitting checks that the conclusion of Cn already holds (in the above example, the conclusion obviously never holds for some $self : A$ before the application of the constraint). A dialog box is presented asking for confirmation of this optimisation, during the Generate Design step. Answer y to confirm the optimisation.

Distinct postcondition constraints should be ordered so that Cm precedes Cn if $wr(Cm)$ has common elements with $rd(Cn)$.

For the above example this means that the correct order is:

```

A::
  x < 0 => self->isDeleted()

```

```

A::
  x > 0 => B->exists( b | b.y = x*x )

```

since the first writes A and the second reads A : this order ensures that the invariant identified above actually is maintained by the transformation.

Constraint analysis during the Generate Design step identifies any potential problems with the ordering of constraints.

Pre-state expressions in use case postcondition constraints refer to the values of the entity types at the start of the execution phase for the constraint (in the case of entity types $E@pre$ defined as the context of the constraint), or the values of features of individual objects at the start of application of individual applications of the constraint (for features $f@pre$). A prestate feature $obj.f@pre$ should only be applied to instances obj of a prestate entity $E@pre$, unless E itself is not written by the constraint. It is bad practice to both read $x.f@pre$ and write $x.f$ for the same object x in the same constraint, this can lead to a failure of confluence, and to constraint implementations that do not establish the constraint (or at least, not those parts of the constraint that use $f@pre$).

$f@pre$ can be used for features f of the context classifier of the constraint, or $x.f@pre$ for features of one of the outer-level variables x of the constraint (introduced in its antecedent), but

cannot be used for features of variables x of $exists(x \mid P)$, etc, $self$ of $select(P)$, etc, or other internal variables within a constraint.

An example of the use of prestate entities and features is the computation of the one-step composition of pairs of edges in a graph: if $e1$ and $e2$ are joined head-to-tail at the same node, then create a new edge that directly links $e1$'s source to $e2$'s target (but do not iterate this process any further):

```
::
  e1 : Edge@pre & e2 : Edge@pre & e2.src@pre = e1.trg@pre =>
    Edge->exists1( e3 | e3.src = e1.src@pre & e3.trg = e2.trg@pre )
```

The iterations of $e1$ and $e2$ over $Edge@pre$ restricts the composition step to operate only upon pre-existing edges, not upon any $e3$ created by the step itself. The use of $src@pre$ and $trg@pre$ emphasises that the features of pre-existing edges are not changed by the constraint. Taken together, these annotations allow a simple double for-loop implementation of the constraint (because its set of written entities and features are disjoint from its set of read entities and features) rather than a fixpoint implementation.

Variables may also be used to express *let* definitions, by placing an equation $v = exp$ in the assumption of a constraint. v can then be used in place of exp in the conclusion. Such v should only be used in read-only contexts after their definition.

6.1 Bidirectional transformations

Support for *bidirectional* transformations is provided by the option to *reverse* a use case (on the edit use case menu). This option generates a new use case whose postconditions are derived as inverses of the postconditions of the original use case: these inverse constraints are often also invariants of the original use case. A type 0 or type 1 constraint

$$S_i \rightarrow \text{forall}(s \mid SCond(s) \Rightarrow T_j \rightarrow \text{exists}(t \mid TCond(t) \ \& \ P_{i,j}(s, t)))$$

is inverted to a constraint of the form

$$T_j \rightarrow \text{forall}(t \mid TCond(t) \Rightarrow S_i \rightarrow \text{exists}(s \mid SCond(s) \ \& \ P_{i,j}^{\sim}(s, t)))$$

where the predicates $P_{i,j}(s, t)$ define the features of t from those of s , and are invertible: an equivalent form $P_{i,j}^{\sim}(s, t)$ should exist, which expresses the features of s in terms of those of t .

In the general case of entity merging/splitting the inverse of

$$S_{i1} \rightarrow \text{forall}(s1 \mid \dots S_{im} \rightarrow \text{forall}(sn \mid SCond(s1, \dots, sn) \Rightarrow \\ T_{j1} \rightarrow \text{exists}(t1 \mid \dots T_{jm} \rightarrow \text{exists}(tm \mid TCond(t1, \dots, tm) \ \& \\ P(s1, \dots, sn, t1, \dots, tm) \dots)) \dots)$$

is:

$$T_{j1} \rightarrow \text{forall}(t1 \mid \dots T_{jm} \rightarrow \text{forall}(tm \mid TCond(t1, \dots, tm) \Rightarrow \\ S_{i1} \rightarrow \text{exists}(s1 \mid \dots S_{im} \rightarrow \text{exists}(sn \mid SCond(s1, \dots, sn) \ \& \\ P^{\sim}(s1, \dots, sn, t1, \dots, tm) \dots)) \dots)$$

Tables 11 and 12 show examples of inverses P^{\sim} of predicates P .

For bx , we use $stat_{LC}(P)$ for the procedural interpretation of expressions P , instead of $stat(P)$. This represents *least-change* semantics: an update is only performed if P does not already hold, or if the update would make no change in the case P holds.

$stat_{LC}(v = e)$ is the same as $stat(v = e)$, $v' := e'$.

$stat_{LC}(v : e)$ is the same as $stat(v : e)$ for sets e , but for sequences it is

if $v' : e'$ then skip else $stat(v : e)$

$P(s,t)$	$P^\sim(s,t)$	Condition
$t.g = s.f$	$s.f = t.g$	
$t.g = s.f.sqrt$	$s.f = t.g.sqr$	f, g non-negative
$t.g = s.f.sqr$	$s.f = t.g.sqrt$	f, g non-negative
$t.g = s.f.exp$	$s.f = t.g.log$	g positive
$t.g = s.f.log$	$s.f = t.g.exp$	f positive
$t.g = s.f.pow(x)$	$s.f = t.g.pow(1.0/x)$	x non-zero, independent of $s.f$
$t.g = s.f.sin$	$s.f = t.g.asin$	$-1 \leq t.g \leq 1$
$t.g = s.f.cos$	$s.f = t.g.acos$	$-1 \leq t.g \leq 1$
$t.g = s.f.tan$	$s.f = t.g.atan$	
$t.b2 = not(s.b1)$	$s.b1 = not(t.b2)$	Boolean attributes $b1, b2$
$t.g = K * s.f + L$ Numeric constants K, L $K \neq 0$	$s.f = (t.g - L)/K$	f, g numeric
$t.g = s.f + SK$	$s.f = t.g.subrange(1, t.g.size - SK.size)$	String constant SK
$t.g = SK + s.f$	$s.f = t.g.subrange(SK.size + 1, t.g.size)$	String constant SK
$t.r = s.f \rightarrow characters()$	$s.f = t.r \rightarrow sum()$	String feature f , sequence r
$R(s,t) \ \& \ Q(s,t)$	$R^\sim(s,t) \ \& \ Q^\sim(s,t)$	

Table 11: Inverse of predicates

$P(s,t)$	$P^\sim(s,t)$	Condition
$t.rr = s.r \rightarrow reverse()$	$s.r = t.rr \rightarrow reverse()$	r, rr ordered association ends
$t.rr = s.r \rightarrow last()$	$s.r \rightarrow last() = t.rr$	rr single-valued, r ordered
$t.rr = s.r \rightarrow first()$	$s.r \rightarrow first() = t.rr$	rr single-valued, r ordered
$t.rr = s.r \rightarrow including(s.p)$	$s.r = t.rr \rightarrow front() \ \&$	rr, r ordered association ends
$t.rr = s.r \rightarrow append(s.p)$	$s.p = t.rr \rightarrow last()$	p 1-multiplicity end
$t.rr = s.r \rightarrow prepend(s.p)$	$s.r = t.rr \rightarrow tail() \ \&$	rr, r ordered association ends
	$s.p = t.rr \rightarrow first()$	p 1-multiplicity end
$t.rr = Sequence\{s.p1, s.p2\}$	$s.p1 = t.rr \rightarrow at(1) \ \&$	rr ordered association end
	$s.p2 = t.rr \rightarrow at(2)$	$p1, p2$ 1-multiplicity ends
$t.rr = s.r \rightarrow select(P)$	$s.r \rightarrow excludesAll(x \mid P(x) \ \&$	r, rr
	$t.rr \rightarrow excludes(x))$	set-valued
	$\rightarrow includesAll(t.rr \rightarrow select(P))$	
$t.rr = s.r \rightarrow reject(P)$	$s.r \rightarrow excludesAll(x \mid not(P(x)) \ \&$	r, rr
	$t.rr \rightarrow excludes(x))$	set-valued
	$\rightarrow includesAll(t.rr \rightarrow reject(P))$	
$t.rr = s.r \rightarrow sort()$	$s.r = t.rr \rightarrow asSet()$	r set-valued, rr ordered
$t.rr = s.r \rightarrow asSequence()$		
$t.rr = s.r \rightarrow sort()$	$s.r = (s.r \cap t.rr) \ \wedge (t.rr - s.r)$	r sequence-valued
$t.rr = s.r \rightarrow asSet()$	$s.r = (s.r \cap t.rr) \ \wedge (t.rr - s.r) \rightarrow asSequence()$	r sequence-valued

Table 12: Inverse of predicates on associations

Similarly for $<:$, $/:$ and $/<:$.

$stat_{LC}(P \& Q)$ is $stat_{LC}(P)$; $stat_{LC}(Q)$ under the assumption that $P \Rightarrow [stat_{LC}(Q)]P$. This is ensured if $wr(Q)$ is disjoint from $wr(P)$ and $rd(P)$.

$stat_{LC}(if\ C\ then\ P\ else\ Q\ endif)$ is

```
if C
then statLC(P)
else statLC(Q)
```

under the assumptions $C \Rightarrow [stat_{LC}(P)]C$ and $\neg C \Rightarrow [stat_{LC}(Q)]\neg C$. These are ensured if $wr(P)$ and $wr(Q)$ are disjoint from $var(C)$.

For $s \rightarrow forAll(x \mid P)$, $stat_{LC}$ is defined as *for* $x : s$ *do* $stat_{LC}(P)$.

For existential quantifiers $E \rightarrow exists(e \mid P_1 \& \dots \& P_n)$, their $stat_{LC}$ effect only creates a new e in cases where there is no existing $e : E$ that satisfies P partially or completely. In the case of partial satisfaction the updates only for the unsatisfied conjuncts are carried out. This can also be written as $E \rightarrow existsLC(e \mid P_1 \& \dots \& P_n)$.

If E has a primary key pk and a conjunct P_i is of the form $e.pk = value$, then $stat_{LC}(E \rightarrow exists(e \mid P_1 \& \dots \& P_n))$ is

```
var e : E;
e := createByPKE(value);
statLC(Pred)
```

Where $Pred$ is $P_1 \& \dots \& P_n$ with P_i omitted.

Otherwise, $stat_{LC}(E \rightarrow exists(e \mid P_1 \& \dots \& P_n))$ has the form:

```
var e : E;
var eset : Set(E);
eset := E.allInstances();
if eset->isEmpty()
then
  e := createE();
  statLC(P1 & ... & Pn)
else
  (e := eset->any();
  eset := eset->select(P1);
  if eset->isEmpty()
  then
    e := createE();
    statLC(P1 & ... & Pn)
  else
    (e := eset->any();
    eset := eset->select(P2);
    if eset->isEmpty()
    then
      statLC(P2 & ... & Pn)
    else
      ... case for 3 ... ) )
```

The general case for $k \geq 2$, $k < n$ is

```
e := eset->any();
eset := eset->select(Pk);
if eset->isEmpty()
then
  statLC(Pk & ... & Pn)
else
  ... case for k+1 ...
```

For n , if P_k is an assignment $result = e$, then the code of the case is simply $e := eset \rightarrow any(); result := e$. Otherwise it is

```
e := eset->any();
eset := eset->select(Pn);
if eset->isEmpty()
then
  statLC(Pn)
else
  e := eset->any()
```

By reusing $e : E$ instances where possible, the redundant creation of instances is avoided, however this also introduces the possibility of conflicts where one target instance is required to have conflicting attribute values to satisfy a constraint wrt 2 source instances.

$stat_{LC}$ produces code that is around 2 to 5 times slower than $stat$ code (for Java 4).

The inversion procedure can be generalised by using *view complements* as in [16]. In particular, any $P(s, t)$ which computes a sequence-valued feature $t.g = v(s.f)$ from sequence-valued $s.f$ by recursion on the elements of $s.f$ can be reversed to $s.f = u(s.f@pre, t.g)$ for a recursively-defined *view update* function u .

If v is:

$$v(x) = \text{if } x.size = 0 \text{ then Sequence}\{\}$$

$$\quad \text{else if } P_1(x.first, x.tail) \text{ then Sequence}\{v_1(x.first)\} \hat{\ } v(x.tail)$$

$$\quad \text{else if } \dots$$

$$\quad \text{else } v(x.tail) \text{ endif}$$

Then u is defined by:

$$u(x, y) = \text{if } x.size = 0 \text{ then Sequence}\{\}$$

$$\quad \text{else if } P_1(x.first, x.tail) \text{ then Sequence}\{u_1(x.first, y.first)\} \hat{\ } u(x.tail, y.tail)$$

$$\quad \text{else if } \dots$$

$$\quad \text{else } u(x.tail, y.tail) \text{ endif}$$

where u_i is an update function inverse to v_i .

This definition includes *select*, *reject* and *collect* cases on sequences, and other functions on sequences, eg., that omit/include elements based on their position in the sequence.

General use cases may have static attributes *att*, defined using the *add attribute* option on the edit use case menu. These attributes may be used to define data used by all of the constraints of the use case but which is not present in the metamodel it operates upon. In the generated code, the attributes become local attributes of the class Uc corresponding to the use case. They should be referred to as $Uc.att$.

Use cases may have static operations, in the implementation these also become operations of the class Uc . Use case operations can be called from within the use case postconditions. They have no instance context so must be supplied with all necessary data as parameters. They should be referred to as $Uc.op(pars)$.

6.2 Importing ATL modules

ATL [5] is one of the leading hybrid model transformation languages. It has a specialised syntax and a more imperative orientation compared to UML-RSDS.

Transformation definitions can be given in ATL syntax and imported into UML-RSDS using the File/Load transformation menu option ‘Load standard ATL’. A class diagram expressing the source and target metamodels should already exist and be loaded into UML-RSDS before selection of the ‘Load standard ATL’ option. The file *mm.atl* is loaded. Source entity types should have the stereotype ‘source’, and target entity types should have the stereotype ‘target’.

<i>ATL</i>	<i>UML-RSDS</i>
Module	Use case
Matched rule	Use case postcondition
Lazy rule	Operation invoked from postconditions
Unique lazy rule	Operation using caching to avoid recreating existing objects
Implicit rule invocation	Object lookup using object indexing or traces
Attribute helper	Local attribute of use case
Function helper	Operation
Action blocks of matched rules	Operation with activity, invoked from postcondition representing the rule
Action blocks of called/lazy rules	Operation with activity, invoked from operation representing the rule

Table 13: Correspondence of ATL and UML-RSDS

Table 13 shows the mapping of ATL (standard mode) elements to UML-RSDS. We assume that all rule inheritance has been statically expanded out so that only leaf rules exist.

The expression language of ATL is closely based upon OCL, as is that of UML-RSDS. There are only two significant points of difference in the languages: (i) in UML-RSDS collections are either sequences or sets, and bags and ordered sets are represented in terms of sequences; (ii) 0..1-multiplicity association ends r are treated as sets (or sequences) of maximum size 1. Thus $r.oclIsUndefined()$ in ATL is expressed by $r \rightarrow isEmpty()$ in UML-RSDS, and the value of such r when defined is $r \rightarrow any()$ in UML-RSDS. OCL operators *div*, *one*, *iterate*, $\langle \rangle$, *let* occurring in ATL expressions must also be translated to UML-RSDS OCL. ATL ordered sets, bags, tuples and maps are not currently translated to UML-RSDS.

We denote the UML-RSDS expression equivalent of an ATL expression e by e' .

Given an ATL module

```

module M;
create OUT : T from IN : S;
rule r1
{ from s1 : S1, ..., sm : Sm (SCond)
  to t1 : T1 (TCond1), ..., tk : Tk (TCondk)
  do (Stat)
}
...
rule rn { ... }

```

the following steps are taken to produce a UML-RSDS specification:

1. A use case M' is defined, operating on metamodels $IN\$S$ and $OUT\$T$, where each entity or enumerated type E in S is renamed to $IN\$E$ to form the input language $IN\$S$ and likewise for T .
2. Each input and output entity type is given a new identity attribute (primary key) $\$id : String$. This is used to express 1–0..1 tracing relationships between the source and target models. The attributes only need to be added to the entity types that are maximal in the inheritance hierarchies of $IN\$S$ and $OUT\$T$. Distinct hierarchies have disjoint sets of $\$id$ values.
3. Each matched rule ri is represented by a postcondition constraint ri' of M' , operating on the first input entity type $IN\$S1$ of the rule, and with the other input variables as (implicitly) quantified LHS variables of ri' . The input conditions $SCond$ are directly translated to a

LHS condition $SCond'$ of ri' . Any **using** variables are added with their definitions to the LHS condition of ri' . The output variables become exists-quantified variables of the RHS of ri' .

4. The output conditions $TCondj$ are expressed as expressions $TCondj'$ as follows. An assignment

$$f \leftarrow e$$

to a feature f of output variable tj is interpreted as

$$tj.f = e'$$

where e' is the UML-RSDS interpretation of e . Any implicit type conversion from an expression p in e of a source entity type $SEnt$ to a target entity type $TEnt$ is expressed as

$$OUT\$TEnt[p.\$id]$$

in e' . This is the (default) target object corresponding to p . Likewise for implicit type conversions from $Set(SEnt)$ to $Set(TEnt)$ and from $Sequence(SEnt)$ to $Sequence(TEnt)$.

5. $thisModule.resolveTemp(s, t)$ becomes $TEnt[t + \text{"_"} + s.\$id]$ for $s : SEnt$ mapped by some rule to $TEnt$. This provides lookup for non-default target objects.
6. If an action block $Stat$ is specified, then a new update operation $opri(s2 : IN\$S2, \dots, sm : IN\$Sm, t1 : OUT\$T1, \dots, tk : OUT\$Tk)$ is introduced to $IN\$S1$, and this operation has activity given by the interpretation $Stat'$ of $Stat$ as a UML-RSDS activity.
7. The resulting RHS of ri' has predicate

$$OUT\$T1 \rightarrow exists(t1 \mid t1.\$id = \$id \ \& \ \dots \ \& \\ OUT\$Tk \rightarrow exists(tk \mid tk.\$id = \text{"tk_"} + \$id \ \& \ TCond1' \ \& \ \dots \ \& \ TCondk' \ \& \\ s1.opri(s2, \dots, tk) \dots)$$

8. Before mapping to UML-RSDS, the ATL rules are sliced and split if necessary so that rules do not read target entity types $TEnt$ until after all creations and updates of instances of $TEnt$ have been performed. Rules that read $TEnt$ and that precede (or are equal to) rules which create/update $TEnt$ instances are split into initial rules whose output bindings do not involve type conversions from input entity types to output types, followed (after all the original rules of the ATL module) by linking rules which perform the remaining bindings and the do actions of the original rules. All the initial rules are therefore (according to the UML-RSDS semantics) executed before any of the linking rules. Rules which do not involve type-conversion bindings do not need to be split. This corresponds to the ATL semantics in which a matching (target object creation) phase precedes initialisation (target object linking).
9. The translation of lazy and called rules r is similar, except that the translated constraint r' is used as the postcondition of an operation r of the first input entity type of the rule. Calls $thisModule.r(v1, \dots, vm)$ to the rule are interpreted as calls $v1.r(v2, \dots, vm)$ of this operation. The operation r is stereotyped as $\ll cached \gg$ in the case of unique lazy rules.

The ATL module is read from a text file in *output/mm.atl*.

The translation from ATL to UML-RSDS also includes update-in-place transformations. We adopt the convention that if the same declaration $tj : Tj$ occurs in both *from* and *to* clauses of an ATL rule, this means that tj is updated by the rule, not created by it. Then such variables do not have an $Tj \rightarrow exists(tj \mid tj.\$id = \$id \ \& \ \dots)$ quantifier in the corresponding UML-RSDS constraint conclusion, instead each reference $tj.f$ to a feature of tj in the rhs e of a binding $p \leftarrow e$ in the *to* clause is interpreted as $tj.f@pre$.

This permits translation of rules such as

```

rule rr {
  from s1 : S1, t1 : T1 (true)
  to t1 : T1
  ( r <- t1.r->including(s1) )
}

```

into:

```

IN$S1 ::
t1 : OUT$T1 =>
  t1.r = t1.r@pre->including(self)

```

Likewise, the class diagram refactoring transformation of [6] can be defined as:

```

rule r1 {
  from c : Entity, p : Property
  (c.specialisation->size() > 1 and
  c.specialisation.specific.ownedAttribute->includes(p) and
  c.ownedAttribute.name->excludes(p.name) and
  c.specialisation.specific->forall( g |
    g.ownedAttribute->exists( q | q.name = p.name and q.type = p.type ) ) )
  to c : Entity
  ( ownedAttribute <- c.ownedAttribute->including(p) )
  do
  { for ( p1 in c.specialisation.specific.ownedAttribute )
    { if (p1.name = p.name)
      { p1->isDeleted(); }
    }
  }
}

```

Adding p to $c.ownedAttribute$ implicitly removes it from the $ownedAttributes$ of its previous entity, due to the 0..1 multiplicity of the opposite association end.

Table 14 compares ATL and UML-RSDS as transformation languages. Generally, ATL is better suited to separate-models transformations such as migrations, whereas UML-RSDS has full support for update-in-place transformations such as refactorings.

	<i>ATL</i>	<i>UML-RSDS</i>
System modularisation	Modules, rules, libraries, helpers	Use cases, operations, constraints, classes
Rule modularisation	Inheritance, called rules	Called operations, operation inheritance
Rule scheduling/priorities	Implicit	Explicit ordering
Multiple input elements in rules	No	Yes
Direct update of input models	No	Yes
Transformation implementation	In ATL interpreter	In Java, C#, C++, C
Analysis	Runtime rule conflicts detected	Data dependency, confluence, determinacy, rule conflicts

Table 14: Comparison of ATL and UML-RSDS

An additional conversion, of refining mode ATL modules to UML-RSDS, is at an experimental stage and can be used by the option ‘Load refining ATL’. Refining mode (ATL 2010 version)

provides a restricted form of update-in-place semantics by executing rules to create updated or new source model elements, or to delete elements. The source model is read by the rules, and their effects are applied to the target model, which is initially a copy of the source [5].

Given a refining mode ATL module

```

module M;
create OUT : S refining IN : S;
rule r1
{ from s1 : S1, ..., sm : Sm (SCond)
  using { v1 : Typ1 = val1; ... }
  to t1 : T1 (TCond1), ..., tk : Tk (TCondk)
  do (Stat)
}
...
rule rn { ... }

```

the following steps are taken to produce a UML-RSDS specification:

1. A use case M' is defined, operating on metamodels $IN\$S$ and $OUT\$S$, these have copies $IN\$E$ and $OUT\$E$ of the classifiers and enumerated types E of S .
2. Each entity type is given a new identity attribute (primary key) $\$id : String$. This is used to express tracing relationships between the source and target (copy) models.
3. The first phase of M' copies the source model data to the target model.
4. Each matched rule ri is represented by a postcondition constraint ri' of the second phase of M' , operating on the first input entity type $IN\$S1$ of the rule, and with the other input variables as (implicitly) quantified LHS variables of ri' . The input conditions $SCond$ are directly translated to a LHS condition $SCond'$ of ri' , with $self$ replacing $s1$. Any **using** variables are added with their definitions to the LHS condition of ri' .
5. The output conditions $TCondj$ are expressed as expressions $TCondj'$ as follows. An assignment $f \leftarrow e$ to a feature f of output variable tj is interpreted as $tj.f = e'$ where e' is the UML-RSDS interpretation of e . For source object(s) e of class $IN\$E$, the interpretation is $OUT\$E[e.\$id]$, the corresponding copied object(s) in the target model.
6. If an action block $Stat$ is specified, then a new update operation $opri(s2 : S2, \dots, sm : Sm, t1 : T1, \dots, tk : Tk)$ is introduced to $S1$, and this operation has activity given by the interpretation $Stat'$ of $Stat$ as a UML-RSDS activity.
7. The resulting RHS of ri' has predicate

$$\begin{aligned}
&OUT\$T1 \rightarrow exists(t1 \mid t1.\$id = \$id \ \& \ \dots \ \& \\
&\quad OUT\$Tk \rightarrow exists(tk \mid tk.\$id = "tk_" + \$id \ \& \ TCond1' \ \& \ \dots \ \& \ TCondk' \ \& \\
&\quad s1.opri(s2, \dots, tk)) \dots
\end{aligned}$$

The semantics of $Ti \rightarrow exists(ti \mid ti.\$id = val$ in UML-RSDS is that ti is only created if there does not already exist a Ti object $Ti[val]$ with the given $\$id$ value – if there is such an object, it is selected and updated. The $\$id$ is a Key in terms of QVT [18].

8. The translation of lazy rules r is similar, except that the translated constraint r' is used as the postcondition of an operation rop of the first input entity type of the rule. Calls $thisModule.r(v1, \dots, vm)$ to the rule are interpreted as calls $v1.rop(v2, \dots, vm)$ of this operation. The operation rop is stereotyped as $\ll cached \gg$ in the case of unique lazy rules.
9. The special form of deletion rules:

```

rule r
{ from s1 : S1 (SCond)
  to drop
}

```

are interpreted as: $IN\$S1 ::$

$$SCond' \ \& \ \$id : OUT\$S1.\$id \ \& \ t1 = OUT\$S1[\$id] \Rightarrow t1 \rightarrow isDeleted()$$

which deletes the target object $t1$ corresponding to $s1$. Cascading delete is used in both ATL and UML-RSDS. Deletion rules are applied after all creation and update rules.

We have permitted multiple input elements in ATL rules, since these are representable in UML-RSDS as a natural extension of single-input rules. Iterative target patterns are not treated by this translation. These are a deprecated feature of ATL since ATL 2.0. Reflective operations and *resolveTemp* are not supported in ATL expressions for this translation.

The ATL refining mode does not support direct update-in-place transformation in the usual sense, where a single model is both read and written by rules. To support direct update-in-place semantics in ATL we adopt the convention that if the same declaration $tj : Tj$ occurs in both *from* and *to* clauses, this means that tj is directly updated by the rule. Then such variables do not have an $OUT\$Tj \rightarrow exists(tj \mid tj.\$id = \$id \ \& \ \dots)$ quantifier in the conclusion of the corresponding UML-RSDS constraint, instead each reference $tj.f$ to a feature of tj in the rhs of a binding $p \leftarrow e$ in the *to* clause is interpreted as $tj.f@pre$. Such rules both write and read the target model.

6.3 Importing QVT-R transformations

QVT is an OMG standard language for model transformation [18], consisting of relational (QVT-R), operational (QVT-O) and core (QVT-Core) languages. In this section we describe the semantic mapping from QVT-R to UML-RSDS. This provides a formal semantics of QVT-R in terms of the UML subset supported by UML-RSDS.

An example transformation, from UML to relational databases, is given in [18]:

```

transformation tau(uml1 : SimpleUML, rdb1 : SimpleRDMS) {
  top relation Class2Table
  { checkonly domain uml1 c : Class { name = n, attribute = a: Attribute {name = an} };
    enforce domain rdb1 t : Table { name = n, column = col: Column{name = an} };
    where { Attribute2Column(a,col) }
  }

  relation Attribute2Column
  { checkonly domain uml1 a : Attribute { type = t:Type { name = tn } };
    enforce domain rdb1 c : Column { coltype = tn };
  }
}

```

The *name* attributes of *Class* and *Table* are identities/keys, and *attribute* and *column* are set-valued roles.

Table 15 expresses the correspondence between QVT-R and UML-RSDS language elements.

In the following we assume that a QVT-R transformation is executed in the direction of the *enforce* domains as targets, with the *checkonly* domains as sources. Other execution variations can be modelled in a similar way. We assume that top-level relations are not called in *where* clauses. We will use the QVT-R metamodel classes such as *TemplateExp* and *Relation* from [18] to define the mapping details.

An *ObjectTemplateExp* *ote* of the form

```
e : E { f1 = val1, ..., fn = valn } { P }
```

<i>QVT-R element</i>	<i>UML representation in UML-RSDS</i>
RelationalTransformation	UseCase
Relation (isTopLevel = true)	Use case postcondition
Relation (isTopLevel = false)	Operation of use case
Relation Variable	Variable defined in constraint antecedent
RelationDomain (isEnforceable = false)	Constraint antecedent
RelationDomain (isEnforceable = true)	Constraint succedent
Variable bindings	Let/quantified variable bindings
Check-before-enforce semantics	Unique instantiation semantics
Key attributes	Identity attributes

Table 15: Correspondence of QVT-R and UML-RSDS

occurring in a *checkonly* domain is interpreted as a constraint antecedent predicate $cpred(ote, bound)$ formed as a conjunction of the interpretations of each *PropertyTemplateItem* $fi = vali$ as follows, where $bound$ is a list of the variables bound in the relation logically prior to the template:

- If $vali$ is itself an *ObjectTemplateExp*, such as $x : F \{g1 = w1, \dots, gm = wm\}$, the interpretation is applied recursively to the items of $vali$, to produce a predicate $cpred(vali, bound)$. The result for $fi = vali$ is $x = e.fi \ \& \ cpred(vali, bound \cup \{x\})$ if fi is single-valued, or $x : e.fi \ \& \ cpred(vali, bound \cup \{x\})$ if fi is collection-valued.
 x is considered to be bound by the first conjunct in both cases.
- Otherwise, if $vali$ is an unbound variable, $vali \notin bound$, and fi is single-valued, the predicate is $vali = e.fi$. If fi is collection-valued, the predicate is $vali : e.fi$.
Subsequently, $vali$ is considered to be bound.
- If $vali$ is a bound variable or other expression, the predicate is $e.fi = vali$ or $vali : e.fi$.

For the domain root variable e , the predicate $e : E$ is added as the first conjunct of the result. Any additional predicate P of the domain is added as the final conjunct of $cpred(ote, bound)$.

The checkonly domain template

```
uml1 c : Class {name = n, attribute = a : Attribute{name = an}};
```

is therefore interpreted as: $c : Class \ \& \ n = c.name \ \& \ a : c.attribute \ \& \ an = a.name$.

For enforce domains, the interpretation of ote as a constraint succedent predicate is $epred(ote, bound)$ formed as follows:

- If $vali$ is itself an *ObjectTemplateExp*, such as $x : F \{g1 = w1, \dots, gm = wm\}$, the result for $fi = vali$ is $F \rightarrow exists(x \mid e.fi = x \ \& \ epred(vali, bound \cup \{x\}))$ if fi is single-valued, or $F \rightarrow exists(x \mid x : e.fi \ \& \ epred(vali, bound \cup \{x\}))$ if fi is collection-valued. The quantification on x is omitted if x is already in $bound$ prior to the *ObjectTemplateExp*.
- Otherwise, if fi is single-valued, the predicate $e.fi = vali$. If fi is collection-valued, the predicate $vali : e.fi$.

The root variable e of the domain is \exists -quantified over its type, if it is not already bound. Any additional predicate P of the domain is added as the final conjunct of $epred(ote, bound)$.

The enforce domain template

```
rdb1 t : Table {name = n, column = col : Column {name = an}};
```

is therefore interpreted as:

$$Table \rightarrow exists(t \mid t.name = n \ \& \ Column \rightarrow exists(col \mid col : t.column \ \& \ col.name = an))$$

For each relation r (top or non-top) we introduce an auxiliary trace entity $r\$trace$ which has attributes/references consisting of the root variables of all domains of r , and attributes for the root variables of object templates occurring within these, together with other variables that occur in the *when* clause.

The main interpretation of a QVT top-level *Relation* r with multiple (at least one) source domains, and multiple (at least one) target domains is then a constraint Con_r which has antecedent the conjunction of predicates $e : E \ \& \ cpred$ for each source domain

checkonly domain src e : E { f1 = val1, ..., fn = valn }

and succedent the conjunction of predicates $epred$ for each target domain

enforce domain trg e : E { f1 = val1, ..., fn = valn }

In addition, the *when* clause is interpreted as a predicate $whenp$ and conjoined to the antecedent, and the *where* clause is interpreted as a predicate $wherep$ and conjoined to the succedent. An additional *when* clause is that the source domain elements do not already appear in some trace:

$$not(r\$trace \rightarrow exists(rx \mid rx.e1 = e1 \ \& \ \dots \ \& \ rx.ek = ek))$$

where the ei are all the root/subordinate object variables of the source domains of r , plus additional *when* variables. The extended $whenp$ is denoted $whenpx$.

An additional *where* predicate, always incorporated into the succedent of Con_r for separate-models transformations, is the creation of a trace object for this rule application:

$$r\$trace \rightarrow exists(rx \mid rx.e1 = e1 \ \& \ \dots \ \& \ rx.en = en)$$

where the ei are all the features of the trace. Note that $n > k$. The extended *where* predicate is denoted $wherepx$.

A *RelationCallExp* of $r(x1, \dots, xm)$ in a *when* clause of relation r' is interpreted by the corresponding query predicate

$$rx : r\$trace \ \& \ x1 = rx.e1 \ \& \ \dots \ \& \ xm = rx.em$$

Where the ei are the domain root variables of r . The xi are then considered bound within the relation r' .

Thus in general, Con_r has the form:

$$\begin{aligned} &:: \\ &whenpx \ \& \ \bigwedge_{cdom} cpred(cdom) \Rightarrow \\ &\quad \bigwedge_{edom} epred(edom) \ \& \ wherepx \end{aligned}$$

Where the conjunction is taken over all source domains in the antecedent, and over target domains in the succedent. The scope of *exists* quantifiers in the succedent is extended to the end of the constraint.

If the *when* clause contains disjunctions, then Con_R is split into multiple constraints, one for each disjunct.

Con_r deals with the case where the relation has not previously been executed for the source elements. In addition, to handle change-propagation for cases where a source-target relation has been established but targets may need to be modified to remain consistent with modified source elements, we define a constraint $Pres_r$ for r with the schematic structure

$$\begin{aligned} &r\$trace :: \\ &whenp \ \& \ \bigwedge_{cdom} cpred(cdom) \Rightarrow \\ &\quad \bigwedge_{edom} epred(edom) \ \& \ wherep \end{aligned}$$

Where the conjunction is taken over all non-target domains in the antecedent, and over target domains in the succedent. Again, the scope of *exists* quantifiers from the $epred$ formulae are extended over the remainder of the constraint, and $Pres_r$ is split into cases for each *when* disjunct.

The interpretation of *Class2Table* as a mapping transformation rule therefore consists of two postcondition constraints. The first, *PresClass2Table*, handles the case where the rule has previously been executed, and performs only updates (but does not recreate the domain objects or trace) to re-establish the relation if necessary. The second, *ConClass2Table*, is used in the case that a trace record for the relation does not already exist.

PresClass2Table is:

$$\begin{aligned} & \textit{Class2Table}\$trace :: \\ & n = c.name \ \& \ a : c.attribute \ \& \ an = a.name \ \Rightarrow \\ & \quad t.name = n \ \& \ col : t.column \ \& \ col.name = an \ \& \\ & \quad \quad \quad \textit{attribute2column}\$op(a, col) \end{aligned}$$

ConClass2Table is:

$$\begin{aligned} & :: \\ & c : \textit{Class} \ \& \ n = c.name \ \& \ a : c.attribute \ \& \ an = a.name \ \& \\ & \quad \textit{not}(\textit{Class2Table}\$trace \rightarrow \textit{exists}(c2t \mid c2t.c = c \ \& \ c2t.a = a)) \ \Rightarrow \\ & \quad \quad \textit{Table} \rightarrow \textit{exists}(t \mid t.name = n \ \& \\ & \quad \quad \quad \textit{Column} \rightarrow \textit{exists}(col \mid col : t.column \ \& \ col.name = an \ \& \\ & \quad \quad \quad \quad \textit{attribute2column}\$op(a, col) \ \& \\ & \quad \quad \quad \quad \textit{Class2Table}\$trace \rightarrow \textit{exists}(c2t \mid \\ & \quad \quad \quad \quad \quad c2t.c = c \ \& \ c2t.a = a \ \& \\ & \quad \quad \quad \quad \quad c2t.col = col \ \& \ c2t.t = t))) \end{aligned}$$

Check-before-enforce semantics is assumed for $E \rightarrow \textit{exists}(x \mid P)$: a new E object x is only created if there is not already one which satisfies the predicate P . *stat_{LC}* semantics is used for the procedural interpretation of the UML-RSDS constraints derived from the QVT-R specification.

The *Con_R* constraint creates target objects and a trace for source elements which have not yet been mapped. The logical interpretation highlights a flaw in the original specification: that a table and column are created in the same relation (the logic is of the form $\forall c \forall a \exists t \exists col$). It is better practice to separate these actions into different relations, so that the logic becomes $\forall c \exists t \forall a \exists col$.

A QVT-R relational transformation τ is interpreted as a use case τ with three sequential sub-use cases. *Pres _{τ}* has postconditions *Pres _{r}* for each relation r of τ , and *Con _{τ}* includes *Con _{r}* for each top-level *Relation* r of τ . As in Medini-QVT, we consider that the order of these relations is significant (projects.ikv.de/qvt). For non-top-level relations r , a constraint *Op _{r}* is used to form the postcondition of an update operation *op _{r}* which has parameters the root variables of the relation domains of r . *Op _{r}* is formed as for *Con _{r}* , but without antecedent conjuncts $e : E$ for the root variables of the checkonly relation domains, and without quantifiers $E \rightarrow \textit{exists}(e \mid \dots)$ for the root variables of the enforce relation domains of r . It is assumed that all the operation parameters (the domain root variables) are bound at the point of call.

For example, the non-top relation

```
relation Attribute2Column
{ checkonly domain uml1 a : Attribute { type = t : Type { name = tn }};
  enforce domain rdb1 c : Column { coltype = tn };
}
```

has the interpretation as an operation

$$\begin{aligned} & \textit{attribute2column}\$op(a : \textit{Attribute}, c : \textit{Column}) \\ & \textit{post} : t = a.type \ \& \ tn = t.name \ \Rightarrow \ c.coltype = tn \ \& \\ & \quad \textit{Attribute2Column}\$trace \rightarrow \textit{exists}1(a2c \mid \\ & \quad \quad a2c.a = a \ \& \ a2c.c = c \ \& \ a2c.t = t) \end{aligned}$$

These operations are added as owned operations of the use case representing the transformation. *exists1* is used here to avoid re-creating a trace if it already exists.

QVT-R has the semantic feature of *implicit deletion*: if a relation is not satisfiable for an existing element of the target model, which matches the target domain pattern of the relation, then this element should be deleted. We express this semantics by deleting any target element which does not appear in any trace relation after execution of the main transformation:

$$E :: \text{not}(r1\$trace \rightarrow \text{exists}(r1 \mid r1.e1 = self)) \ \& \ \dots \ \& \ \text{not}(rm\$trace \rightarrow \text{exists}(rm \mid rm.em = self)) \Rightarrow self \rightarrow isDeleted()$$

where the rk , $k = 1$ to m , are all the relations in which the target entity E occurs as the type of a target object variable $ek : E$. The constraint is denoted $CleanTarget_E$.

For the above example this produces the constraint

$$Table :: \text{not}(Class2Table\$trace \rightarrow \text{exists}(class2table \mid class2table.t = self)) \Rightarrow self \rightarrow isDeleted()$$

in the case of $Table$, and

$$Column :: \text{not}(Attribute2Column\$trace \rightarrow \text{exists}(attribute2column \mid attribute2column.c = self)) \ \& \ \text{not}(Class2Table\$tr) \Rightarrow self \rightarrow isDeleted()$$

in the case of $Column$.

The cleanup constraints are executed in a sub use case $Cleanup_\tau$ after the main transformation constraints.

The invariant for a transformation is in three parts: (i) that any target object must appear as the target domain of some trace; (ii) that all trace objects in $R\$trace$ satisfy the logical relation defined by R ; (iii) that source objects that satisfy the conditions of rule R appear together in some trace.

The $CleanTarget$ rules establish (i), the $Pres$ rules establish (ii), the Con rules establish (iii). For separate-models transformations, $CleanTarget$ does not invalidate (ii) or (iii) because they only update the target model, not the source or trace.

The invariant implies that if a source-target tuple (s, t) for $s : S$, $t : T$ satisfies relation R , then it appears in $R\$trace$ and vice-versa. This holds since if (s, t) satisfy R after execution of the transformation, then s must appear as the source elements in some $tr \in R\$trace$, because R would have been executed for s and some $t' : T$ would have been created or selected as a match for s . Also, t must appear as a target element in some $tr' \in R'\$trace$, where R' creates elements $t : T$, otherwise t would have been deleted by the cleanup rules.

If R is the only relation that creates T elements, then $R = R'$ and there are tuples (s, t) , (s', t) both in $R\$trace$. If R was executed on s' before s , then t would already exist when R is executed on s , and hence – by the check-before-enforce semantics of QVT-R, would be selected as a match for s , hence $t' = t$. If R was executed on s before s' then again $t' = t$ by the check-before-enforce property, and the result follows.

6.4 QVT-R in-place mode

The semantics of in-place transformations is described very briefly in [18] and it is unclear how this semantics interacts with aspects such as implicit deletion and change propagation. The translation to UML-RSDS described above can be used to give a semantics for in-place QVT-R specifications, but with a fixed-point execution model.

An example of an update-in-place transformation is a graph processing transformation, operating on models of a language $NodeMM$ with a single entity type $Node$, with a key attribute $name : String$ and a many-many symmetric association $neighbours$ on $Node$. The transformation should remove all nodes which do not have a single neighbour. In UML-RSDS this would be written as:

```
Node:: neighbours->size() /= 1 => self->isDeleted()
```

and this has a fixpoint implementation: nodes with no neighbours, or with more than one neighbour, are deleted until no such nodes remain.

In contrast, the ATL refining mode implementation only deletes nodes which have more/less than one neighbour in the *initial* model:

```
module M;
create OUT : NodeMM refining IN : NodeMM;
rule r1
{ from n : Node (n.neighbours->size() <> 1)
  to drop
}
}
```

This has the translation

```
IN$Node::
  OUT$Node->exists( node | node.name = name )

IN$Node::
  node = OUT$Node[name] =>
    node.neighbours = OUT$Node[neighbours.name]

IN$Node::
  node = OUT$Node[name] &
  neighbours->size() /= 1 => node->isDeleted()
```

The first two constraints copy the source model to the target. In the final constraint, target elements are deleted if they correspond to source elements with $neighbours \rightarrow size() \neq 1$. All the constraints are of type 1. For this ATL version nodes in the result model need not have one neighbour, for example if the initial graph had three nodes linked in a chain: $a - b - c$, then b would be deleted, leaving a and c with 0 neighbours.

In QVT-R the transformation could be expressed as:

```
relational transformation (nodemm : NodeMM)
{ top relation cleangraph
  { checkonly domain nodemm n : Node { };
    enforce domain nodemm n : Node { };
    when { n.neighbours->size() = 1 }
  }
}
```

This has the *Con* constraint:

```
::
n : Node & n.neighbours->size() = 1 &
not( cleangraph$trace->exists( cleangraph | cleangraph.n = n ) ) =>
  cleangraph$trace->exists( cleangraph | cleangraph.n = n )
```

(The $Pres_{cleangraph}$ rule is vacuous in this case).

The cleanup rule expresses that if the above relation is not applicable, then the node is deleted:

```
Node::
not(cleangraph$trace->exists( cleangraph | cleangraph.n = self )) =>
  self->isDeleted()
```

In addition for this mode we add a cleanup constraint that if a tuple (s, t) is in Rtrace$ and does not satisfy the antecedent of R , then it is deleted. Here this means:

```
cleangraph$trace::
not(n.neighbours->size() = 1) =>
  self->isDeleted()
```

The sequence $Pres_\tau$; Con_τ ; $Cleanup_\tau$ is iterated until the conditions (i), (ii) and (iii) hold.

6.5 Importing ETL modules

The Epsilon Transformation Language (ETL) is a hybrid transformation language similar in many respects to ATL, but with a more complex execution semantics [7]. Table 16 shows the mapping of ETL elements to UML-RSDS. As with the translation from ATL to UML-RSDS, we assume that all rule inheritance has been statically expanded out so that only concrete (non-abstract) rules exist. Likewise for module import. We map rule code blocks to UML-RSDS operations, because of the procedural nature of ETL rules.

<i>ETL</i>	<i>UML in UML-RSDS</i>
Module M	Use case M'
Pre/Post block, Non-lazy rule	Operation, and use case postcondition invoking the operation
Lazy rule	Operation invoked via <i>equivalent()</i> operation
Implicit rule invocation	Object lookup using traces; object creation
Statement blocks	Operation with activity, invoked from postcondition
Operation	Use case or entity operation

Table 16: Correspondence of ETL and UML-RSDS

An ETL transformation rule

```
rule R
transform s : S
to t1 : T1, ..., tn : Tn
{ guard: G
  code
}
```

is mapped to a postcondition

```
S::
G[self/s] =>
  T1->exists( t1 | ... Tn->exists( tn |
    $Trace->exists($t1 | $t1 : $trace & $t1.target = t1) & ...
    $Trace->exists($tn | $tn : $trace & $tn.target = tn) &
    R(t1,...,tn)) ...)
```

Where R is a new operation of S with activity $code[self/s]$. $\$Trace$ is a new auxiliary entity which records trace links between source and target elements. The interpretation creates target objects, and trace objects for each target element, then calls the body code operation R to carry out bindings of the target object features.

A lazy rule

```
@lazy
rule R
transform s : S
to t1 : T1, ..., tn : Tn
{ guard: G
  code
}
```

is instead mapped to an operation $R() : T1$ of S , with pre and postconditions

```
pre:
  G[self/s]
post:
```

```

T1->exists( t1 | ... Tn->exists( tn |
  $Trace->exists($t1 | $t1 : $trace & $t1.target = t1) & ...
  $Trace->exists($tn | $tn : $trace & $tn.target = tn) &
  $R(t1,...,tn) & result = t1) ...)

```

Where $\$R$ is a new operation of S with activity $code[self/s1]$. As with non-lazy rules, the effect of the interpretation is to create target elements and corresponding trace objects, and to perform bindings. Additionally, the first target element is returned as the result of the rule.

$equivalent() : \$OUT$ is defined as an operation of the superclass $\$IN$ of all source entity types. It has the schematic definition

```

equivalent() : $OUT
post:
  result = if $trace->notEmpty() then $trace.target->any()
           else if self : S1 then self->oclAsType(S1).LR1()
           ...
           else if self : Sm then self->oclAsType(Sm).LRm()
           endif ... endif else null endif

```

Where LR1 to LRm are all the lazy rules in the transformation, and their source entity types are S1 to Sm respectively. This returns the first target element corresponding to the source $self$ object, if this exists, otherwise, it applies a matching lazy rule to create and return the target.

Further details of the mapping and an example are given in [14].

An operation

```

operation T op(pars) : RT
{
  code
}

```

is mapped to an operation of T with the same signature, and with activity defined by `code`.

6.6 Importing Flock modules

Flock is an Epsilon language for migration transformations [20]. It defines migrations by combinations of deletion, retyping and migration rules, and by implicitly copying source model elements to the target model if they conform to the target language and there is no explicit rule that they satisfy. The translation to UML-RSDS creates a use case for the Flock module, and constraints to define the explicit and implicit rules of the module.

Table 17 shows the mapping of Flock elements to UML-RSDS.

<i>Flock</i>	<i>UML-RSDS</i>
Module	Use case
Deletion rule	Guard on use case postcondition
Retyping rule for E	Use case postcondition (if no migration rules for E) or part of migrate rule postcondition
Migrate rule for E	Postconditions based on retyping rules for E , with operation to define body of rule
Object lookup using <i>migrated</i> and <i>equivalent</i>	Object lookup using object indexing
Body code blocks	Operation with activity, invoked from postcondition
Operation	Operation of context entity type

Table 17: Correspondence of Flock and UML-RSDS

Each entity type E of the source language is renamed to $IN\$E$, and each entity type F of the target language is renamed to $OUT\$F$. Each root class of the source or target language is extended with a new principal primary key $\$id : String$. Each deletion rule

delete E when: Del

is expressed by a new additional guard $not(Del)$ for each retyping and migration rule on E or on a subtype of E .

We make use of the following semantics-preserving transformations on Flock specifications: 1) Each retyping rule

retype E to F when: Cond

on an abstract class E can be replaced by rules

retype Ei to F when: Cond

on each concrete leaf subclass Ei of E , and likewise for migrate rules. 2) For each concrete leaf source class E which is also a concrete leaf class in the target language, a default retyping rule r_E

retype E to E when: Cond

is added, where $Cond$ is the conjunction of $not(Cond_i)$ for each condition $Cond_i$ of an explicit retyping rule for E . $Cond$ is *true* if there are no explicit retyping rules for E . These transformations assist in the translation to UML-RSDS and in identifying errors such as contradictory retyping rules.

As with ATL, the representation of a Flock module m is formed of two phases in UML-RSDS: in the first phase for each retyping rule

retype E to F when: Cond

a constraint

$$Cond' \Rightarrow OUT\$F \rightarrow exists(f \mid f.\$id = \$id)$$

on $IN\$E$ is added. $Cond'$ substitutes *self* for *original* in $Cond$. Finally, for concrete source entity types E which are also target language entity types, and are concrete entity types in the target language, a constraint

$$Cond' \Rightarrow OUT\$E \rightarrow exists(e \mid e.\$id = \$id)$$

on $IN\$E$ is added, for the implicit retyping rule r_E of E to itself.

In the second phase the migration and retyping rules are used together to define feature values of the target objects. For each migration rule:

migrate E when: Cond1 ignoring ig
{ body }

on E , and each retyping rule

retype E to F when: Cond2

on E , a (potential) constraint on $IN\$E$ is:

$$f = OUT\$F[\$id] \ \& \ Cond1 \ \& \ Cond2 \Rightarrow \\ f.att = att \ \& \ \dots \ \& \ f.r = FRef[r.\$id] \ \& \ self.op(f)$$

is included in the phase, provided that $Cond1$ and $Cond2$ are consistent. The attribute att and role r values of $IN\$E$ are copied to corresponding features of $OUT\$F$ for all same-named and compatible-typed features which do not appear in $ig \cup \{\$id\} \cup wr(body)$. $FRef$ is the element type

of r for $OUT\$F$. op is an operation of $IN\$E$ with activity the translation of $body$ to a UML-RSDS activity, and with parameter $migrated : OUT\$F$. f plays the role of $migrated$ in the Flock code, $self$ represents $original$. An expression $exp.equivalent()$ is interpreted as $ET[exp.\$id]$ where ET is the expected element type of the result. Likewise for $exp.equivalents()$. Additionally the implicit retyping r_E is combined with the migrate rule, if this rule exists.

If there is no migrate rule for E , then the second phase constraints for E are derived from the retyping rules. They have the form:

$$f = OUT\$F[\$id] \ \& \ Cond2 \Rightarrow \\ f.att = att \ \& \ \dots \ \& \ f.r = FRef[r.\$id]$$

The attribute att and role r values of $IN\$E$ are copied to corresponding features of $OUT\$F$ for all same-named and compatible-typed features, except for $\$id$. $FRef$ is the element type of r for $OUT\$F$. The default retyping is also given a phase 2 constraint if it exists.

As an example, the process migration example of [20] is translated as follows. The Flock specification

```
delete Port when: not(original.isInput()) and not(original.isOutput())

delete Port when: original.isInput() and original.isOutput()

retype Port to InputPort when: original.isInput()

retype Port to OutputPort when: original.isOutput()

migrate Connector when: true
{ migrated.in := original.from;
  migrated.out := original.to
}
```

is translated to the following constraints:

```
self.isInput() & not(self.isOutput()) => OUT$InputPort->exists( q | q.$id = self.$id )

self.isOutput() & not(self.isInput()) => OUT$OutputPort->exists( q | q.$id = self.$id )

on IN$Port, expressing the deletion and retyping rules (first phase),

OUT$Component->exists( c | c.$id = self.$id )

OUT$Connector->exists( c | c.$id = self.$id )

self.isInput() & not(self.isOutput()) &
q = OUT$InputPort[self.$id] => q.name = self.name

self.isOutput() & not(self.isInput()) &
q = OUT$OutputPort[self.$id] => q.name = self.name
```

on $IN\$Component$, $IN\$Connector$ and $IN\$Port$, expressing the implicit copy of Component and Connector (first phase), and the second phase of the retyping of Port. The second phase of implicit copy for Component and Connector, including the migration rule for Connector, is formed by the constraints:

```
c = OUT$Component[self.$id] =>
  c.name = self.name &
  c.connections = OUT$Connector[self.connections.$id] &
  c.ports = OUT$Port[self.ports.$id] &
  c.subcomponents = OUT$Component[self.subcomponents.$id]
```

```

c = OUT$Connector[self.$id] =>
  c.name = self.name &
  c.in = OUT$InputPort[self.from.$id] &
  c.out = OUT$OutputPort[self.to.$id]

```

on IN\$Component and IN\$Connector, respectively. These constraints can then be used to verify invariants of the transformation, such as

$$OUT\$InputPort \rightarrow \text{forall}(p \mid IN\$Port \rightarrow \text{exists}(q \mid q.\$id = p.\$id \ \& \ q.isInput()))$$

and to generate a reverse transformation.

Consistency restrictions, such as that two distinct retyping rules must apply to disjoint sets of objects, can also be expressed using this translation.

7 Refactoring

On the *Transform* menu a set of class diagram refactorings are provided to help in improving the structure of a class diagram:

- Remove redundant inheritance: removes generalisations which link classes which already have a generalisation relationship via an intermediate class.
- Introduce superclass: implements the ‘Extract superclass’ refactoring to identify common parts of two or more classes.
- Pushdown abstract features: used to copy ‘virtual’ features of interfaces down to subclasses, in order to implement a form of multiple inheritance.

From version 1.6, Move attribute and Move operation refactorings are also supported.

There are also inbuilt refinement transformations, mainly used for web application construction (transforming a UML class diagram into a relational database schema):

- Express statemachine on class diagram: creates an enumerated type and conditional expressions within the owning entity of a statemachine, in order to express the statemachine semantics.
- Introduce primary key: adds a String-valued identity attribute to the specified classes.
- Remove many-many associations: replaces a many-many association with a new class and two many-one associations.
- Remove inheritance: replaces inheritance by an association.
- Aggregate subclasses: merges all subclasses of a given class into the class, in order to remove inheritance.
- Remove association classes: replaces association classes by two many-one associations.
- Introduce foreign key: define a foreign key to represent a many-one association.
- Matching by backtracking: introduces a search algorithm for constructing a mapping.

8 Design patterns

Several design patterns are provided to assist developers:

- Singleton: constructs a singleton class, following the classic GoF pattern structure.
- Facade: identifies if a Facade class would improve the modularity of a given class diagram.
- Phased Construction: converts a transformation postcondition which has a nested *forAll* clause within an *exists* or *exists1* succedent into a pair of postconditions without the nested clause, and using instead an internal trace association.
- Implicit Copy: looks for a total injective mapping from the source language entities and features to the target language and defines a transformation which copies source data to target data according to this mapping.
- Auxiliary Metamodel: constructs an auxiliary trace class and associations linking source entity types to the trace class and linking the trace class to target entity types.

Other patterns are incorporated into the design generation process:

- Unique Instantiation: as described following Table 2, the update interpretation of $E \rightarrow \text{exists}(e \mid e.id = val \ \& \ P)$ where *id* is the principal unique/primary key of *E* checks if there is already an existing $e : E$ with $e.id = val$ before creating a new object. The design for $E \rightarrow \text{exists1}(e \mid P)$ checks if there is an existing $e : E$ satisfying *P* before creating a new object.
- Object Indexing: indexing by unique/primary key is built-in: if entity type *E* has attribute $att : String$ as its principal primary key (identity attributes directly declared in *E* are considered first, then those in direct superclasses, etc), then lookup by string values is supported: $E[s]$ denotes the unique $e : E$ with $e.att = s$, if such an object exists. Lookup by sets of string is also supported: $E[st]$ is the set of *E* objects with *att* value in *st*. If there is no identity attribute defined for *E*, but some string-valued attribute from *E* or a superclass satisfies $E \rightarrow \text{isUnique}(att)$, then lookup by string value can also be used for the first most local such *att*.
- Omit Negative Application Conditions (NACs): if type 2 or 3 (Section 6) constraints $Ante \Rightarrow Succ$ have that *Ante* and *Succ* are mutually inconsistent, then the design generator will simplify the generated design by omitting a check for *Succ* if *Ante* is true. It will also prompt the user to ask if this simplification should be applied. For example, any constraint with the form

$$Ante \Rightarrow P \ \& \ self \rightarrow \text{isDeleted}()$$

can be optimised in this way, likewise for

$$x : E \ \& \ Ante \Rightarrow P \ \& \ x \rightarrow \text{isDeleted}()$$

Note that the use of operation calls within constraints to define update functionality may prevent correct implementation of type 2 or 3 constraints. The implementation of a constraint

$$G \Rightarrow Post \ \& \ op(pars)$$

in such cases tests the NAC $G \ \& \ \text{not}(Post)$ before applying the constraint, meaning that cases where *G* and *Post* both hold will not result in application of the constraint, even if the effect of $op(pars)$ has not been established. A partial solution to this problem is to use Auxiliary Metamodel to keep an explicit record of those source elements which have been already processed, and avoid re-application of the constraint to such elements.

- Replace fixed-point iteration by bounded iteration: Again, for type 2 or 3 constraints $Ante \Rightarrow Succ$, which would normally be implemented by a fixed-point iteration, a simpler and more efficient implementation by a bounded loop is possible if the effect of $Succ$ never produces new cases of objects satisfying $Ante$ which did not exist at the start of the constraint implementation. In some cases this can be automatically detected, e.g.:

$$Ante \Rightarrow self \rightarrow isDeleted()$$

on entity E where there is no data-dependency of $Ante$ upon E . The user is prompted to confirm such optimisations.

Developers can enforce the use of bounded iteration by using expressions of the form $E@pre$, $e@pre$ instead of E , e in read expressions within the constraint. In particular, if there are two mutually data-dependent postcondition constraints $C1$, $C2$ where $C1$ reads entity type E , and $C2$ creates instances of E , we can assert that the instances created by $C2$ are irrelevant to $C1$ by using $E@pre$ in $C1$, denoting the set of E objects created before $C1$ begins execution, thus potentially removing the need for fixed-point iteration of the constraint group $\{C1, C2\}$. However, use of $@pre$ in this way can produce code which does not establish the constraint (as written) as a postcondition, as discussed in Section 6 above.

- Factor out Duplicated Expressions: complex duplicated read-only expressions in constraints can be factored out by defining a let variable in the constraint antecedent. Operations can be used to factor out expressions and (by caching) avoid repeated evaluation of a single expression.

9 State machines

State machines define the dynamic behaviour of objects and operations. They can be used to give operation definitions (instead of pre and postconditions), and to express the life cycle of objects.

For example, a student could have a linear lifecycle of successive states $Year1$, $Year2$, $Year3$ and $Graduated$.

An editor for state machines is provided, Figure 13 shows the interface for this editor.

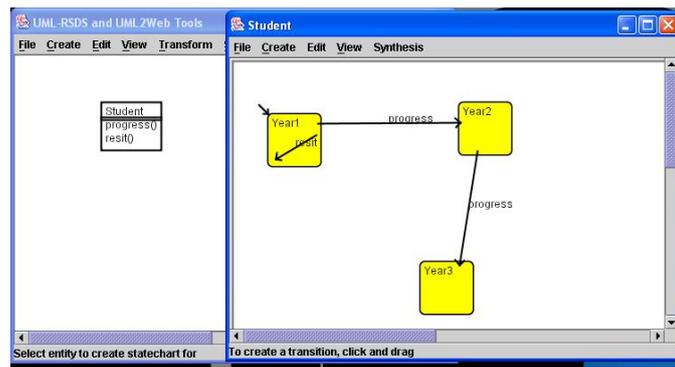


Figure 13: State machine editor

There are the following options:

File menu – options to save, load and print the state machine, and options to instantiate a class state machine for a particular object, and to take the product of two state machines. State machine data for model element m is saved and retrieved from the file $output/m.dat$.

Create menu – options to create states and transitions.

Edit menu – options to edit the state machine states and transitions and to resize the diagram.

View menu – options to view the lists of all states, transitions, events and attributes of the model.

Synthesis menu – options to analyse the state machine structure, to generate B and to carry out slicing of the state machine.

On the class diagram editor, the *Transformation* menu option *Express state machine in class diagram* adds variables and expressions to the class diagram to express the meaning of a state machine for an entity.

When an operation state machine is specified, the option *Check structure* on the state machine editor Synthesis menu should be applied before code is generated for the operation (on either the class diagram or state machine editor tool). This option identifies terminal, loop and decision states (displayed as green, red and blue, respectively), and warns the user if the state machine is not in the form of structured code. This analysis is then used to map the state machine into Java, C# or C++ code by the Generate code options on the class diagram Synthesis menu.

Currently, only the subset of UML state machine notation with basic states is supported by UML-RSDS, although in principle larger subsets could be encoded [9].

10 Synthesis of EIS applications

An enterprise information system (EIS) typically consists of five tiers of components:

Client tier Web pages or other interfaces by which clients use the system. Consists of HTML files.

Presentation tier Components which construct the GUI of the system and handle requests from the client tier. Consists of Java servlets and JSPs.

Business tier Components which represent the business operations (services) offered by the system, and the business entities of the system. Consists of Session Beans and Entity Beans, and Value Objects.

Integration tier Components which serve as a mediating interface between the business and resource tiers: abstracting the resource components. Consists of database interfaces and web service interfaces.

Resource tier Contains databases and other resources used by the system.

Such systems can be defined using many different technologies and techniques. We use Java technologies, and three alternative system styles:

- Servlet style: presentation tier is coded using only servlets and auxiliary classes to generate web pages.
- JSP style: presentation tier is coded only using JSPs to handle web requests and to generate web pages.
- J2EE style: Java Entity beans and Session beans are constructed to define the business tier of an application, together with value objects and a database interface class.

In each case, the system specification is given as a class diagram and set of use cases defining the user operations to be provided by the system. Use cases can be:

- create E: create an instance of entity E (setting its attribute values)
- delete E: delete an instance of E
- edit E: edit an instance of entity E (setting its attribute values)

- add E r: add an element to the role set r of an instance of E
- remove E r: remove an element from the role set r of an instance of E
- list E: list all instances of E
- searchBy E a: find instances of E with a given value for attribute a.

The use cases and class diagram elements map to EIS code components as follows (Table 18).

<i>Element</i>	<i>Servlet style</i>	<i>JSP style</i>	<i>J2EE style</i>
Class <i>E</i>	Database table	Session bean Value object	+ Entity bean as prev.
Attribute	Table column form field	as prev. as prev.	as prev. as prev.
Use case <i>op</i>	CommandPage.java (view) opPage.java (form generator) op.html (form) opResultPage.java (view) Dbi.java (data access object)	commands.html op.jsp (view + controller) op.html (form) Dbi.java	 as prev.

Table 18: Mapping of UML-RSDS to web code

More details and examples are given in [15].

11 Mapping to B

B is a formal language with powerful analysis capabilities for checking the correctness of specifications [8]. Several commercial and free tools exist for B:

- BToolkit
- Atelier B
- B4Free

The following restrictions on UML-RSDS specifications are necessary before they can be translated to B:

- Attributes of different classes (except for inherited attributes) must have different names.
- Update operations cannot have return values or be recursively defined.
- Feature, variable and class names should have more than one letter. Underscore cannot be used in class, variable or feature names.
- Real values cannot be used, only integers, strings and booleans, and enumerated types. Some B implementations only support natural numbers: non-negative integers.

The UML-RSDS type `int` corresponds to `INT` in B.

Two options for generating B are provided: (i) separate B machines for each root entity of the class diagram; (ii) one B machine for the entire system.

The first can be used if there are not invocations in both directions between two entities (in different inheritance hierarchies), and no bidirectional associations between such entities. In addition, if no entity in one hierarchy contains code creating entity instances in another hierarchy. Otherwise, in the case of such mutual dependencies, option (ii) must be used.

The following verification properties of a UML-RSDS transformation specification τ can be checked using B:

1. Syntactic correctness: if a source model m of S satisfies all the assumptions Asm of τ , then the transformation will produce valid target models n from m which satisfy the language constraints Γ_T of T .
2. Semantic preservation: if a predicate φ holds in m , then any target model n produced from m satisfies an interpretation $\chi(\varphi)$ of the formula.
3. Semantic correctness: that a given implementation for τ satisfies its specification.
4. Confluence: that all result models n produced from a given source model m must be isomorphic.
5. Termination: that τ is guaranteed to terminate if applied to valid source models which satisfy Asm .

Proof-based techniques for verifying transformation correctness properties have two main advantages: (i) they can prove the properties for all cases of a transformation, that is, for arbitrary input models and for a range of different implementations; (ii) a record of the proof can be produced, and subjected to further checking, if certification is required. However, proof techniques invariably involve substantial human expertise and resources, due to the interactive nature of the most general forms of proof techniques, and the necessity to work both in the notation of the proof tool and in the transformation notation.

We have selected B AMN as a suitable formalism for proof-based verification, B is a mature formalism, with good tool support, which automates the majority of simple proof obligations. We provide an automated mapping from transformation specifications into B, and this mapping is designed to facilitate the comprehension of the B AMN proof obligations in terms of the transformation being verified.

The entity types and features of the languages involved in a transformation τ are mapped into mathematical sets and functions in B. OCL expressions are systematically mapped into set-theory expressions. Appendix G gives details of this mapping.

A B AMN specification consists of a linked collection of modules, termed *machines*. Each machine encapsulates data and operations on that data. Each transformation is represented in a single main B machine, together with an auxiliary *SystemTypes* machine containing type definitions.

The mapping from UML-RSDS to B performs the following translations:

- Each source and target language (class diagram) L is represented by sets es for each entity type E of the language, with $es \subseteq objects$, and maps $f : es \rightarrow Typ$ for each feature f of E , together with B encodings of the constraints Γ_L of L for unmodified L . In cases where a language entity type or feature g is both read and written by the transformation, a syntactically distinct copy g_pre is used to represent the initial value of g at the start of the transformation. A supertype F of entity type E has B invariant $es \subseteq fs$. Abstract entity types E have a B invariant $es = f1s \cup \dots \cup fls$ where the Fi are the direct subtypes of E .

For each concrete entity type E of a source language, there is an operation $create_E$ which creates a new instance of E and adds this to es . For each data feature f of an entity type E there is an operation $setf(ex, fx)$ which sets $f(ex)$ to fx .

- The assumptions Asm of the transformation can be included in the machine invariant (using g_pre in place of g for data which is written by the transformation). Asm is also included in the preconditions of the source language operations $create_E$ and $setf$.
- Each use case postcondition *rule* is encoded as an operation with input parameters the names of the non-modifiable objects which are read by *rule*, and with its effect derived from the rule relation or from the *behavior* of an implementation of *rule*. The operation represents transformation computation steps δ_i of the *rule* implementation.

- Orderings of the steps for particular implementations can be encoded by preconditions of the operations, expressing that the effect of one or more other *rule'* has been already established for all applicable elements.
- Invariant predicates *Inv* are added as B invariants, using *g_pre* to express pre-state values *g@pre*.

In contrast to the mapping to Z3 described in Section 12, the mapping to B explicitly represents the computation steps of the transformation, and can therefore support verification that these maintain *Inv* and decrease any variant, *Q*. For a separate-models transformation $\tau : S \rightarrow T$ the machine invariant expresses $\Gamma_S \cup Asm0 \cup Inv$. *Asm0* are the conjuncts of *Asm* which are constraints on *S* only. For an update-in-place transformation $\tau : S \rightarrow S$ the invariant expresses $\Gamma_S \cup Asm@pre \cup Inv$. The machine represents the transformation *at any state during its computation*.

For transformations involving procedural code (ie., as the definition of activities of operations invoked from use case postconditions), the translation to B needs to use imperative constructs such as sequencing and loops in the B *generalised substitution* language. A REFINEMENT component also needs to be used for the B representation of the transformation instead of a MACHINE, since imperative constructs are not permitted in machines.

The procedural language used in UML-RSDS activities is very close to that used in B AMN generalised substitutions, however, the interpretation of loop statements requires encoding of OCL-style iteration over collections into the elementary WHILE loop construct of B. Table 19 shows this encoding. *unprocessed* and *i* are new variables. *e'* represents *e*.

<i>UML-RSDS statement</i>	<i>Condition</i>	<i>B AMN generalised substitution</i>
for (v : e) do stat	e set-valued	VAR unprocessed IN unprocessed := e'; WHILE (unprocessed /= {}) DO ANY v WHERE v : unprocessed THEN unprocessed := unprocessed - { v }; stat' END INVARIANT unprocessed \subseteq e' VARIANT card(unprocessed) END END
for (v : e) do stat	e sequence-valued	VAR i IN i := 1; WHILE (i \leq card(e')) DO VAR v IN v := e'(i); i := i + 1; stat' END INVARIANT i \leq card(e') + 1 VARIANT card(e') + 1 - i END END

Table 19: Mapping from UML-RSDS loop statements to B AMN

The mapping to B is suitable to support the proof of syntactic correctness, semantic preservation and semantic correctness by using internal consistency proof in B, a more complex mapping is used for the proof of confluence and termination, using refinement proof [11].

The general form of a B machine M_τ representing a separate-models transformation τ with source language *S* and target language *T* is:

```

MACHINE Mt SEES SystemTypes
VARIABLES
  /* variables for each entity type and feature of S */
  /* variables for each entity type and feature of T */
INVARIANT
  /* typing definitions for each entity type and feature of S and T */
GammaS &

```

```

    Asm0 & Inv
INITIALISATION
    /* var := {} for each variable */
OPERATIONS
    /* creation operations for entity types of S, restricted by Asm */
    /* update operations for features of S, restricted by Asm */
    /* operations representing transformation steps */
END

```

SystemTypes defines the type *Object_OBJ* of all objects, and any other type definitions required, eg., of enumerated types.

The operations to create and update *S* elements are used to set up the source model data of the transformation. Subsequently, the operations representing transformation steps are performed.

If *Asm0* consists of universally quantified formulae $\forall s : S_i \cdot \psi$, then the instantiated formulae $\psi[sx/s]$ are used as restrictions on operations creating $sx : S_i$ (or subclasses of S_i). Likewise, operation *setf*(*sx*, *fx*) modifying feature *f* of S_i has a precondition $\psi[sx/s, fx/s.f]$. All these operations will include the preconditions *Asm1* from *Asm* which concern only the target model.

As an example, the transformation of Figure 12 can be defined by the following partial machine:

```

MACHINE Mt SEES SystemTypes
VARIABLES objects, as, x, bs, y
INVARIANT
    objects <: Object_OBJ &
    as <: objects & bs <: objects &
    x : as --> INT & y : bs --> INT &
    !b.(b : bs => #a.(a : as & x(a) > 0 & y(b) = x(a)*x(a)))
INITIALISATION
    objects, as, x, bs, y := {}, {}, {}, {}, {}

```

The invariant expresses Γ_S and the *Inv* properties of the transformation. $\#b.P$ is B syntax for $\exists b \cdot P$, $!a.P$ is B syntax for $\forall a \cdot P$. $\&$ denotes conjunction, $<$ denotes \subseteq and $-->$ is \rightarrow (the total function type constructor). A universal set *objects* of existing objects is maintained, this is a subset of the static type *Object_OBJ* declared in *SystemTypes*.

Occurrences *E@pre* or *f@pre* in *Inv* or *Post* are interpreted by the additional variables *es_pre*, *f_pre* which have the same typing as *es* and *f*. They are modified in parallel with *es* and *f* by the source model creation and modification operations, and are unchanged by the operations for transformation computation steps.

The operations representing computation steps are derived from the rule implementations δ_i of the constraints *Cn*. This modelling approach facilitates verification using weakest precondition calculation, compared to more abstract encodings. If *Cn* has the form

$$SCond \Rightarrow Succ$$

on context entity S_i then the operation representing a computation step δ_i of *Cn* is:

```

delta_i(si) =
    PRE si : sis & SCond & not(si.Succ) &
      C1 & ... & Cn-1 & def(si.Succ)
    THEN
      stat'(si.Succ)
    END

```

where *stat'*(*P*) encodes the procedural interpretation *stat*(*P*) of *P* in B program-like statements, AMN generalised substitutions. These have a similar syntax to the programming language described in Appendix C, and use the same weakest-precondition semantics. B has an additional statement form $v := e1 || w := e2$ of *parallel assignment*: the assignments are performed order-independently, with the values of *e1*, *e2* being simultaneously assigned to *v*, *w*. The ANY WHERE THEN statement of B corresponds to a UML-RSDS creation statement.

If the implementation of τ defines a non-standard rule implementation of C_n , this implementation could be encoded in B in place of the above definition of $\delta_{i,n}$.

If τ 's implementation requires that all constraints C_1, \dots, C_{n-1} are established before C_n , this ordering can be encoded by including C_1, \dots, C_{n-1} in the preconditions of $\delta_{i,n}$. $\text{not}(Succ)$ can be omitted if negative application conditions are not checked by the implementation of C_n . For the mapping to B, $\text{def}(Succ)$ includes checks that numeric expressions in $Succ$ are within the size bounds of the finite numeric types NAT and INT of B, and that $objects \neq Object_OBJ$ prior to any creation of a new object.

The computational model of a transformation τ expressed in M_τ therefore coincides with the definition of transformation computation described in [12]: a computation of τ is a sequence of transformation steps executed in an indeterminate order, constrained only by the need to maintain Inv , and, if a specific implementation I is defined, to satisfy the ordering restrictions of I 's behaviour.

For the example of Figure 12, using a fixpoint implementation, the resulting completed B machine has:

```

OPERATIONS
  create_A(xx) =
    PRE xx : INT & objects /= Object_OBJ & bs = {}
    THEN
      ANY ax WHERE ax : Object_OBJ - objects
      THEN
        as := as \ { ax } || objects := objects \ { ax } ||
        x(ax) := xx
      END
    END;

  setx(ax,xx) =
    PRE ax : as & xx : INT & bs = {}
    THEN
      x(ax) := xx
    END;

  r1(ax) =
    PRE ax : as & x(ax) > 0 & not( #b.(b : bs & y(b) = x(ax)*x(ax)) ) &
      objects /= Object_OBJ & x(ax)*x(ax) : INT
    THEN
      ANY b WHERE b : Object_OBJ - objects
      THEN
        bs := bs \ { b } || objects := objects \ { b } ||
        y(b) := x(ax)*x(ax)
      END
    END
END

```

$r1$ defines the transformation step of the postcondition constraint. The machine is generated automatically by the UML-RSDS tools from the UML specification of the transformation¹. UML-RSDS encodes into B the semantics of all cases of updates to associations, including situations with mutually inverse association ends.

Using these machines we can verify syntactic correctness and semantic preservation properties of a model transformation, by means of *internal consistency* proof of the B machine representing the transformation and its metamodels. Internal consistency of a B machine consists of the following logical conditions:

- That the state space of the machine is non-empty: $\exists v.I$ where v is the tuple of variables of the machine, and I its invariant.

¹In practice, single-letter feature, variable and entity type names should be avoided, since these have a special meaning in B AMN.

- That the initialisation establishes the invariant: $[Init]I$
- That each operation maintains the invariant:

$$Pre \wedge I \Rightarrow [Code]I$$

where Pre is the precondition of the operation, and $Code$ its effect.

B machines implicitly satisfy the *frame axiom* for state changes: variables v which are not explicitly updated by an operation are assumed not to be modified by the operation. This corresponds to the assumption made in UML-RSDS that v is unmodified by activity act if $v \notin wr(act)$.

Proof of verification properties can be carried out using B, as follows (for separate models transformations):

1. Internal consistency proof of M_τ establishes that Inv is an invariant of the transformation.
2. By adding the axioms of Γ_T to the INVARIANT clause, the validity of these during the transformation and in the final state of the transformation can be proved by internal consistency proof, establishing syntactic correctness.
3. By adding φ and $\chi(\varphi)$ to the INVARIANT of M_τ , semantic preservation of φ can be proved by internal consistency proof. Creation and update operations to set up the source model must be suitably restricted by φ .
4. At termination of the transformation, all the application conditions of the transformation rules are false. The inference

$$(\bigwedge \neg (ACond)) \Rightarrow Post$$

can be encoded in the ASSERTIONS clause of M_τ and proved using the invariants $\Gamma_S \cup Inv \cup Asm0$.

Termination, confluence and semantic correctness proof needs to use suitable Q variants for each constraint, considered below.

In order to prove that a postulated Q measure is actually a variant function for a constraint, refinement proof in B can be carried out, with an abstraction of the transformation model machine M_τ defined as MO_τ :

```

MACHINE MOt SEES SystemTypes
VARIABLES /* variables for source model data */, q
INVARIANT
  /* typing of source model data */ &
  q : NAT
INITIALISATION
  es, q := {}, 0
OPERATIONS
  /* creation and update operations for source
     model: these set q arbitrarily in NAT */

  delta() =
    PRE q > 0
    THEN
      q ::= 0..q-1
    END
END

```

$delta$ represents a transformation step of the constraint for which q is the postulated variant. The operator $q :: s$ assigns an unspecified element of s to q . Each constraint C_i may have a corresponding variant q_i , the operation $delta_i$ for abstracted transformation steps of C_i then has the form:

```

delta_i() =
  PRE qi > 0 & qk = 0 /* for k < i */
  THEN
    qi :: 0..qi-1 || qj :: NAT /* for j > i */
  END

```

where the constraint implementations of C_1, \dots, C_{i-1} are designed to terminate prior to any execution of delta_i and therefore their variants are assumed to be 0 in the precondition of delta_i .

The original M_τ machine is then used to define a refinement of M_{0_τ} , with the refinement relation giving an explicit definition of the q_i variants. Refinement proof then attempts to verify that the explicit definition of each q_i obeys the abstract specification, ie, that it is strictly decreased by every execution of delta_i .

Refinement obligations in B are [8]:

- The joint invariants $Inv_A \wedge Inv_R$ of the abstract and refined machines are satisfiable together.
- The refined initialisation $Init_R$ establishes the invariants:

$$[Init_R] \neg [Init_A] \neg (Inv_A \wedge Inv_R)$$

- Each refined operation $PRE\ Pre_R\ THEN\ Code_R\ END$ satisfies the pre-post relation of its abstract version:

$$Inv_A \wedge Inv_R \wedge Pre_A \Rightarrow Pre_R \wedge [Code_R] \neg [Code_A] \neg (Inv_A \wedge Inv_R)$$

Refinement proof is usually manually intensive, with the majority of proof obligations requiring interactive proof.

To verify confluence of a transformation, it is sufficient to show that there is a unique state (up to structural isomorphism) where all the variants q_i have $q_i = 0$. This can be verified in the refinement by adding an *ASSERTIONS* clause of the schematic form:

ASSERTIONS

```
q1 = 0 & ... & qn = 0 => tstate = f(sstate)
```

This clause expresses that the target model state $tstate$ has specific values in terms of the source model state $sstate$, when all the q_i are zero.

The B tools will produce proof obligations for this assertion, and will act as a proof assistant in structuring the proof and carrying out routine proof steps automatically. A further consequence of the proof of the assertion is semantic correctness of the implementation: that the desired semantics of target model elements relative to source model elements also holds at termination.

The above procedures can be used as a general process for proving *Inv* and *Pres* properties, and for proving termination, semantic correctness and confluence, by means of suitable postulated Q variant functions. The techniques can be generalised to any transformation implementation whose transformation steps can be *serialised*, that is, each execution of the implementation is equivalent to one in which the computation steps occur in a strict sequential order. This assumption is made implicitly in the B model.

The size of the B formal model M_τ is linear in terms of the size of the model transformation τ , however the complexity of the transformation rules has a considerable effect on the proof effort required. Postcondition constraints using *forall* quantifications nested within *exists* in their succedents cannot be effectively verified, and the use of the conjunctive-implicative form pattern [13] is necessary instead. Recursive update operations cannot be represented. B is not suitable for establishing satisfiability properties asserting the existence of models of certain kinds, and tools such as Z3, UMLtoCSP [2], Alloy [1] or USE [4] are more appropriate for these.

12 Mapping to Z3

An option to generate Z3 theories for a system is also available. This produces type definitions and Z3 predicates for the class and use case constraints (but not for any operations). General use cases should have disjoint write and read frames, ie., update-in-place transformations are not supported.

Table 20 shows examples of the mapping of OCL expressions to Z3.

<i>OCL expression/operator e</i>	<i>Z3 expression/operator e'</i>
Integer	Int
Boolean	Bool
Real	Real
Entity type <i>E</i>	Sort E
Attribute <i>att</i> : <i>Typ</i> owned by <i>E</i>	function <i>att</i> : <i>E</i> → <i>Typ'</i>
Single-valued role <i>r</i> to <i>F</i> owned by <i>E</i>	function <i>r</i> : <i>E</i> → <i>F</i>
Collection-valued role <i>r</i> to <i>F</i> owned by <i>E</i>	function <i>r</i> : <i>E</i> → <i>List</i> (<i>F</i>)
forall	forall
exists	Expressed by skolemisation
first	head
prepend	insert
includes	memberE (for each entity type E)
Set{ }	nil
Set{ x1, ..., xn }	(insert x1' (... (insert xn' nil) ...))

Table 20: Mapping of UML and OCL to Z3

The most significant step in this translation is the expression of existential quantifiers by skolem functions: in a formula

$$E \rightarrow \text{forall}(x \mid \text{Cond} \Rightarrow F \rightarrow \text{exists}(y \mid \text{Pred}))$$

the existential quantifier is replaced by a new function $f_{\text{new}} : E \rightarrow F$ not occurring in any other part of the OCL or Z3 theories, and the formula is then translated as:

$$(\text{forall} ((x \ E)) (\Rightarrow \text{Cond}' \ \text{Pred}'[f_{\text{new}}(x)/y]))$$

In general, an *exists*-quantifier in the scope of several *forall*-quantifiers is replaced by a new function depending on all the *forall*-quantified types. This provides an explicit representation of the transformation as a set of (possibly under-specified) functions from source entities to target entities.

Sets and sequences are both modelled as Z3 lists. Operators for OCL select and collect are not in-built in Z3 and need to be defined using auxiliary functions. In addition, operators to check membership in a list and to obtain a list element by its index need to be added as auxiliary functions. An alternative to using lists to model OCL collections would be to use bitsets [21], however this involves a highly complex encoding and requires size bounds to be placed on entity type extents and collection sizes.

For separate-models transformations, or update-in-place transformations where the read and updated sets of entity types and features are disjoint, the mapping from UML-RSDS to Z3 performs the following translations:

- Each source and target language L (or language part) is represented by sorts E for each entity type E of the language, and maps of the form $f : E \rightarrow Typ$ for each owned feature f of E , together with Z3 encodings of the constraints of Γ_L for source languages L .
- The assumptions $Asm0$ on unmodified data are encoded and included.
- Each transformation postcondition *rule* of the form

$$E \rightarrow \text{forall}(x \mid SCond \Rightarrow F \rightarrow \text{exists}(y \mid PCond))$$

is encoded as a function $taurule : E \rightarrow F$ and a predicate

$$\forall x : E \cdot SCond' \Rightarrow PCond'[taurule(x)/y]$$

- The invariant expressing the inverse of *rule* is encoded by a function $sigmarule : F \rightarrow E$ and a predicate

$$\forall y : F \cdot SCond'[sigmarule(y)/x] \text{ and } PCond'[sigmarule(y)/x]$$

For separate-models transformations, the consistency of the resulting theory Γ_τ with individual constraints $\varphi \in \Gamma_T$ can be checked by adding φ' to the theory, counter-examples can be searched for by instead adding $\neg \varphi'$.

Syntactic correctness can be shown by adding the negation of $\bigwedge \Gamma_T$ for the target language T , and establishing that the resulting theory is unsatisfiable. Semantic preservation of φ can then be shown by adding Γ_T , φ and the negation of $\chi(\varphi)$ and showing unsatisfiability. Unlike FOL or B, Z3 does not contain proof techniques for unbounded induction, and so may fail to establish true inferences which could be proved within FOL or B. In addition it provides decision procedures for only some of the decidable subsets of first order logic. There may therefore be situations where Z3 may neither demonstrate counter-examples to a theory, nor establish its validity, and for these cases analysis using proof in B or another theorem-prover will be necessary.

The mapping from UML and OCL to Z3 has been automated in the UML-RSDS tools.

As in first-order set theory, Z3 has no notion of undefined values, unlike in standard OCL. Division in Z3 is a total function, but its value is unspecified for division by 0. A transformation specifier should ensure that all expressions within a transformation invariant, assumption or rule relation Cn have a defined value, ie., $def(Cn)$ is true, whenever the predicate may be evaluated. When reasoning about transformations using Z3, the results are only meaningful for transformation implementations which satisfy this condition, ie., where no expression evaluation to OCL *invalid* ever occurs. In Z3 it is also assumed that all sorts are non-empty, ie., that the transformation is not applied to trivial models.

There is no inbuilt Z3 representation for strings: these can be modelled as an unspecified sort or as lists of integers. There is no direct representation for subtyping.

An example of mapping OCL to Z3 is our transformation example from Figure 12. The corresponding Z3 theory is:

```
(declare-sort A)
(declare-sort B)
(declare-fun x (A) Int)
(declare-fun y (B) Int)
```

This expresses the theory of the class diagram. For the transformation postcondition a new skolem function is introduced, and an axiom for the skolemised postcondition:

```
(declare-fun tau1 (A) B)
(assert (forall ((a A))
  (=> (> (x a) 0) (= (y (tau1 a)) (* (x a) (x a))))))
```

Note that this is not within a decidable subset of first-order logic, since it involves non-linear arithmetic.

The invariant is expressed by:

```
(declare-fun sigma1 (B) A)
(assert (forall ((b B))
  (and (> (x (sigma1 b)) 0) (= (y b) (* (x (sigma1 b)) (x (sigma1 b)))))))
```

This combined theory expresses the state at termination of the transformation, and also defines precisely how the terminal state of the system should relate to the starting state, by relating the modified entity types and features $\{B, y\}$ to the unchanged values of $\{A, x\}$.

Consistency analysis shows that this combined theory is indeed consistent. If we add the additional constraint

```
(not (forall ((b B)) (> (y b) 2)))
```

a counter-example to $B \rightarrow \text{forAll}(b \mid b.y > 2)$ is found, where b is derived from $a1 : A$ with $a1.x = 1$.

An alternative approach for representing UML and OCL in Z3 is given in Appendix H.

13 Mapping to SMV

We utilise the SMV/nuSMV language and model checker [17] to analyse temporal properties of systems. We select SMV because it is an established industrial-strength tool which supports linear temporal logic (LTL). SMV models systems in *modules*, which may contain variables ranging over finite domains such as subranges $a..b$ of natural numbers and enumerated sets. The initial values of variables are set by initialisation statements $\text{init}(v) := e$. A $\text{next}(v) := e$ statement identifies how the value of variable v changes in execution steps of the module. In the UML to SMV encoding objects are modelled by integer object identities, attributes and associations are represented as variables, and classes are represented by modules parameterised by the object identity values (so that separate copies of the variables exist for each object of the class). A specific numeric upper bound must be given for the number of possible objects of each class. The bound is specified in the class creation dialog (Figure 2). A UML-RSDS system is modelled in SMV by modelling system execution steps as module execution steps.

The structure of an SMV specification of a class diagram is as follows:

```
MODULE main
VAR
  C : Controller;
  MEntity1 : Entity(C,1);
  .... object instances ....

MODULE Controller
VAR
  Entityid : 1..n;
  event : { createEntity, killEntity, event1, ..., eventm };

MODULE Entity(C, id)
VAR
  alive : boolean;
  ... attributes ...
DEFINE
  TcreateEntity := C.event = createEntity & C.Entityid = id;
  TkillEntity := C.event = killEntity & C.Entityid = id;
  Tevent1 := C.event = event1 & C.Entityid = id & alive = TRUE;
  ...
ASSIGN
  init(alive) := FALSE;

  next(alive) :=
    case
```

```

TcreateEntity : TRUE;
TkillEntity : FALSE;
TRUE : alive;
esac;

```

Each class *Entity* is represented in a separate module, and each instance is listed as a module instance in the main module. System events are listed in the Controller, and the effect of these events on specific objects are defined in the module specific to the class of that object. The value of the *Controller* variable *event* in each execution step identifies which event occurs, and the value of the appropriate *Eid* variable indicates which object of class *E* the event occurs on.

Associations $r : A \rightarrow B$ of 1-multiplicity are represented as SMV module attributes

```
r : 1..bcard
```

where *bcard* is the maximum permitted number of objects of *B*. Events to set and unset this role are included. 0 represents the null object and is used as the lower bound for 0..1-multiplicity roles. For *-multiplicity unordered roles $r : A \rightarrow Set(B)$, an array representation is used instead:

```
r : array 1..bcard of 0..1
```

with the presence/absence of a B element with identity value *i* being indicated by $r[i] = 1$ or $r[i] = 0$. Operations to add and remove elements are provided. The Controller Aid and Bid variables identify which links are being added/removed.

For transformations, this mapping needs to be extended as follows. Source entities are modelled by SMV modules with frozen attributes: all source model data has values fixed at the initial time point. If a transformation rule *r* iterates over $s : S_i$ and has the form

$$Ante \Rightarrow T_j \rightarrow exists(t \mid TCond \ \& \ P)$$

then the SMV module for T_j has a parameter of module type S_i , and *r* is an event of the *Controller*, and S_iid is a Controller variable.

The T_j module has a transition defined as

```
Tr := C.event = r & C.Siid = id
```

identifying that the event for *r* takes place on the S_i object with *id* equal to the current T_j object. The updates to features *f* of *t* defined in *TCond* and *P* are then specified by *next(f)* statements, using *Tr* as a condition, and *next(alive)* is set to *TRUE* under condition *Tr*.

The restrictions of SMV/nuSMV imply that only attributes and expressions of the following kinds can be represented in SMV:

- Booleans and boolean operations.
- Integers and operations on integers within a bounded range.
- Strings represented as elements of an enumerated type. String operations cannot be represented.

14 Interactions

Interactions can be created as UML sequence diagrams, these give examples of system behaviour in terms of object communications.

The editor for interactions is shown in Figure 14.

The main options are:

File menu – options to save, load and print models.

Create menu – options to create lifelines, messages, states, execution instances, and annotations.

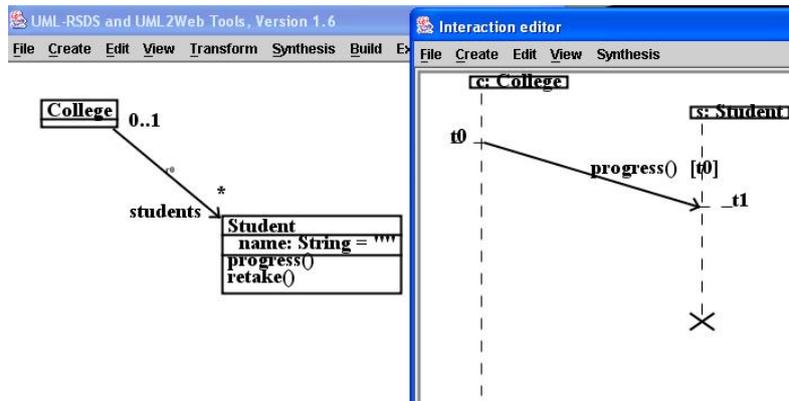


Figure 14: Interactions editor

Edit menu – options to edit an interaction.

View menu – options to view lifelines, messages, etc.

Synthesis menu – options to generate the formal real-time logic meaning of the diagram.

15 Interoperation with Eclipse

Eclipse is the leading MDD toolset and provides many tools for creating and manipulating models, including the ATL transformation language.

Metamodels and models can be imported and exported from Eclipse in XML text format, and these can also be imported and exported from UML-RSDS, providing a means to exchange data between Eclipse and UML-RSDS.

15.1 Exchanging metamodels

Metamodels are exported from Eclipse in XML Ecore format, consisting of header lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" ... name="MM" ... >
```

where most entries are standard, but others refer to the name MM of the metamodel (the file is called MM.ecore).

Each XML element within the ecore:EPackage element then describes either a class, eg:

```
<eClassifiers xsi:type="ecore:EClass" name="A">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType=... />
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="x" eType=... />
</eClassifiers>
```

or an enumerated type.

The associations between classes are also listed in the class description, as eStructuralFeatures of xsi:type ecore:EReference. Metamodels of the above format can be read in to UML-RSDS using the *Import from ECore* option on the File menu.

In the opposite direction, such files can also be exported from UML-RSDS by the *Save ECore* option on the *Save* menu. The file and metamodel are called *mm* by default. The eClassifier descriptors correspond to the entity types and enumerated types in the UML-RSDS model, except that an additional *model* eClassifier is introduced, with many-valued reference links *es* to each concrete entity type *E*. This class represents the model as a single object.

15.2 Exchanging models

There are two main formats of XML Model Interchange (XMI) data files for models. The first consists of a list of model element descriptors implicitly numbered (for each concrete class) from 0 upwards. These elements are then referred to elsewhere in the file by this number. Eg., the first object of class *A* could be described by the line

```
<as name="a1" x="5"/>
```

This is then referred to elsewhere in the file as `//@as.0`, eg., if each *B* had a list *ar* of *A* objects, a *B* element descriptor could be:

```
<bs name="b1" y="7" ar="//@as.0 //@as.1"/>
```

An alternative format is to allocate a unique `xmi:id` value to each element, and use this to cross-reference elements. In this format, the above A and B objects could be, for example:

```
<as xmi:id="$11" name="a1" x="5"/>
<bs xmi:id="$6" name="b1" y="7" ar="$11 $9"/>
```

Either format can be used as input in a file *xsi.txt* for the *Convert* option on the UML-RSDS File menu: the output will be produced in model data format in *model.txt*. The above XML would produce the data:

```
as0 : A
...
bs0 : B
...
as0.name = "a1"
as0.x = 5
...
bs0 : B
...
bs0.name = "b1"
bs0.y = 7
as0 : bs0.ar
as1 : bs0.ar
```

The metamodel for the language concerned must be loaded before the conversion is applied.

There are two convert options: the first applies an XML parser to the entire *xsi.txt* file, it is therefore suitable for smaller files (eg., up to 100Kb), where the source formatting is not strict. The second only parses one line at a time and is appropriate for larger files where each element descriptor strictly occupies one line only.

Models are exported from UML-RSDS data format to XMI format by the generated GUI executable of a system: a model in file *in.txt* can be read into memory by the *loadModel* option, the *saveModel* option then outputs the model to *xsi.txt* in XMI format (the first of the two formats described in this section).

16 Extensions and importing libraries

The UML-RSDS language can be extended with new binary and unary \rightarrow operators, and external libraries and classes may be declared for use by a UML-RSDS specification. For example, to use Java regular expressions, the `java.util.regex.*` library package would be declared as a new import, and the classes `Pattern` and `Matcher` of this package declared as *external* classes, along with any methods that we wish to use from them, eg., the static `matches(p : String, s : String) : boolean` operation of `Pattern`.

These extension/import mechanisms are usually language-specific, and there is no support for formal analysis of system elements that depend upon extension/imported components. They are a convenience to support rapid code generation of systems.

New plugins can be defined and used from the UML-RSDS tools, in order to support new code generators and the use of DSLs.

16.1 External classes and operations

Each external class used by the system should be added to the class diagram, with the stereotype *external*. Any operations required should also be added to their class. Note that currently, array input and output parameters are not permitted, and only the types supported by UML-RSDS (boolean, int, long, double, String, enumerations and class types) can be used for parameters of such operations.

In expressions, declared operations of external entities may be invoked as with any other operation, eg., `result = Pattern.matches(".a.", "bar")`. Their form in generated Java code is the same as in the UML-RSDS expression.

Instances of external entity types may be created locally, for example, a statement

```
p : Pattern := Pattern.compile(".a.")
```

defines a local variable *p* holding a *Pattern* object.

For Java code generation, provided that the external entity type *E* has a zero-argument constructor, objects of the entity may be created in constraints by using an $E \rightarrow exists(e \mid \dots)$ construct. For example:

```
Date->exists( d | d.getTime()->display() )
```

The generated code initialises *d* with the *Date()* constructor.

For Java code generation, existing methods of the Java String class may be called without declaration, eg., `str.replaceAll(p, s)` can be used as an expression for a Java system.

External systems/applications *M* used by a UML-RSDS system are defined as classes with the stereotype *externalApp*. This signifies that *M* has been developed as a UML-RSDS application with system name *M*. For Java implementations, the code of *M*'s classes should be in a subdirectory with this name. Calls $M.op(p)$ within the client system are interpreted as $M.Controller.inst().op(p)$ in the complete generated code. The operations of *M* that are used in the client system should be declared in the *M* class in the client system class diagram.

16.2 Adding an import

The “Add import” option allows the user to enter an additional library package. The exact text required should be entered, eg.:

```
import java.util.regex.*;
```

for a Java import.

16.3 Adding a new operator

New unary and binary \rightarrow operators can be added. The dialog asks for the name of the operator, including the arrow, and the result type. For an operator application $_1 \rightarrow op(_2)$ the default text in Java and C# is $_1.op(_2')$, where $_i'$ is the translation of $_i$. If an alternative is needed, this should be specified in the following text areas. For example, an operator $_1 \rightarrow time()$ to print out the time after a string message could be given the Java template text

```
(_1 + (new Date()).getTime())
```

16.4 Libraries

The following libraries are defined as metamodel files in *uml2web/libraries*:

- *mathlibmm.txt* – mathematical functions such as gcd, factorial, etc
- *statlibmm.txt* – statistical functions
- *seqmatlibmm.txt* – sequence and matrix functions
- *finlibmm.txt* – financial functions
- *stringlibmm.txt* – string functions
- *gamm.txt* – the framework of a genetic algorithm.

ocl.h defines OCL operators in ANSI C, whilst *ocl.py* defines them in Python. These are used by the implementations of UML-RSDS applications in these languages.

The *excel.mm* file in the */output* directory defines Excel functions which can be used in spreadsheet specifications *mm.csv* to be loaded by the Load CSV option.

In a *mm.txt* file the instruction

```
Import:  
f.txt
```

loads in the metamodel file in *f.txt* before resuming the loading of *mm.txt*.

16.5 Plugins and DSLs

Developers may add plugins *f.jar* by placing them in a */f* subdirectory of the UML-RSDS tool directory. They then appear as an option on the Extensions menu.

To create a new plugin from a UML-RSDS specification, set the name of the application (File menu), generate the design and code, which will be saved in the *name* directory. The *Run* option will compile the code and build *name.jar* in the *name* directory, and then execute the jar file.

To define a code generator for a new programming language, use the structure of *uml2Camm.txt* and *uml2Cbmm.txt* (for a case where a model of the target language needs to be built) or of *uml2pymm.txt* (for model to code mappings). You will need to construct an OCL library in your target language (as in *ocl.h* and *ocl.py*). The code generator uses design data file output by the “save as design” option of UML-RSDS, which produces a file *output/model.txt*.

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 18 as an abstract syntax for pseudocode statements), the concrete text syntax is given by text lines of the three forms *object : Entity*, *object.feature = value* and *object1 : object2.feature*. 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 using the DSL metamodel as their source language.

17 The software development process using UML-RSDS

UML-RSDS supports model-driven development and the OMG Model-driven Architecture (MDA). In terms of MDA, UML-RSDS permits the definition of PSMs for the Java/C#/C++ family of languages. These PSMs can then be automatically mapped to code in any of these languages (Figure 15).

The model structure corresponds closely with the code structure (each model class becomes a programming language class, each model feature becomes a programming language feature, etc),

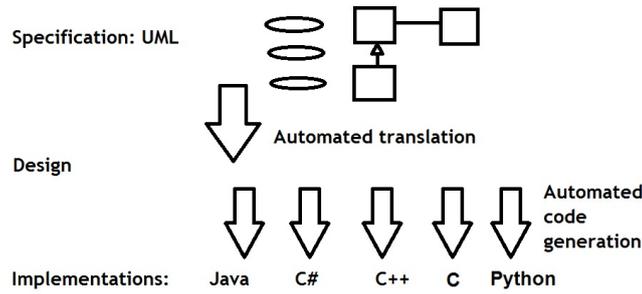


Figure 15: UML-RSDS process

but a high level of abstraction may be used to specify functionality, as logical postconditions of operations and use cases. Automatically generated code carries out aspects such as the maintenance of inverse associations and correct deletion of objects, in addition to evaluating a large part of the OCL library expression language.

The structure of the generated system is shown in Figure 16. The *Controller.java* class contains the operations corresponding to the use cases of the system, and invokes operations of classes representing the entity types of the system. The elaborate structure of this class is necessary to ensure that class invariants are globally enforced.

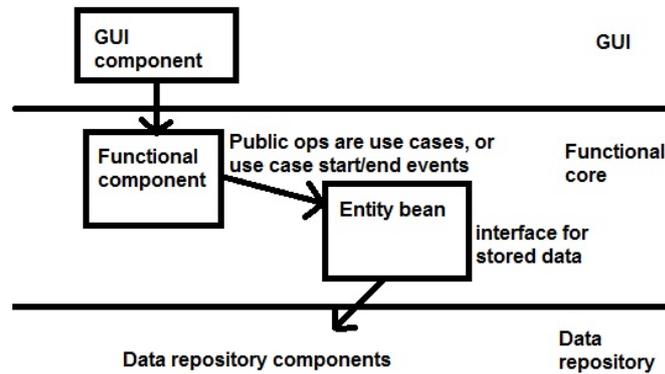


Figure 16: UML-RSDS code architecture

Maintenance of the system should be carried out by modifying the PSM and then regenerating the code: ideally, developers should never need to even look at the generated source code, let alone edit it.

Platform-independent models could also be defined, these would consist entirely of a list of high-level operations (use cases) without any data model, but with text descriptions and parameters for the use cases. These PIMs define the interface of the system when it is viewed as a component within a larger system, ie., the PIM defines the services that the system provides as a black box. If system name M is specified for the component system, then a PIM interface for M can be defined as an *externalApp* class in the client system class diagram, with declarations and outline definitions of its supplied services given as operations.

18 Current state of the tools

The tools are available for download as a jar file: `umlrsds.jar`, as part of the `umlrsds.zip` download. The jar can be executed from the command line as

```
java -jar umlrsds.jar
```

A writable subdirectory *output* needs to exist in the directory where `umlrsds.jar` is executed. This is already provided in the zip file, along with libraries and the C and Python code generators.

18.1 Planned extensions

Some new features which are being introduced, and are present in prototype versions in the latest version of the tools, are:

- *obj.oclAsType(T)* and *obj.oclIsKindOf(T)* for entity types *T*.
- Import of ecore files defining metamodels, in XML format. This will permit transfer of Eclipse metamodels to UML-RSDS.
- Export of metamodels in USE [4] tool format.
- Export/import and editing of metamodels in KM3 format.
- Generation of Z3 theories [22] for systems.
- Generation of ANSI C and Python code for systems.
- Import of external libraries (currently *Date* and *XMLParser*) for use within a system: new libraries can also be defined by creating classes with the stereotype *external*. Libraries could include database interfaces or user interface components.
- Definition of systems in packages, by using the ‘Set name’ option on the main File menu. The name is used as the package name.
- Extension of *select*, *collect* and *reject* to work on collections of collections and collections of primitive values.
- A binary version of *any*, ie.: $s \rightarrow any(P)$.
- Inheritance of constraints/postconditions in use cases.
- Conditional expressions *if E then E1 else E2 endif*.
- *oclIsUndefined()*

long integers have been available from version 1.5.

18.2 Release version history

From version 1.4 we support *forall*, *exists*, *exists1* quantification over *Integer.subrange(st, en)* sequences, eg:

$$result = Integer.subrange(1, 10) \rightarrow exists(x \mid x * x > 50)$$

and *Set(E)* for entities *E*, for both query and update uses, eg:

$$(s \rightarrow subcollections()) \rightarrow forall(x \mid x \rightarrow display())$$

exists quantification is supported for query uses.

From version 1.6, *select*, *collect* and *reject* can be used with collections of numbers or strings, or collections of collections, in addition to collections of class instances. Some translations (eg., to C or Z3) do not support such collections.

We also support ordered *forall* quantification over entities, this may be specified when a constraint is created, and indicates that the executable implementation of the quantification should iterate over the collection elements in ascending order of the specified expression on these elements (this feature is for type 1, 2 and 3 postcondition constraints for general use cases only).

Version 1.6 introduces a translation to Java 7, including the use of generic types and operations. This means that an OCL type such as *Sequence(int)* is represented by *ArrayList < Integer >* in the Java code.

A requirements editor using SysML notation is available from version 1.6, as shown in Figure 17.

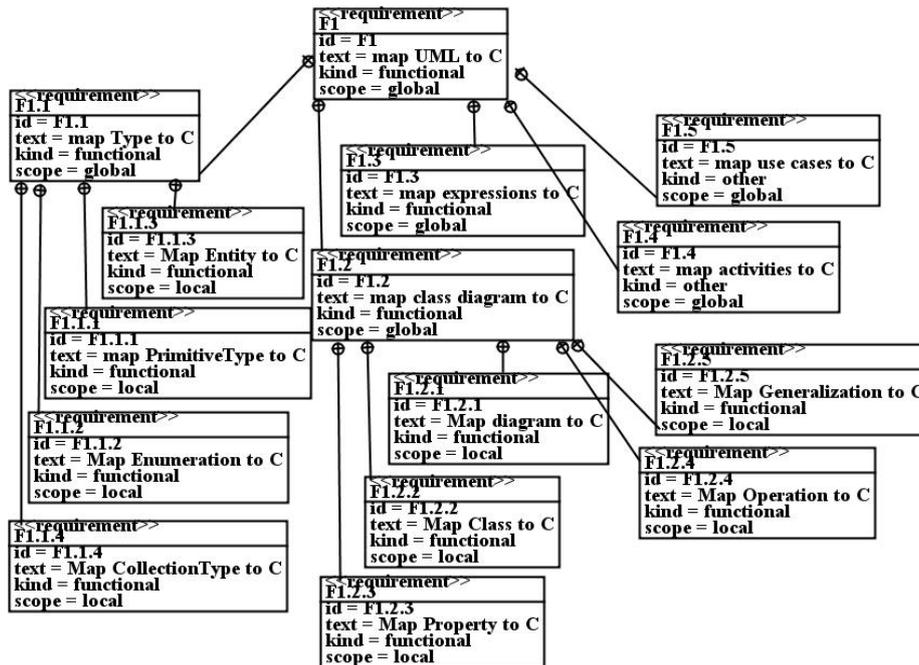


Figure 17: Requirements editor

In version 1.6, a text editor for use cases has been added (option Edit use case on the Edit menu), which allows all the postconditions of a use case to be edited at once, to be checked before saving, and to support copy/paste operations.

Version 1.7 introduces a translation to ANSI C, and input/output of model data from CSV files. Multi-valued attributes are also supported.

Version 1.8 provides improved support for mapping ATL, ETL and QVT-R to UML-RSDS, and various usability improvements.

Version 1.9 introduces a Python code generator and a tool for synthesising transformations from metamodels (Appendix J). Support for QVT-R has been extended. Prototype support for a *Map* collection type is included.

Projected for Version 2.0 are Android app synthesis, model-based testing facilities, an improved KM3 editor, and metamodel synthesis from models.

18.3 Known bugs

Current deficiencies with the tools are:

- Initial values of attributes are not stored in mm.txt, nor are constraint names.
- Running ‘Generate Design’ multiple times on the same specification can produce erroneous designs with duplicated variables.
- Inheritance of use cases has no effect – it should permit superposition of use cases, with postconditions of the parent use case inherited by its sub-use cases unless these have a same-named postcondition, which is used instead.
- select/reject/collect cannot be applied to collections of primitive types or strings, only to collections of objects.
- Let variables and other variables introduced in the assumption of a use case postcondition should have distinct names from those in other postconditions of the same use case, eg.:

```
a = Set{1,2,3} => a->sum()->display()
```

```
b = Set{1,2,3} => b->prd()->display()
```

a could not be used also in the second constraint.

- Let variables cannot be defined to be of nested collection types, eg:

$$mm = Sequence\{Sequence\{1, 2\}, Sequence\{2, 6\}\}$$

- Active classes can only be implemented using Java, there is no C#, C++, C or Python implementation.

Nested select/reject/collect operators should not use naked features of external classes in the inner operators: in an expression of the form

```
col1->select(rs = col2->select( f = g ))
```

where *col1* has element type *E1*, *rs* is a feature of *E1*, *col2* has element type *E2*, and *f* is a feature of *E2*, then *g* should be a feature (or an expression using only features) of *E1* or *E2* and not of some external class *E3*.

Translation to B of select, collect, reject, unionAll, intersectAll and related operators is only supported when these operate over sets, not sequences. SelectMaximals, selectMinimals, sort and sortedBy are not supported for B translation.

The C and Python translations have the following limitations:

1. *select*, *reject*, *collect*, *forAll*, *exists*, *exists1* should always use an iterator variable. In query mode, the predicate of *forAll*, *exists*, *exists1* can only refer to data of that iterator variable.
2. The *cached* operation stereotype is not yet supported.

References

- [1] K. Anastakis, B. Bordbar, J. Kuster, *Analysis of Model Transformations via Alloy*, Modevva 2007.
- [2] J. Cabot, R. Clariso, D. Riera, *UMLtoCSP: a tool for the verification of UML/OCL models using constraint programming*, Automated Software Engineering '07, pp. 547–548, ACM Press, 2007.
- [3] M. Davis, V. Lo, *Infectious Defaults*, Quantitative Finance, vol. 1, no. 4, 2001, pp. 382–387.
- [4] M. Gogolla, J. Bohling, M. Richters, *Validating UML and OCL models in USE by automatic snapshot generation*, Software and Systems Modeling, vol. 4, no. 4, pp. 386–398, 2005.
- [5] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, *ATL: A model transformation tool*, Sci. Comput. Program. **72**(1-2) (2008) 31–39.

- [6] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, *Evaluation of model transformation approaches for model refactoring*, Science of Computer Programming, 2013, <http://dx.doi.org/10.1016/j.scico.2013.07.013>.
- [7] D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, in ICMT 2008, LNCS Vol. 5063, pp. 46–60, Springer-Verlag, 2008.
- [8] K. Lano, *The B Language and Method*, Springer-Verlag, 1996.
- [9] K. Lano, D. Clark, *Direct semantics of extended state machines*, Journal of Object Technology, vol. 6, no. 9, 2007.
- [10] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, *Comparative evaluation of model transformation specification approaches*, International Journal of Software Informatics, Vol. 6, Issue 2, 2012.
- [11] K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparing verification techniques for model transformations*, Modevva workshop, MODELS 2012.
- [12] K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, February 2013.
- [13] K. Lano, S. Kolahdouz-Rahimi, *Optimising Model-transformations using Design Patterns*, MODEL-SWARD 2013.
- [14] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, *Analysis of hybrid model transformation language specifications*, FSEN 2015.
- [15] K. Lano, *Agile model-based development using UML-RSDS*, CRC Press, 2016.
- [16] K. Matsuda et al., *Bidirectionalisation transformation based on automatic derivation of view complement functions*, ICFP '07, 2007.
- [17] NuSMV, <http://nusmv.fbk.eu>, 2015.
- [18] OMG, *MOF 2.0 Query/View/Transformation Specification v1.1*, 2011.
- [19] OMG, *Object Constraint Language 2.4 Specification*, 2014.
- [20] L. Rose, D. Kolovos, R. Paige, F. Polack, S. Poulding, *Epsilon Flock: A model migration language*, SoSyM vol. 13, no. 2, 2014.
- [21] M. Soeken, R. Wille, R. Drechsler, *Encoding OCL data types for SAT-based verification of UML/OCL models*, University of Bremen, 2012.
- [22] Z3 Theorem Prover, <http://research.microsoft.com/en-us/um/redmond/projects/z3/>, 2012.

A Reserved words

The UML-RSDS reserved words are the same as for OCL, in addition operator names introduced in UML-RSDS, such as “selectMaximals” should not be used as identifiers. The “ \wedge ”, “#”, “!”, “?” and “_” characters cannot be used within identifiers, nor can other arithmetic, logical or punctuation symbols.

On the other hand, the “\$” symbol can be used as a package name prefix for identifiers: “*m1\$Entity*” for example. “.” is used to separate objects and feature names.

Identifiers should avoid the use of “_” if possible, since this is used by the tools for generated identifiers. Currently the tools do not accept this as a valid symbol in user identifiers. Lowercase versions *e* of entity names *E* are used as operation variables *ex* in place of *self*, and as sets *es* of all instances of *E* in *Controller*, so features with names *ex* and *es* should also be avoided.

If a system will be used for synthesis of Java, then Java keywords and library class names should be avoided in identifiers and names of use cases: “Class”, “Package”, “class”, “final”, etc. Likewise for synthesis of other programming languages.

For synthesis of B, keywords of B, such as “MACHINE”, should be avoided. In SMV, “next” and “self” are keywords. All identifiers for B translation should contain two or more characters. \$ is not valid as a symbol within identifiers in B.

B Expression syntax of UML-RSDS

UML-RSDS uses both classical set theory expressions and OCL. It only uses sets and sequences, and not bags or ordered sets, unlike OCL. It provides most of the OCL 2.4 standard library operators for sets and sequences. Symmetric binary operators such as \cup and \cap can be written in the classical style, rather than as operators on collections. Likewise for the binary logical operators. There is no null or undefined element. Maps are supported in prototype form (Version 1.9).

<code>< expression ></code>	<code>::=</code>	<code>< bracketed_expression > < equality_expression > </code> <code>< logical_expression > < factor_expression ></code>
<code>< bracketed_expression ></code>	<code>::=</code>	<code>"(" < expression > ")"</code>
<code>< logical_expression ></code>	<code>::=</code>	<code>< expression > < logical_op > < expression ></code>
<code>< equality_expression ></code>	<code>::=</code>	<code>< factor_expression > < equality_op > < factor_expression ></code>
<code>< factor_expression ></code>	<code>::=</code>	<code>< basic_expression > < factor_op > < factor_expression > </code> <code>< factor2_expression ></code>
<code>< factor2_expression ></code>	<code>::=</code>	<code>< expression > < unary_operator > "(" < expression > ")" </code> <code>< expression > < "-" > exists(" < identifier > " " < expression > ")" </code> <code>< expression > < "-" > exists1(" < identifier > " " < expression > ")" </code> <code>< expression > < "-" > forAll(" < identifier > " " < expression > ")" </code> <code>< expression > < "-" > exists(" < expression > ")" </code> <code>< expression > < "-" > exists1(" < expression > ")" </code> <code>< expression > < "-" > forAll(" < expression > ")" </code> <code>< expression > < "-" > select(" < expression > ")" </code> <code>< expression > < "-" > select(" < identifier > " " < expression > ")" </code> <code>< expression > < "-" > reject(" < expression > ")" </code> <code>< expression > < "-" > reject(" < identifier > " " < expression > ")" </code> <code>< expression > < "-" > collect(" < expression > ")" </code> <code>< expression > < "-" > unionAll(" < expression > ")" </code> <code>< expression > < "-" > intersectAll(" < expression > ")" </code> <code>< expression > < "-" > < binary_operator > "(" < expression > ")" </code> <code>< basic_expression ></code>
<code>< basic_expression ></code>	<code>::=</code>	<code>< set_expression > < sequence_expression > </code> <code>< call_expression > < array_expression > </code> <code>< identifier > < value ></code>
<code>< set_expression ></code>	<code>::=</code>	<code>"Set{" < fe_sequence > "}"</code>
<code>< sequence_expression ></code>	<code>::=</code>	<code>"Sequence{" < fe_sequence > "}"</code>
<code>< call_expression ></code>	<code>::=</code>	<code>< identifier > "(" < fe_sequence > ")"</code>
<code>< array_expression ></code>	<code>::=</code>	<code>< identifier > "[" < factor_expression > "]" </code> <code>< identifier > "[" < factor_expression > "]" < identifier ></code>

A `< unary_operator >` is one of: *any*, *size*, *isDeleted*, *display*, *min*, *max*, *sum*, *prd*, *flatten*, *sort*, *asSet*, *asSequence*, *sqrt*, *sqr*, *last*, *first*, *front*, *tail*, *closure*, *characters*, *subcollections*, *reverse*, *isEmpty*, *notEmpty*, *toUpperCase*, *toLowerCase*, *isInteger*, *isReal*, *toInteger*, *toReal*. The mathematical functions *ceil*, *round*, *floor*, *exp*, etc. can also be written as unary expressions, which is convenient if their argument is a function call or a complex bracketed expression.

Other unary and binary operators may be used in a `< factor2_expression >`, as described in Tables 3, 4, 5, 6 and 7. Other binary operators are *includes*, *including*, *excludes*, *excluding*, *union*, *intersection*, *selectMaximals*, *selectMinimals*, *includesAll*, *excludesAll*, *append*, *prepend*, *count*, *hasPrefix*, *hasSuffix*, *indexOf*, *sortedBy*, *at*. Ternary operators are expressed as operation calls, eg., *s.subrange(i, j)*, *s.insertAt(i, x)*.

A `< logical_op >` is one of `=>`, `&`, or `or`. An `< equality_op >` is one of `=`, `/=`, `>`, `<`, `<:` (subset-or-equal), `<=`, `>=`, `:`, `/:` (not-in). A `< factor_op >` is one of `+`, `/`, `*`, `-`, `\/` (union), `^` (concatenation of sequences), `/\` (intersection). An `< fe_sequence >` is a comma-separated sequence of factor expressions. Identifiers can contain `"` (to denote reference to a feature of an object), but not as the first or last character, and occurrences must be separated by at least one other character. Identifiers can also contain `"$"`.

Valid function symbols are the numeric functions such as *sqrt*, *floor*, *abs*, and OCL operators such as *oclIsKindOf*, *oclAsType*, *isReal*, *isInteger*, *toReal*, *toInteger*.

C Activity syntax of UML-RSDS

The following concrete syntax is used for a subset of UML structured activities:

```

< statement > ::= < loop_statement > | < creation_statement > |
                < conditional_statement > | < sequence_statement > |
                < basic_statement >
< loop_statement > ::= "while" < expression > "do" < statement > |
                    "for" < expression > "do" < statement >
< conditional_statement > ::= "if" < expression > "then" < statement >
                            "else" < basic_statement >
< sequence_statement > ::= < statement > "," < statement >
< creation_statement > ::= "var" < identifier > ":" < identifier >
< basic_statement > ::= < basic_expression > "==" < expression > | "skip" |
                    < identifier > ":" < identifier > "==" < expression > |
                    "execute" < expression > |
                    "return" < expression > | "(" < statement > ")" |
                    < call_expression >

```

Spaces are necessary around all the keywords and operators such as ; and :. Brackets should be avoided unless necessary. For statement tests have the form *variable* : *collection* to iterate over all elements in *collection* (in the order of the collection, if it is ordered). The variable is introduced by this test. The syntax corresponds to the metamodel of Figure 18.

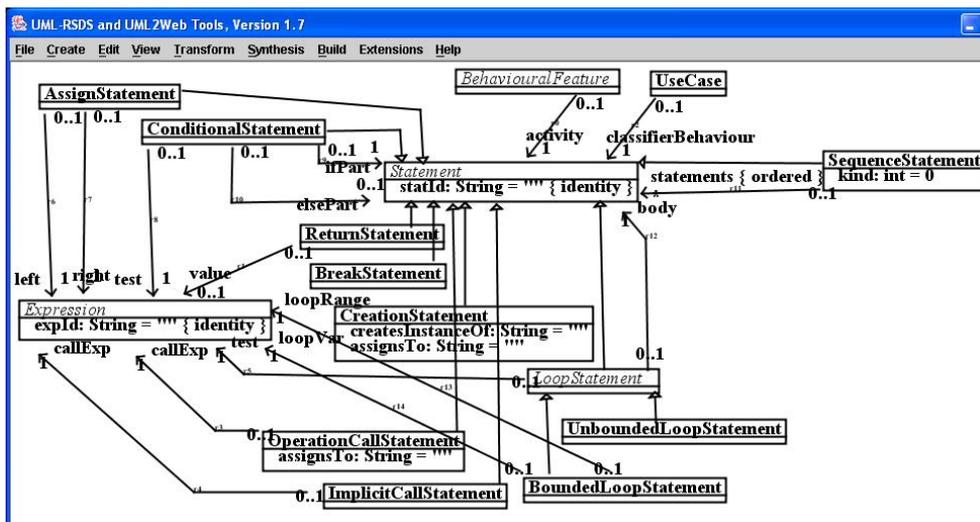


Figure 18: UML-RSDS activity metamodel

Example statements are:

```

var x : int ;
x := 0 ;
for i : Integer.subrange(1,10)
do
  ( x := x + i ;
    if x mod 2 = 0
    then
      displayint(x)
    else skip
    ) ;
return x

```

D Mappings from UML-RSDS to Java, C#, C, C++ and Python

The following tables show how UML and OCL elements are interpreted via the different implementation mappings.

UML/OCL	Java 4	Java 6	C#	C++
String	String	String	string	string
Boolean	boolean	boolean	bool	bool
Entity type A	A	A	A	A*
Sequence(A)	Vector	ArrayList	ArrayList	vector< A* > *
Set(A)	Vector	HashSet	ArrayList	set< A* > *
col->size()	col.size()	col.size()	col.Count	col->size()
sq->at(i)	sq.get(i-1)	sq.get(i-1)	sq[i-1]	(*sq)[i-1]
st->includes(x)	st.contains(x)	st.contains(x)	st.Contains(x)	st->find(x) != st->end()
Set st				find(sq->begin(), sq->end(), x) != sq->end()
sq->includes(x)	sq.contains(x)	sq.contains(x)	sq.Contains(x)	
Sequence sq				
st1 = st2	Set. equalsSet(st1,st2)	st1.equals(st2)	SystemTypes. equalsSet(st1,st2)	*st1 == *st2
Sets st1, st2				
sq1 = sq2	sq1.equals(sq2)	sq1.equals(sq2)	sq1.Equals(sq2)	*sq1 == *sq2
Sequences sq1, sq2				

Table 21: Mappings from UML/OCL to Java, C#, C++

In Java 7, *Sequence(A)* is represented by *ArrayList < A >*, and *Set(A)* by *HashSet < A >*. In C, entities are mapped to structs, and String to char*. The details of the mapping are in <https://nms.kcl.ac.uk/kevin.lano/uml2C.pdf>.

OCL *and* (UML-RSDS &) is interpreted by && in each language, and *or* by ||. The *int* type of UML-RSDS is interpreted by *int* in each language, likewise for *long*, *double*.

UML/OCL	Java 7	Python	C
String	String	str	char*
Boolean	boolean	bool	unsigned char
Entity type A	A	A	A*
Sequence(A)	ArrayList< A >	list	A**
Set(A)	HashSet< A >	set	A**
col->size()	col.size()	len(col)	length(col)
sq->at(i)	sq.get(i-1)	sq[i-1]	(et) at((void**) sq, i)
sq element type et			
st->includes(x)	st.contains(x)	x in st	isIn((void*) x, (void**) st)
Set st			
sq->includes(x)	sq.contains(x)	x in sq	isIn((void*) x, (void**) sq)
Sequence sq			
st1 = st2	st1.equals(st2)	st1 == st2	equalsSet((void**) st1, (void**) st2)
Sets st1, st2			
sq1 = sq2	sq1.equals(sq2)	sq1 == sq2	equalsSequence((void**) sq1, (void**) sq2)
Sequences sq1, sq2			

Table 22: Mappings from UML/OCL to Java 7, C and Python

With regard to code efficiency, it is more efficient to specify the addition of element x to a collection col in a postcondition by

$$x : col$$

rather than by the semantically equivalent

$$col = col@pre \rightarrow including(x)$$

because the second version makes a copy of *col*. Likewise with the operators / :, <: and / <:. Direct assignment $ax.br = brx$ of the value of a many-valued association end *br* can be inefficient, particularly if the opposite end is named, due to the code needed to maintain the consistency of the two association ends. Incremental changes, using :, <:, etc, to *br* are more efficient.

To speed up loading of models, automatic consistency checking and enforcement of model data can be omitted by commenting out the *Controller* operation *checkCompleteness* from *loadModel*. If this is done, the model should already be correct and consistent (inverse ends of bidirectional associations set correctly, and identity attributes should be set to different values for different objects).

C++ seems ill-suited to the complex manipulation of associations that occurs in some refactoring problems. The Java 6 implementation seems to be usually the fastest of the Java implementation choices, but does not guarantee any ordering for set-valued association ends, which may make it difficult to validate results. In addition, the Java 6 and Java 7 implementations use advanced features of Java which may not be available on some systems. Table 23 illustrates the comparative efficiency of the implementation versions on three large test cases: 1) composition of two associations $br : A \rightarrow Set(B)$ and $cr : B \rightarrow Set(C)$ into a relation $xr : A \rightarrow Set(C)$, with 50 *A* objects, 2500 *B* and 125000 *C*'s (a refinement transformation); 2) A Petri-Net to Statecharts transformation (migration and refactoring); 3) Class diagram restructuring, with two test cases, one of size 5500, another of size 50,000 (refactoring).

Test	Java 4	Java 6	C#	C++
Composition (127,550 elements)	245s	671ms	122s	117s
PN2SC (8000 elements) (40,000 elements)	290ms 3.8s	94ms 1.6s	198ms 5s	6s 161s
CD restructuring (5500 elements)	87s	2.3s	142s	13s
CD restructuring (50000 elements)	1063s	66s	1781s	–

Table 23: Efficiency comparisons

Table 24 shows the results of implementing a complex numerical computation (CDO) [3] in the three language versions. The generated code was considerably more efficient than hand-written C++.

Version	Execution time for first 20 $P(S = s)$ calls	Execution time for first 50 $P(S = s)$ calls
Java 4	32ms	93ms
C#	10ms	20ms
C++	62ms	100ms

Table 24: Execution times for CDO versions

E Read and write frames, definedness and determinacy

Table 25 (an updated version of the corresponding table of [12]) gives some cases of the definitions of these frames. $var(P)$ is the set of all features and entity type names used in *P*, likewise for $var^*(P)$.

$e2.r \rightarrow excludes(e1)$ for single-valued *r* is treated as $e2 \rightarrow excludesAll(e2 \rightarrow select(r = e1))$, and $e2.r \rightarrow excludesAll(e1)$ for single-valued *r* is treated as $e2 \rightarrow excludesAll(e2 \rightarrow select(r : e1))$, likewise for : and / :. $e1.f \rightarrow first() = e2$ has the same read and write frames as $e1.f \rightarrow last() = e2$.

In computing $wr(P)$ we also take account of the features and entity types which depend upon the explicitly updated features and entity types of *Cn*, such as inverse association ends.

If there is a constraint φ in the class diagram which implicitly defines a feature *g* in terms of feature *f*, ie: $f \in rd(\varphi)$ and $g \in wr(\varphi)$, then *g* depends on *f*. In particular, if an association end *role2* has a named opposite end *role1*, then *role1* depends on *role2* and vice-versa.

P	$rd(P)$	$wr(P)$	$rd^*(P)$	$wr^*(P)$
Basic expression e without quantifiers, logical operators or $=, :, E[],$ $\rightarrow includes,$ $\rightarrow includesAll,$ $\rightarrow excludesAll,$ $\rightarrow excludes,$ $\rightarrow isDeleted$	Set of features and entity type names used in P : $var(P)$	$\{\}$	Set of pairs (obj, f) of objects and features, $obj.f,$ in P , plus entity type names in P : $var^*(P)$	$\{\}$
$e1 : e2.r$ $e2.r \rightarrow includes(e1)$ r many-valued $e1, e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e2.r \rightarrow excludes(e1)$ $e1 / : e2.r$ r many-valued $e1, e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e1.f = e2$ $e1$ single-valued	$var(e1) \cup var(e2)$	$\{f\}$	$var^*(e1) \cup var^*(e2)$	$\{(e1, f)\}$
$e1.f[i] = e2$ $e1$ single-valued, f sequence-valued	$var(e1) \cup var(e2)$ $\cup var(i)$	$\{f\}$	$var^*(e1) \cup var^*(e2)$ $\cup var^*(i)$	$\{(e1, f)\}$
$e1.f \rightarrow last() = e2$ $e1.f \rightarrow first() = e2$ $e1$ single-valued f sequence-valued	$var(e1) \cup var(e2)$	$\{f\}$	$var^*(e1) \cup var^*(e2)$	$\{(e1, f)\}$
$e2.r \rightarrow includesAll(e1)$ $e1 <: e2.r$ $r, e1$ many-valued $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e2.r \rightarrow excludesAll(e1)$ $r, e1$ many-valued $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$E[e1]$	$var(e1) \cup \{E\}$	$\{\}$	$var^*(e1) \cup \{E\}$	$\{\}$
$E \rightarrow exists(x Q)$ (E concrete entity type)	$rd(Q)$	$wr(Q) \cup \{E\} \cup$ all superclasses of E	$rd^*(Q)$	$wr^*(Q) \cup \{E\} \cup$ all superclasses of E
$E \rightarrow exists1(x Q)$ (E concrete entity type)	$rd(Q)$	$wr(Q) \cup \{E\} \cup$ all superclasses of E	$rd^*(Q)$	$wr^*(Q) \cup \{E\} \cup$ all superclasses of E
$E \rightarrow forAll(x Q)$	$rd(Q) \cup \{E\}$	$wr(Q)$	$rd^*(Q) \cup \{E\}$	$wr^*(Q)$
$x \rightarrow isDeleted()$ x of element type entity type E	$var(x)$	$\{E\} \cup$ all superclasses of $E \cup$ E -valued roles r	$var^*(x)$	$\{E\} \cup$ all superclasses of $E \cup$ E -valued roles r
$C \Rightarrow Q$	$var(C) \cup rd(Q)$	$wr(Q)$	$var^*(C) \cup rd^*(Q)$	$wr^*(Q)$
$Q \& R$	$rd(Q) \cup rd(R)$	$wr(Q) \cup wr(R)$	$rd^*(Q) \cup rd^*(R)$	$wr^*(Q) \cup wr^*(R)$

Table 25: Definition of read and write frames

Creating an instance x of a concrete entity type E also adds x to each supertype F of E , and so these supertypes are also included in the write frames of $E \rightarrow exists(x \mid Q)$ and $E \rightarrow exists1(x \mid Q)$ in the above table.

Deleting an instance x of entity type E by $x \rightarrow isDeleted()$ may affect any supertype of E and any association end owned by E or its supertypes, and any association end incident with E or with any supertype of E . Additionally, if entity types E and F are related by an association which is a composition at the E end, or by an association with a mandatory multiplicity at the E end, i.e., a multiplicity with lower bound 1 or more, then deletion of E instances will affect F and its features and supertypes and incident associations, recursively.

In a constraint succedent, a read of a variable written by a preceding conjunct is not included in the read frame:

$P \Rightarrow r = e \ \& \ f = r$

Here, r counts as written but not read.

The read frame of an operation invocation $e.op(pars)$ is the read frame of e and of the $pars$ corresponding to the input parameters of op together with the read frame of the postcondition $Post_{op}$ of op , excluding the formal parameters v of op . Its write frame is that of the actual parameters corresponding to the outputs of op , and $wr(Post_{op}) - v$. $wr(G)$ of a set G of constraints is the union of the constraint write frames, likewise for $rd(G)$, $wr^*(G)$, $rd^*(G)$.

For each postcondition, precondition and invariant constraint Cn of a transformation, the definedness condition $def(Cn)$ is a necessary assumption which should hold before the constraint is applied or evaluated, in order that its evaluation is well-defined. Postcondition constraints should normally also satisfy the condition of determinacy. Examples of the clauses for the definedness function $def : Exp(L) \rightarrow Exp(L)$ are given in Table 26. It is also required that numeric expressions have values within the range of their computational numeric type.

Definedness of an operation call requires proof of termination of the call. Definedness of or , $\&$ requires definedness of both arguments because different implementation/evaluation strategies could be used in different formalisms or programming languages: it cannot be assumed that short-cut evaluation will be used (in ATL, for example, strict evaluation of logical operators is used [5]). Only in the case of implication is the left hand side used as a ‘guard’ to ensure the definedness of the succedent. We treat $A \Rightarrow B$ equivalently to OCL *if A then B else true endif*.

Examples of the clauses for the determinacy function $det : Exp(L) \rightarrow Exp(L)$ are given in Table 27.

Specifiers should also ensure that calls of update operations do not occur in mapping conditions or in other contexts where pure values are expected.

F Confluence checks

The tools use syntactic checking to check the confluence of type 0 or type 1 constraints (other forms of constraint need proof using variants, as described in Section 11). A constraint Cn on class E , of the form

$Ante \Rightarrow Succ$

is checked as follows:

1. *self* of type E is added to a set *iterated*, as is each implicitly universally quantified variable $c : C$ of class type occurring in *Ante*. Each distinct combination of bindings to these variables is iterated over exactly once by a bounded-loop implementation of the constraint. (Type 2 constraints may reapply the constraint to particular combinations more than once, and type 3 constraints may also introduce new combinations to be iterated over). Type 0 constraints do not have a context class or implicit *self* variable.
2. A list *created* of the existentially-quantified variables t in $T \rightarrow exists(t \mid pred)$ formulae in *Succ* is maintained. These are the objects which are newly created in each iteration (not looked-up and modified). t is only added to *created* if:
 - (a) T is a concrete entity type and not of pre-form,
 - (b) either T has no unique/primary key, or its key is assigned by a top-level equation $t.key = c.ckey$ to the primary key of the only element c of *iterated*.

These conditions ensure that t is genuinely new.

<i>Constraint expression e</i>	<i>Definedness condition def(e)</i>
<i>e.f</i> Data feature application	$def(e) \ \& \ E.allInstances() \rightarrow includes(e)$ where E is the declared classifier of e
Operation call $e.op(p)$	$def(e) \ \& \ E.allInstances() \rightarrow includes(e) \ \& \ def(p) \ \& \ def(e.Post_{op}(p))$ where E is the declared classifier of e , $Post_{op}$ the postcondition of op
a/b	$b \neq 0 \ \& \ def(a) \ \& \ def(b)$
$s \rightarrow at(ind)$ sequence, string s	$ind > 0 \ \& \ ind \leq s \rightarrow size() \ \& \ def(s) \ \& \ def(ind)$
$E[v]$ entity type E with identity attribute id , v single-valued	$E.id \rightarrow includes(v) \ \& \ def(v)$
$s \rightarrow last()$ $s \rightarrow first()$ $s \rightarrow max()$ $s \rightarrow min()$	$s \rightarrow size() > 0 \ \& \ def(s)$
$s \rightarrow any(P)$	$s \rightarrow exists(P) \ \& \ def(s) \ \& \ def(P)$
$v.sqrt$	$v \geq 0 \ \& \ def(v)$
$v.log, v.log10$	$v > 0 \ \& \ def(v)$
$v.asin, v.acos$	$(v.abs \leq 1) \ \& \ def(v)$
$v.pow(x)$	$(v < 0 \Rightarrow x : int) \ \& \ (v = 0 \Rightarrow x > 0) \ \& \ def(v) \ \& \ def(x)$
$A \ \& \ B$	$def(A) \ \& \ def(B)$
$A \ or \ B$	$def(A) \ \& \ def(B)$
$A \Rightarrow B$	$def(A) \ \& \ (A \Rightarrow def(B))$
$E \rightarrow exists(x \mid A)$	$def(E) \ \& \ E \rightarrow forAll(x \mid def(A))$
$E \rightarrow forAll(x \mid A)$	

Table 26: Definedness conditions for expressions

<i>Constraint expression e</i>	<i>Determinacy condition det(e)</i>
$s \rightarrow any(P)$	$s \rightarrow select(P) \rightarrow size() = 1 \ \& \ det(s) \ \& \ det(P)$
$s \rightarrow asSequence()$	$s \rightarrow size() \leq 1 \ \& \ det(s)$ for set-valued s
Case-conjunction $(E1 \Rightarrow P1) \ \& \ \dots \ \& \ (En \Rightarrow Pn)$	Conjunction of $not(Ei \ \& \ Ej)$ for $i \neq j$, and each $(det(Ei) \ \& \ (Ei \Rightarrow det(Pi)))$
$A \ \& \ B$	$det(A) \ \& \ det(B)$
$A \Rightarrow B$	$det(A) \ \& \ (A \Rightarrow det(B))$
$E \rightarrow exists(x \mid A)$	$det(E) \ \& \ E \rightarrow forAll(x \mid det(A))$
$E \rightarrow exists1(x \mid A)$	
$E \rightarrow forAll(x \mid A)$	$det(E) \ \& \ E \rightarrow forAll(x \mid det(A))$ Additionally, order-independence of A for $x : E$.

Table 27: Determinacy conditions for expressions

3. Assignments $t.f = e$ for a direct feature f of $t \in created$ are confluent, if $rd(e)$ is disjoint from $wr(Cn)$, as are assignments $self.f = val$ or $c.f = val$ for a value val (with no variables, including $self$) and $c \in iterated$. Assignments $c.f = c.g$ for direct features of $c \in iterated$ are also confluent.
4. Assignments $T_j[sId].g = e$ are confluent if sId is the primary key of the only *iterated* entity type S_j , and if there is a preceding confluent constraint which maps S_j to T_j (so that there is a 1-to-0..1 relation between these entity types based on the primary key values).
5. A formula $t : e.r$ is confluent if $t \in created$, and r is an unordered role. $e : t.r$ and $e <: t.r$ are confluent if $t \in created$ and r is unordered, or if e is an *iterated* variable which has an ordered iteration range (not an entity type). $rd(e)$ must also be disjoint from $wr(Cn)$ in both cases.
6. $e1 \rightarrow includes(e2)$ is treated as for $e2 : e1$, and $e1 \rightarrow includesAll(e2)$ is treated as for $e2 <: e1$.
7. In the case of bidirectional associations, the explicit update to one end, and the corresponding implicit update to the other end must both be confluent.
8. A conjunction $e1 \& e2$ is confluent if $e1$ and $e2$ are.
9. $e1 \Rightarrow e2$ is confluent if $e2$ is.
10. $T \rightarrow exists(t \mid pred)$ is confluent if $pred$ is confluent under the addition of t to *created*, if it is eligible to be added.
11. $T \rightarrow forAll(t \mid pred)$ is confluent if $pred$ is confluent under the addition of t to *iterated*.
12. $e \rightarrow display()$ is confluent if there is at most one variable in e , from *iterated*, and with an ordered iteration range.

These rules ensure that data written on previous iterations of Cn cannot affect the execution of the current application of Cn . Set-valued collections, such as the set $E.allInstances$ of objects of a class E , should not be used directly to produce a sequence-valued collection, because the order of the latter will be arbitrary. The sorting operators can instead be used to make the order determinate.

As an example, the classic UML to RDB transformation is confluent if written as:

```
UMLClass::
  Table->exists( t | t.name = name &
    attributes->forAll( a |
      Column->exists( c | c.name = a.name & c : t.columns & c.typeName = a.typeName ) ) )
```

If $name$ is a primary key for *UMLClass* and *Table*, but not for *Column*, and *attributes* and *columns* are unordered, then t and c are added to *created*, and all updates are confluent.

G OCL expression semantics

This section defines the mathematical interpretation of OCL in first-order logic and set theory, and serves as the primary definition of the semantics of our restricted version of OCL (without *null* or *invalid* elements). The data types and operators of B AMN will be used, with some adaptations for the types.

The interpretation of types is shown in Table 28.

OCL type e	Semantics e' in B AMN
int	<i>INT</i> (32-bit signed integers)
long	<i>LONG</i> (64-bit signed integers)
double	Appropriate representation, eg., IEEE 754 double-precision floating-point
String	seq(<i>INT</i>)
class E	domain <i>F_OBJ</i> of references

Table 28: Semantic mapping for types

For primitive literal expressions e – numbers, strings, Booleans, and elements of enumerations, the semantic denotation e' of e directly corresponds to e (Table 29). Although B only directly represents integers, computational real numbers can be represented as rational numbers or by pairs of mantissas and exponents.

<i>OCL Term e</i>	<i>Semantics e' in B AMN</i>
number n	n
<i>true</i>	<i>TRUE</i>
<i>false</i>	<i>FALSE</i>
String “t”	Sequence denoted “t” consisting of characters of t in left to right order.
<i>val</i> from enumeration T	Representation of $T :: val$

Table 29: Semantic mapping for primitive literals

If v and w are two distinct enumeration literals (of the same or different enumerations) then their semantic denotations satisfy $v' \neq w'$. If v and w are syntactically the same, but belong to two distinct enumerations, then $v' \neq w'$. Otherwise $v' = w'$.

An entity type E is represented by its current extent, the set es of existing objects of type E . These range over a set E_OBJ of unspecified atomic object references, for root classes E . For subclasses F of E , the axiom $fs \subseteq es$ holds.

Attributes $att : T$ owned by entity type E are represented as total functions $att_E : es \rightarrow T'$.

We will consider two kinds of collections (sets and sequences), and restrict these to consist only of elements of a single type, as in [19]. This type can either be a numeric type, the Boolean type, the string type, a particular enumeration, or a particular class. Apart from elements of subclasses of a common superclass, mixtures of elements of different types are not allowed. Ordered sets can be considered as a subtype $iseq(T)$ of the sequence type $seq(T)$.

Collection literal expressions have a direct interpretation: a set literal $Set\{e_1, \dots, e_n\}$ is interpreted by the mathematical set $\{e'_1, \dots, e'_n\}$. This has type $\mathbb{F}(T)$ where T is the semantic representation of the common type of the elements of the set. A sequence literal $Sequence\{e_1, \dots, e_n\}$ is interpreted by the mathematical sequence s of length n , which has $s(1) = e'_1, \dots, s(n) = e'_n$. This is also written in B as $[e'_1, \dots, e'_n]$. The typing of s is a sequence type $seq(T)$, ie., $1..n \rightarrow T$ where T is the semantic representation of the common type of the elements.

One-multiplicity (single-valued) association ends r owned by entity E , and of type F are represented as total functions $r_E : es \rightarrow fs$.

Many-valued association ends r owned by entity E and of element type F are represented as $r_E : es \rightarrow \mathbb{F}(fs)$ if unordered, and as $r_E : es \rightarrow seq(fs)$ if ordered. Multiplicity bounds on r can be expressed in terms of $card(r_E(ex))$ for $ex \in es$. Axioms expressing the mutual inverse properties of opposite association ends can also be expressed.

Navigation expressions are formalised following the rules of Table 30.

<i>OCL Term e1</i>	<i>OCL Term e2</i>	<i>Condition</i>	<i>Semantics of e1.e2</i>
<i>obj</i>	f	Feature f of object <i>obj</i>	$f'(obj')$
<i>objs</i>	f	1-valued feature f of set <i>objs</i>	$f'[objs']$
<i>objsq</i>	f	1-valued feature f of sequence <i>objsq</i>	$objsq'; f'$
<i>objs</i>	r	set <i>objs</i> , set-valued r	$union(r'[objs'])$
<i>objs</i>	r	set <i>objs</i> , sequence-valued r	$ran(union(r'[objs']))$
<i>objsq</i>	r	sequence <i>objsq</i> , set-valued r	$union(r'[ran(objsq')])$
<i>objsq</i>	r	sequence <i>objsq</i> , sequence-valued r	$conc(objsq'; r')$

Table 30: Semantic mapping for navigation expressions

Numeric operators such as $*$, $+$, $/$, $-$ are represented as corresponding function symbols of arity 2 on INT. The definitions of [19] are used, likewise for *abs*, *floor*, $>$, $<$, \leq , \geq , *div* and *mod*. The logical operators are interpreted by the corresponding semantic operators: $\&$ by \wedge , *or* by \vee , *not* by \neg and \Rightarrow by \Rightarrow .

size, $=$ and $+$ (*concat*) are defined on strings as in [19]. Equality of strings means that they have the same characters in the same order (as in sequence equality). The Boolean operators are defined according to the usual 2-valued truth tables on BOOL.

On collections the operators *includes*, *excludes*, *includesAll*, *excludesAll* and *size* are given the usual definitions of membership, non-membership, subset, disjointness and cardinality. Table 31 shows some

examples of interpretations of collection operators.

<i>OCLE Term e</i>	<i>Condition</i>	<i>Semantics e'</i>
$s \rightarrow \text{size}()$	set or sequence s	cardinality $\text{card}(s')$
$s \rightarrow \text{includes}(x)$	set s	$x' \in s'$
$s \rightarrow \text{excludes}(x)$	set s	$x' \notin s'$
$s \rightarrow \text{includes}(x)$	sequence s	$x' \in \text{ran}(s')$
$s \rightarrow \text{excludes}(x)$	sequence s	$x' \notin \text{ran}(s')$
$s \rightarrow \text{asSet}()$	s set	s'
$s \rightarrow \text{asSet}()$	s sequence	$\text{ran}(s')$
$s \rightarrow \text{includesAll}(t)$	sets s and t	$t' \subseteq s'$
$s \rightarrow \text{includesAll}(t)$	sequences s and t	$\text{ran}(t') \subseteq \text{ran}(s')$
$s \rightarrow \text{excludesAll}(t)$	sets s and t	$s' \cap t' = \{\}$
$s \rightarrow \text{excludesAll}(t)$	sequences s and t	$\text{ran}(s') \cap \text{ran}(t') = \{\}$
$s \rightarrow \text{sum}()$	set s	sum of elements of s'
$s \rightarrow \text{sum}()$	sequence s , $\text{card}(s') = n$	$s'(1) + \dots + s'(n)$

Table 31: Semantic mapping for collection operations

Properties such as

$$s \rightarrow \text{includes}(x) \equiv s \rightarrow \text{asSet}() \rightarrow \text{includes}(x)$$

follow directly from this semantics. Notice that $\rightarrow \text{includes}$ and $\rightarrow \text{excludes}$ can be applied on both sets and sequences – sequences are mapped to sets before the operators are evaluated.

The operators *union* and *intersection* are defined in terms of their mathematical counterparts (Table 32). *union* on sequences is defined to be the same as concatenation, as in [19]. Again, we can give a

<i>OCLE Term e</i>	<i>Condition</i>	<i>Semantics e'</i>
$s \rightarrow \text{union}(t)$	s and t sets	$s' \cup t'$
$s \rightarrow \text{union}(t)$	s and t sequences	$s' \hat{\ } t'$
$s \rightarrow \text{intersection}(t)$	s and t sets	$s' \cap t'$
$s \rightarrow \text{intersection}(t)$	s and t sequences	$\text{contract}(\{i \mapsto x \mid (i \mapsto x) \in s' \wedge x \in \text{ran}(t')\})$
$s \rightarrow \text{excluding}(t)$	s a set	$s' - \{t'\}$
$s \rightarrow \text{including}(t)$	s a set	$s' \cup \{t'\}$

Table 32: Semantic mapping for collection construction operations

semantics for operators applied to mixes of sets and sequences by converting where necessary.

Sequence-specific operations are defined in Table 33. *including* for a sequence is defined as *append*.

<i>OCLE Term e</i>	<i>Condition</i>	<i>Semantics e'</i>
$s = t$	s and t sequences	$s' = t'$ as maps
$s \rightarrow \text{first}()$	non-empty sequence s	$s'(1)$
$s \rightarrow \text{last}()$	non-empty sequence s	$s'(\text{card}(s'))$
$s \rightarrow \text{front}()$	non-empty sequence or string s	subsequence $[s'(1), \dots, s'(\text{card}(s') - 1)]$ of s'
$s \rightarrow \text{tail}()$	non-empty sequence or string s	subsequence $[s'(2), \dots, s'(\text{card}(s'))]$ of s'
$s \rightarrow \text{sort}()$	sequence s	reordering of s' such that elements are in non-descending $<$ order
$s \rightarrow \text{reverse}()$	sequence s , $n = \text{card}(s')$	$\{i \mapsto s'(n - i + 1) \mid i \in \text{dom}(s')\}$
$s \rightarrow \text{append}(x)$	sequence s	$s' \hat{\ } [x']$
$s \rightarrow \text{prepend}(x)$	sequence s	$[x'] \hat{\ } s'$
$s \rightarrow \text{subSequence}(i, j)$	sequence s	subsequence $[s'(i), \dots, s'(j)]$ of s'

Table 33: Semantic mapping for sequence operations

$s \rightarrow \text{excluding}(x)$ for sequence s is defined as $s \rightarrow \text{select}(self \neq x)$.

Select expressions evaluate to sets or sequences depending on the collection they operate over (Table 34). Their first argument must denote a finite set or sequence. In the clauses for *select*, x is a new variable, not occurring in P' .

<i>OCL Term e</i>	<i>Condition</i>	<i>Semantics e'</i>
$objs \rightarrow select(P)$	set <i>objs</i>	$\{x \mid x \in objs' \wedge x.P'\}$
$objs \rightarrow select(P)$	sequence <i>objs</i>	$contract(\{i \mapsto x \mid (i \mapsto x) \in objs' \wedge x.P'\})$
$objs \rightarrow collect(e)$	set <i>objs</i>	bag $\{x.e' \mid x \in objs'\}$ as a sequence
$objs \rightarrow collect(e)$	sequence <i>objs</i>	$\{i \mapsto objs'(i).e' \mid i \in dom(objs')\}$
$s \rightarrow unionAll(e)$	<i>s, e</i> set-valued	$\bigcup \{x.e' \mid x \in s'\}$
$s \rightarrow unionAll(e)$	<i>s, e</i> sequence-valued	$conc(\{i \mapsto x.e' \mid (i \mapsto x) \in s'\})$
$s \rightarrow intersectAll(e)$	<i>s, e</i> set-valued	$\bigcap \{x.e' \mid x \in s'\}$
$s \rightarrow intersectAll(e)$	<i>s, e</i> sequence-valued	$\bigcap \{ran(x.e') \mid (i \mapsto x) \in s'\}$
$s \rightarrow flatten()$	<i>s</i> set-valued	$\bigcup \{x \mid x \in s'\}$
$s \rightarrow flatten()$	<i>s</i> sequence-valued	$conc(s')$

Table 34: Semantic mapping for select expressions

$contract(m)$ turns a map $m : 1..n \mapsto T$ into a sequence sq by removing gaps in the index set, maintaining the same order of elements. For example $contract(\{2 \mapsto a, 3 \mapsto b, 7 \mapsto c\})$ is $[a, b, c]$.

The notation $a.P$ denotes P with a prefixing each feature of the class of a which has no prefix in P , or has *self* prefix. For example $a.(att > 10)$ is $att(a) > 10$. $a.self$ is a .

UML-RSDS activities are semantically interpreted as B AMN generalised substitutions.

H Alternative Z3 mapping

The mapping from UML and OCL to Z3 described in Section 12 becomes complex for representing sets and set operators. An alternative is to represent sets as arrays, as in the mapping to SMV. A *-valued association end $br : Set(B)$ can be modelled in Z3 as a B-indexed array of Bool elements:

```
(define-sort Set (T) (Array T Bool))
(declare-fun br (A) (Set B))
```

Using this interpretation, the mapping of UML-RSDS OCL expressions into Z3 is given by Table 35. The

<i>OCL e</i>	<i>Z3 interpretation e'</i>
$bx : ax.br$	$(select (br ax') bx') = true$
$bs1 \cap bs2$	$((_ map and) bs1' bs2')$
$bs1 \cup bs2$	$((_ map or) bs1' bs2')$
$bs1 - bs2$	$((_ map and) bs1' ((_ map not) bs2'))$
$Set\{\}$	$((as const (Set B)) false)$
$bs \rightarrow including(bx)$	$(store bs' bx' true)$
$bs \rightarrow excluding(bx)$	$(store bs' bx' false)$

Table 35: Mapping to Z3 using Arrays

advantage of this mapping is that operators such as $\rightarrow select$ and $\rightarrow reject$ can be directly defined. For example, $bs \rightarrow select(bx \mid P)$ can be expressed by defining a Bool-valued function

```
(define-fun pf ((bx B)) Bool P')
```

where P' is the interpretation of P , and the select expression is then $((_ map and) bs' (_ as-array pf))$.

A collect expression $bs \rightarrow collect(bx \mid e)$ where e has values in type T can be expressed by defining a Bool-valued function

```
(define-fun collect_i ((tx T)) Bool
  (exists ((bx B))
    (and (= (select bs bx) true) (= e tx))))
```

and the array of this is then $(_ as-array collect_i)$. However, the mapping does not model sequences.

I Format of mm.txt file

This file holds the data of a UML-RSDS system. For classes, the format is:

```
Entity:
name x y
superclass * stereotype1 ...
att1 type1 3 isfinal isidentity isstatic att2 type2 3 isfinal isidentity isstatic ...
```

where the name and type of each attribute are listed, together with booleans true/false indicating if it is final, an identity or static. The number '3' indicates an internal (ordinary) attribute, as opposed to a sensor (1) or actuator (5). Superclass is the name of the superclass if this exists, or null if the class is a root class.

For associations it is:

```
Association:
entity1 entity2 mult1 startx starty endx endy mult2 role2 role1
waypoints
```

The code for * multiplicity is 0, the code for 0..1 multiplicity is -1, the code for 1 multiplicity is 1. Waypoints (if any) are printed as pairs xcoord ycoord.

For generalisations there is the representation:

```
Generalisation:
superclass subclass startx starty endx endy
waypoints
```

For operations, the format is:

```
Operation:
name
entityname
resulttype
parameter1 type1 parameter2 type2 ...
stereotypes
precondition
postcondition
```

The stereotypes can include query, abstract, cached, static, etc.

For general use cases the format is:

```
GeneralUseCase:
name p1 t1 ... pn tn
ext1 ... extm
incl ... incq
att1 at1 ... attr atr
```

where p1 : t1, ..., pn : tn are the parameters and parameter types of the use case, ext1 to extm the names of extension use cases, and incl to incq the names of included use cases. Attribute names and types are listed in the last line.

For constraints the format is:

```
Constraint:
cond0
cond1
succedent
ownerEntity ownerUsecase
ordered
```

In this representation, cond0 is a first conjunct of the antecedent, and cond1 is the remainder of the antecedent. The boolean ordered indicates if the constraint should be iterated over the ownerEntity in a particular order. Cond0 is null if not present, cond1 has default value true. The ownerEntity is null if the constraint is of type 0, without a particular entity context.

J Transformation synthesis from metamodels

On the *Synthesis* menu, a set of options are provided to synthesise transformations from pairs MM_1 and MM_2 of source and target metamodels. The metamodels should be defined using UML-RSDS syntax in text files (eg., *mm1.txt* and *mm2.txt*) in the *output* directory. Source entities should have the stereotype *source*, target entities should have stereotype *target*. An overall metamodel file *mm.txt* can import both of these:

```
Import:  
mm1.txt
```

```
Import:  
mm2.txt
```

Source classes and target classes must have different names. Classes which are neither source or target are considered shared between the metamodels.

Load this model using *Recent* as usual, and select one of the transformation synthesis menu items. These are:

<i>Measure</i>	<i>Definition</i>
<i>Name syntactic similarity (NSS)</i>	Classes have names with low string edit distances
<i>Name semantic similarity (NMS)</i>	Class names are synonymous terms or in the same/linked term families according to a thesaurus
<i>Data structure similarity (DSS)</i>	Classes possess similar data in their owned, inherited or composed features
<i>Graph structural similarity (GSS)</i>	Class neighbourhoods in the 2 metamodels have similar graph structure metrics
<i>Graph edit similarity (GES)</i>	Class reachability graphs in the 2 metamodels have low graph edit distance
<i>Semantic context similarity (SCS)</i>	Classes play similar semantic roles in the 2 metamodels.

In each case, the system tries to infer a correspondence m of source classes to target classes. For medium/large metamodels, start with NSS or NMS to infer plausible matchings based on names, then add the matchings you wish to enforce to *mm.txt* via the lines

```
EntityMapping:  
Src |-> Trg
```

More computationally intensive matching approaches, such as DSS and SCS can then be used to infer other matchings based on these.

Following automated matching of classes, source and target features are also matched. The process is interactive, with the developer asked to confirm proposed matchings if there are choices.

The end result is a set of high-level transformation rules, and transformation code in QVT-R, QVT-O, UML-RSDS and ATL. Two files are produced in *output*: *forward.txt*, which holds the code for the source to target direction, and *reverse.txt*, holding the target to source code.