Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 131–149
www.elsevier.com/locate/entcs

Refinement Patterns for UML

K. Lano[1]

Dept. of Computer Science
King's College London
London, UK

K. Androutsopolous[2]

Dept. of Computer Science
King's College London
London, UK

D. Clark[3]

Dept. of Computer Science
King's College London
London, UK

Abstract

This paper describes strategies or 'patterns' for the refinement of UML specifications into executable implementations, using a semantically precise subset, UML-RSDS, of UML.

Keywords: UML-RSDS, Refinement, Model Transformation

1 Introduction

UML is a widely used notation for object-oriented specification and design, and it is also an international standard. Together with the Object Constraint Language (OCL), it represents a fusion of formal and graphical specification

[1] Email: kcl@dcs.kcl.ac.uk
[2] Email: kelly@dcs.kcl.ac.uk
[3] Email: david@dcs.kcl.ac.uk

1571-0661
doi:10.1016/j.entcs.2005.04.028

languages which has high potential for introducing the benefits of formal specification techniques into mainstream software development. In this paper a subset, UML-RSDS [10], will be used as a language for precise specification in UML, and to illustrate how systematic rules for refinement of UML specifications into executable code can be defined.

Figure 1 shows the overall development process supported by UML-RSDS and its accompanying toolset. A developer can construct analysis or design class diagrams and state machines using the tool, analyse these for conformance to the UML or platform-specific metamodel (currently only Java and Java-based web systems are supported), transform models to improve their quality or refine them, translate to B [7] or SMV [1] for semantic analysis, and generate Java code from a Java UML model. A specific tool for generating web systems (using Servlets and JDBC in an MVC architecture) from class diagrams is also provided.

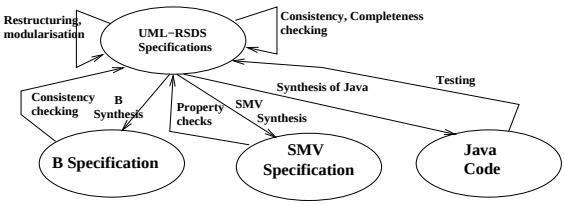

Fig. 1. UML-RSDS Process

2 Specification in UML

UML specifications can consist of a number of different complementary models, such as Use-Case models, Class Diagrams, Sequence Diagrams, etc. We consider only class diagrams, statecharts and constraints here.

Figure 2 shows a typical class diagram, of a Scrabble game system, consisting of classes with attributes and operations, associations, inheritance, and constraints.

Constraints can be placed on a number of UML elements:

- Classes, as invariants of a class, giving properties relating different features of the class.

  Eg., constraint $endx = startx \ or \ endy = starty$ on $Word$ means that a word object must be either vertical (first case) or horizontal (the second).

- Operations, as pre and postconditions of operations, defining the behaviour of the operation.

  Eg., the constraint $pre: \ score = 0$ on $setSymbol(c : char)$ expresses the fact that only the blank letter, with score zero, can be used as any character.

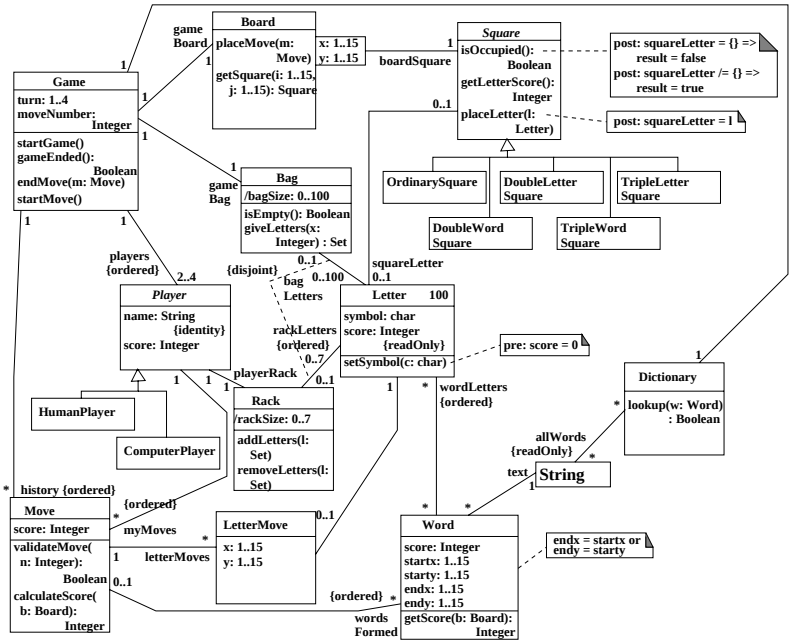

Fig. 2. Scrabble class diagram

- Associations, giving properties which describe which pairs of objects from classes at the ends of the association can be connected by the association.

  The constraint

  $$wordsFormed.text <: allWords$$

  on the associations *Game_Move* and *Game_Dictionary* expresses that all words formed in accepted moves of the Scrabble game must be in the dictionary.

2.1 UML-RSDS Constraints

Either the LOCA (logic of objects, constraints and associations) language [10] or OCL (http://www.omg.org/ocl) can be used to define constraints in a UML-RSDS model. The significant extension of UML-RSDS over standard UML class diagrams however is that constraints may be attached to associations:

> Constraints attached to sets of associations have as their context the set of all object pairs linked by these associations. This means that in many formulas quantifiers or reference to specific objects can be avoided completely.

For example the constraint $C1$:

$$aatt = On \ \rightarrow \ batt = On$$

of Figure 3 has the semantics

$$\forall a : A; b : B \ \cdot \ (a, b) \in A\_B \ \wedge \ a.aatt = On \ \rightarrow \ b.batt = On$$

where A_B is the extent of the association between A and B.

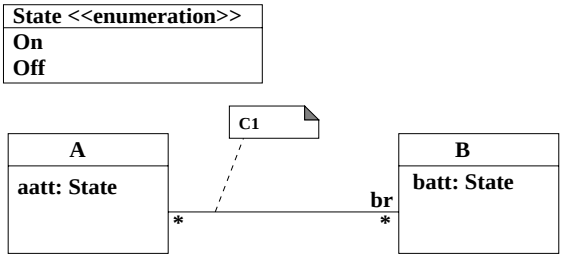

Fig. 3. Simple class diagram

Association constraints permit a more abstract and less implementation biased specification of properties than OCL navigation expressions: which always express a property starting from the context of a particular class, so biasing the specification towards implementations in which that class is responsible for maintaining the constraint.

Also in contrast to the navigation expressions used in OCL, association constraints are more resilient to changes in the structure of the model: the formula can often remain unchanged, only the set of associations it is linked to need to change.

LOCA is a simplified subset of OCL designed to be easier to teach and use, and avoids the use of mathematical constructs such as quantifiers, and also simplifies OCL syntax. Metamodel features of OCL are omitted from LOCA: $oclIsTypeOf$, $oclIsKindOf$, $oclIsNew$, $oclAsType$, $allInstances$, $OclType$, $oclInState$, $OclState$, $OclAny$, $OclExpression$.

$x.oclIsKindOf(t)$ is expressible in LOCA as $x : t$ for class names t. $x.oclInState(s)$ is expressible by $x.att = s$ where s is a state of the state machine attached to the class, and att is an attribute which identifies the current state. $C.allInstances()$ is expressed by the name C by itself.

The procedural operator *iterate* of OCL is also omitted from LOCA, as are the bag and ordered set types.

Table 1 shows the syntax of LOCA expressions currently accepted in UML-RSDS constraints, within the UML-RSDS tools.

A *valueseq* is a comma-separated sequence of values. A factor level operator *op*1 can be:

$< value >$	::=	$< ident > \mid < number > \mid$
		$< string > \mid < boolean >$
$< objectref >$	::=	$< ident > \mid$
		$< objectref >.< ident > \mid$
		$< objectref >$ `!(` $< expression >$ `)`
$< arrayref >$	::=	$< objectref > \mid$
		$< objectref >$`[`$< value >$`]`
$< factor >$	::=	$< value > \mid \{ < valueseq > \} \mid$
		`Sequence{` $< valueseq >$ `}` $\mid < arrayref > \mid$
		$< factor >$ op1 $< factor >$
$< expression1 >$	::=	$< factor >$ op2 $< factor >$
$< expression >$	::=	$< expression1 > \mid$
		`(` $< expression >$ `)` $\mid$
		$< expression1 >$ op3 $< expression >$
$< staticinvariant >$	::=	$< expression > \mid$
		$< expression >$ `=>` $< expression >$
$< temporalinvariant >$	::=	$< temporalop >^{+} < expression > \mid$
		$< expression >$ `=>` $< temporalop >^{+} < expression >$

Table 1
LOCA Syntax

(i) $+$, $-$, $*$, $/$, div, mod

(ii) $\backslash/$, $/\backslash$ (also written as $\cup$ and $\cap$), $\uparrow$

A comparator operator $op2$ is one of $=$, $/=$, $<$, $>$, $<=$, $>=$, $:$, $<:$, $/:$, $/ <:$. A logical operator $op3$ is one of &, or. A temporal operator is one of AX (in all next states), EX (in some next state), AF (in some future state on all paths), EF (in some future state in some path), AG (in all future states on all paths) and EG (in all future states on some path). Identifiers are either class names, function names, class features (attribute, operation or role names), elements of enumerated types, or represent variables or constants (if in upper case). Variables are implicitly universally quantified over the entire formula. Operations can also be written with parameters as $op(p_1, ..., p_n)$, etc.

The functions currently supported in the UML-RSDS tool are $size$, $toUpper$, $toLower$ on strings and $size$, $asSet$, max, min, sum, prd on collections (sets and sequences), rev, $sort$ on sequences, and sqr, sqr, $floor$, $round$, abs on numbers. Extension to other functions of OCL is planned.

2.2 Derivation of Operational Constraints

The recommended approach for UML-RSDS development is to specify a system using *declarative* constraints only: constraints which do not refer to op-

eration names. Declarative constraints define the invariant properties of the system without any bias towards particular algorithms for maintaining them, so they form a *computation-independent model* (CIM) in MDA terminology. From such models a more explicit operational model can be systematically derived, as described in this section.

A declarative UML-RSDS specification consists of a UML class diagram, protocol state machines defining the intended life histories of classes in the class diagram, and declarative constraints on the class diagram classes and associations.

Declarative constraints, such as $aatt = On \ \rightarrow \ batt = On$ in Figure 3, can be interpreted as describing the reaction the system must perform in response to an event which makes the antecedent of the constraint true. In this case, if an event $setaatt(val)$ occurs on some A object ax, setting $ax.aatt$ to val, and $val = On$, then the constraint $(C1)$ will require that $setbatt(On)$ is performed on all B objects related to ax.

Thus the operational form of the constraint is

$$setaatt(val) \ \& \ val = On \ \rightarrow \ AX(batt = On)$$

where $AX(P)$ asserts that P holds in the 'next' state, ie, the state at termination of the reaction to the event. This has the same meaning as an OCL postcondition constraint.

In general, from a declarative constraint I on associations rs:

$$P \ \& \ G \ \rightarrow \ Q$$

we can deduce, for each event α that may affect the truth of I:

$$\alpha \ \& \ P1 \ \rightarrow \ AX(G[e/v] \ \rightarrow \ Q[e/v])$$

where v are the free variables of I, and α establishes $P[e/v]$ at its termination, for some expression e, when $P1$ is true (provided α does not modify any of the rs):

$$\alpha \ \& \ P1 \ \rightarrow \ AX(P[e/v])$$

$E[e/v]$ denotes the substitution of expression(s) e for identifier(s) v in E.

For each event α, of the forms $setf$ for a feature f, or $addr$, $remover$ for a many-valued role r, the UML-RSDS tool determines the set of constraints affected by α, ie, those constraints whose antecedent may be made true by α. These are collected together and the updates derived from them are used to define the system response to α: this response is that required in order to maintain the truth of the affected invariants. This algorithm is the basis of the Java and B synthesis processes, and could also be applied to generate code in other languages, such as C++ or C$^{\#}$.

For example, if a class invariant had the form

$$att = v \ \& \ att1 = v1 \ \rightarrow \ att2 = v2$$

for attributes att, $att1$ and $att2$ of the class, then the operational form for $setatt$ is:

$$setatt(x) \ \& \ x = v \ \rightarrow \ AX(att1 = v1 \ \rightarrow \ att2 = v2)$$

The implication inside the AX is used to synthesise code for the $setatt$ operation, eg:

```
public void setatt(T x)
{ att = x;
  if (x == v)
  { if (att1 == v1)
    { setatt2(v2); }
  }
}
```

3 Refinement Transformations on UML Models

UML models can be systematically transformed to refine them to forms closer to implementation on specific platforms. For example, the elimination of association classes (which are not expressible in any mainsteam OO programming language), the elimination of many-many associations (for refinement to a relational data model), etc. Such transformations can be viewed as patterns.

A useful refinement transformation, which reduces the complexity of data in a program, is:

Name *Make Association into Index*

Description This replaces an association i that identifies a member of a role set, by an integer index.

Motivation The resulting data structures are simpler and more efficient to implement.

Conditions The relationship

$$br.size > 0 \ \rightarrow \ i = br[index]$$

holds between the original and refined model.

Another example of a refinement transformation is the introduction of foreign keys to represent an explicit many-one association between two persistent entities.

Name *Replace Associations by Foreign Keys*

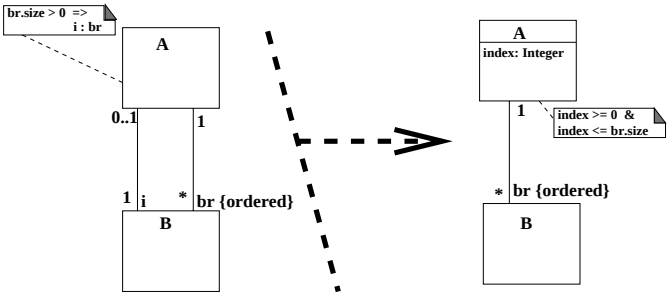

Fig. 4. Transforming index association into an attribute

Description This transformation applies to any explicit many-one association between persistent classes. It assumes that primary keys already exist for the classes linked by the association. It replaces the association by embedding values of the key of the entity at the 'one' end of the association into the entity at the 'many' end.

Motivation This is an essential step for implementation of a data model in a relational database. In particular, it can be used to implement the data repository of a web application in such a database.

Diagram This is shown in Figure 5.

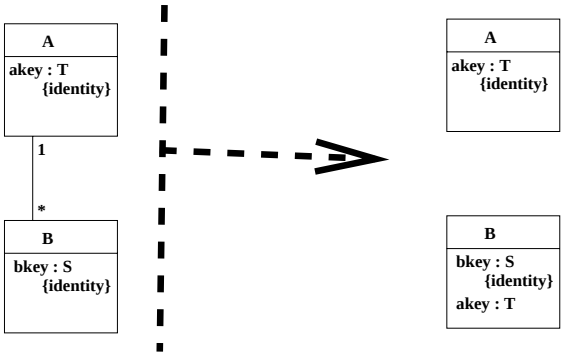

Fig. 5. Replacing Association by Foreign Key

Conditions $b.akey$ is equal to $a.akey$ exactly when $a \mapsto b$ in the original association:

$$(a, b) \in A\_B \iff b.akey = a.akey$$

This correspondence must be maintained by implementing *addbr* and *removebr* operations in terms of the foreign key values.

To apply this transformation, the user of the UML-RSDS tool selects the many-one association from a list of those in the current model, and a new

model with the foreign key is produced, replacing the original model. The new attribute is labelled (stereotyped) as a foreign key, so that correct SQL can be generated for lookups and modifications on the association, in a generated web system [12].

The usual refinement calculus transformations on pre/post specifications of operations are valid [13]:

Name *Weakening preconditions or strengthening postconditions*

Description An operation precondition can be weakened (so the operation can be applied in more situations) and/or its postcondition strengthened.

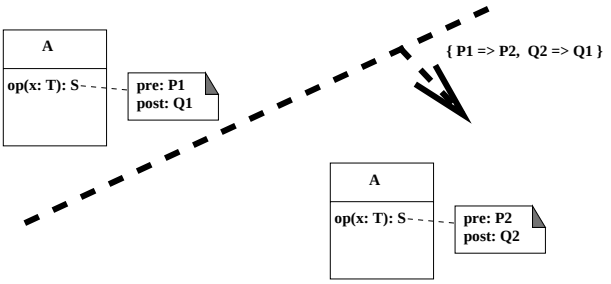

Fig. 6. Weakening preconditions/strengthening postconditions

4 Refinement Patterns for Constraint Implementation

More fine-grain refinement transformations can be carried out by considering the constraints of a model.

The following definitions will be used to define strategies for refining constraints to executable code.

Definition: Write frame

$wr(Code)$ denotes the set of object features that can be modified by $Code$. For example, $obj.setf(x)$ has write frame $\{obj.f\}$ and $E.setAllf(objs, x)$ has write frame the set of $obj.f$ for $obj \in objs$.

The write frame also takes into consideration indirect effects of updates. If there is a further feature g of the same class, which depends (for each object) on the value of f, then these write frames are $\{obj.f, obj.g\}$ and $\{obj : objs|obj.f\} \cup \{obj : objs|obj.g\}$.

Definition: $free(E)$

For an expression E, $free(E)$ denotes the set of object features referred to in E. Thus $free(obj.f = 1 + obj.g)$ is $\{obj.f, obj.g\}$.

Definition: Query form

Each LOCA expression e is given an interpretation e' in Java, based on the variables ranging over each set of objects in a particular context. For entity E, let $vare$ be the variable ranging over the set of objects of E. Then we have the interpretation given in Table 2.

LOCA	*Java*
Variable, constant, string or primitive value x	x
Attribute att of entity E	$vare.getatt()$
Role $role$ of entity E	$vare.getrole()$
$obj.f$ where obj is a single object	$obj'.getf()$
$objs.f$ where $objs$ is a set of objects of type E.	$E.getAllf(objs')$
$x : y$	$y'.contains(x')$
$x \ / : y$	$!(y'.contains(x'))$
$x = y$ for primitive x, y	$x' == y'$
$x = y$ for objects x, y	$x'.equals(y')$
$x \ div \ y$	$x' \ / \ y'$
$x \ mod \ y$	$x' \ \% \ y'$
$x \cup y$	$SystemTypes.Set.union(x', y')$
$x - y$ for sets x, y	$SystemTypes.Set.subtract(x', y')$
$\{x_1, \ldots, x_n\}$	$(new \ SystemTypes.Set()).add(x'_1).$ $\ldots.add(x'_n).getElements()$
$x \cap y$	$SystemTypes.Set.intersection(x', y')$
$P \ \& \ Q$	$P' \ \&\& \ Q'$
$P \ or \ Q$	$P' \ \|\| \ Q'$
$e[i]$	$e'.get(i' - 1)$

Table 2
Java query form of LOCA expressions

Definition: Update form

Given certain forms of postcondition/constraint conclusions X, a corresponding piece of program code $Code_X$ can be defined, which ensures that X holds after $Code_X$ is executed: $[Code_X]X$.

Table 3 shows the basic cases. In the table, an update form only exists

LOCA	*Java*
$x : obj.role$	if $(obj'.getrole().contains(x'))$ {} else $obj'.addrole(x')$;
$role$ set-valued, single object obj.	
Multiple objects obj of E	$E.addAllrole(obj', x')$;
x /$: obj.role$ $role$ many-valued	$obj'.removerole(x')$; obj single-valued $E.removeAllrole(obj', x')$; obj set-valued, of type $FIN(E)$
$obj.f = x$	$obj'.setf(x')$; obj single-valued $E.setAllf(obj', x')$; obj set-valued, of type $FIN(E)$
$result = val$	$result = val'$;
$val : result$	$result.add(val')$;

Table 3
Java update form of LOCA expressions

if the feature to be updated is modifiable, ie, it is not a *readOnly* feature in the UML model (in UML-RSDS we also forbid update of *input* attributes, representing sensor or other input data).

If there is no update form for the succedent of a constraint, then the query form of the succedent is added to the precondition of each operation implementing the constraint. In Scrabble, the constraint

$$wordsFormed.text <: allWords$$

is such a case, so that $addwordsFormed(wd)$ has precondition $wd.text : allWords$.

4.1 Conjunction Refinement

Given a set of class or association invariants, code can generated for operations $setf$, $addr$, $remover$ for any feature f or role r of the class/the classes linked by the invariants. This code is designed to preserve the invariants by modifying features in response to the operation, or by asserting preconditions to prevent the operation being invoked in situations where it would violate an invariant.

This code synthesis process can often be carried out in a compositional manner, ie, code for different subsets S_1 and S_2 of the invariants can be synthesised separately and then combined to provide code for $S_1 \cup S_2$.

A pair of constraints

$$A \rightarrow B$$

$$C \rightarrow D$$

can be implemented by

```
if (A') { Code_B }
if (C') { Code_D }
```

provided that:

$$wr(Code_B) \cap wr(Code_D) = \emptyset$$

$$wr(Code_D) \cap free(A') = \emptyset$$

The first condition is necessary because otherwise $Code_D$ could undo the changes implemented by $Code_B$. The second is necessary so that $Code_D$ cannot make A' true.

The general case of this rule is:

$$A_1 \rightarrow B_1$$

$$\vdots$$

$$A_n \rightarrow B_n$$

can be implemented by

```
if (A'_1) { Code_B1 }
⋮
if (A'_n) { Code_Bn }
```

provided that:

$$wr(Code_{B_i}) \cap wr(Code_{B_j}) = \emptyset \quad for\ i \neq j$$

$$wr(Code_{B_i}) \cap free(A'_j) = \emptyset \quad for\ j < i$$

If the antecedents of two constraints cannot both be true, then they can

instead be refined by a conditional choice:

$A \rightarrow B$

$C \rightarrow D$

can be implemented by

```
if (A') { Code_B }
else if (C') { Code_D }
```

provided $Code_B$ cannot make C' true and $Code_D$ cannot make A' true:

$not(A' \ \& \ C')$

$A' \rightarrow [Code_B]not(C')$

$C' \rightarrow [Code_D]not(A')$

This is in particular the case for a constraint

$A \rightarrow B$

and its contrapositive

$not(B) \rightarrow not(A)$

The general case for implementation of a series of constraints with pairwise disjoint antecedents is:

$A_1 \rightarrow B_1$

$\vdots$

$A_n \rightarrow B_n$

can be implemented by

```
if (A'_1) { Code_B1 }
else if (A'_2) { Code_B2 }
⋮
else if (A'_n) { Code_Bn }
```

provided that:

$not(A_i \ \& \ A_j) \quad for \ i \neq j$

$A_i \rightarrow [Code_{B_i}]not(\bigvee_{j \neq i} A_j)$

In general, the pattern for implementing a conjunction of constraints is to carry out the implementation of the separate constraints, as in the coordination contracts formalism of [9] and in Aspect-oriented programming [6]. The condition of correctness for this pattern are that the two implementations are

non-interfering.

4.2 Generate and Test

Another common refinement pattern is the 'generate and test' strategy. This is particularly used when assembling a set of elements which satisfy a certain condition. The candidate elements are generated one by one, eg, by an Iterator [4] and then checked to see if they satisfy the property. Those which pass are added to the result set.

In a UML specification this pattern can be applied in two places:

- To produce the result of a query operation, when this is specified as a set of objects with particular properties.
- To maintain an association constraint between one object and all those related to it via the association.

In the second case, the general situation is that classes A and B are related by an association A_B, and a constraint C attached to A_B refers to attributes of both classes (Figure 7). If an event $ax.e$ occurs on one A object ax, and

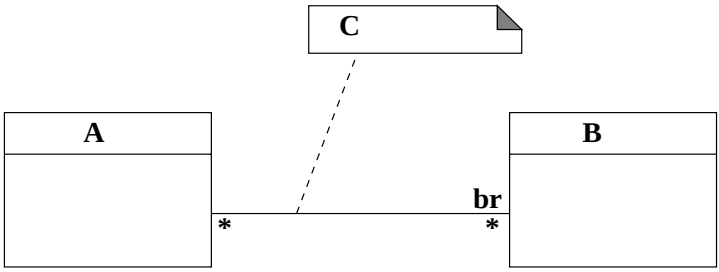

Fig. 7. Association Constraint

this event could affect the truth of C, then the system must:

- Iterate through the collection of B objects bx in $ax.br$ connected to ax by A_B, checking if the constraint, instantiated to ax and bx, remains true or not.
- For those ax, bx where the constraint needs to be re-established, carry out an update action on ax, bx or other connected objects.

How the elements of the association linked to ax are obtained varies depending on the representation of the association.

In Java, we could assume that only one direction of an association is explicitly stored in data. In Table 4 we show the outline Java code used to retrieve all objects related to others via an association $E1 \stackrel{role1}{\longrightarrow}{}^{role2} E2$ from class $E1$ to class $E2$, where only $role2$ has an explicit representation.

$e1s$ denotes the list of all existing instances of $E1$ maintained by the system *Controller* class. A role is of ONE multiplicity if the multiplicity indication at its association end is 1 or 1..1.

UML	*Java*
Class 2 at role 2 (ONE) end of association, class 1 variable is $var1$	$E2\ var2 = var1.getrole2();$ ...
Class 2 at role 2 (non-ONE) end, class 1 variable is $var1$	$for\ (int\ i = 0; i < var1.getrole2().size(); i++)$ $\{\ E2\ var2 = (E2)\ var1.getrole2().get(i);\ \ldots\ \}$
Class 1 at role 1 end, class 2 variable is $var2$, $role2$ is ONE multiplicity	$for\ (int\ i = 0; i < e1s.size(); i++)$ $\{\ E1\ var1 = (E1)\ e1s.get(i);$ $if\ (var1.getrole2().equals(var2))\ \{\ \ldots\ \}\ \}$
Class 1 at role 1 end, class 2 variable is $var2$, $role2$ non-ONE multiplicity	$for\ (int\ i = 0; i < e1s.size(); i++)$ $\{\ E1\ var1 = (E1)\ e1s.get(i);$ $if\ (var1.getrole2().contains(var2))\ \{\ \ldots\ \}\ \}$

Table 4
Mapping of Inter-class Constraints to Java

For example, the constraint $C1$ of Figure 3 leads to the following code:

```
public void setaatt(A ax, int aattx)
{ ax.setaatt(aattx);
  if (aattx == On)
  { for (int i = 0; i < ax.getbr().size(); i++)
    { B bx = (B) ax.getbr().get(i);
      setbatt(bx,On);
    }
  }
}
```

The construction of an interator for a particular data structure or set is itself a complex problem for which different strategies can be applied. For example, a set defined in a recursive manner can often be iterated over using a recursively defined iterator.

Eg, for an iterator *PermutationIterator* to generate all permutations of a list:

> Given list $l = [a_1, a_2, \ldots, a_n]$, its permutations are either: (i) a_1 followed by a permutation of a_2 to a_n, or (ii) a_2 followed by a permutation of a_1, a_3, ..., a_n, ..., (n) a_n followed by a permutation of a_1, ..., a_{n-1}.

These n cases are disjoint, and together give all possible permutations of l.

PermutationIterator can also be defined using this recursion:

- Initiation will be done in constructor $PermutationIterator(l : List)$, which supplies list to be permuted. 'First' permutation is l in original order.

  If l not empty, an *index* variable is set to point to first element of l. A permutation iterator for remainder of l is created.
- Termination testing is done by an operation $hasNext() : Boolean$, true if

further permutations can be generated. If list empty, $hasNext()$ is false, otherwise is true if either *index* is not at the last element of l, or if permutation iterator for sublist (list with index element removed) has next element.

- Stepping to next element has two cases:

(i) If iterator for sublist has a next element, advance it.

(ii) Otherwise, increment *index*, set sublist to the list with index element removed, and create new permutation iterator for sublist.

To get current permutation, get current permutation of sublist, and add indexed element to its head.

4.3 Completion of Partial Operation Specifications

Often a specifier might provide some cases of the definition of a state-changing operation, but not provide a complete definition, ie, a definition which ensures that all constraints are preserved by the operation when it is executed within its precondition.

The requirement is that

$$Invs \ \& \ Pre \ \rightarrow \ [Post]Invs$$

where $Invs$ is the conjunction of constraints possibly affected by the operation, Pre is the conjunction of the operation, and $Post$ is the postcondition, considered as an update transformation/generalised assignment.

The condition P: $Invs \ \& \ Pre \ \rightarrow \ [Post]Invs$ can itself be used as an additional precondition to ensure that the existing pre-post specification of the operation maintains the invariants, since

$$Invs \ \& \ Pre \ \& \ P \ \rightarrow \ [Post]Invs$$

However in many cases the precondition should not be strengthened to this extent, instead the postcondition can be analysed and refined.

Consider the situation in which the postcondition is a disjunction $Post_1$ *or* $Post_2$. Figure 8 shows an example for a system which carries out a generalised sorting in which all elements of a list $l2$ are less than any element of list $l1$, and the size of $l2$ is at most n. For $addElement(x)$ to maintain the invariants, we need:

$$Invs \ \rightarrow \ [Post_1]Invs$$

($P1$) and

$$Invs \ \rightarrow \ [Post_2]Invs$$

($P2$), where $Post_1$ is the first disjunct, etc. This allows us to calculate the

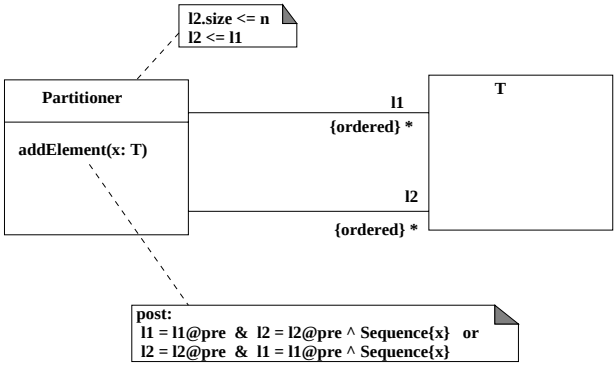

Fig. 8. Sorting by Partitioning

conditions under which each disjunct of the postcondition should execute, refining the disjunction into a conditional statement:

$$if\ (P1)\ then\ Post_1\ else\ if\ (P2)\ then\ Post_2$$

In this example, $[Post_1]Invs$ is

$$(l2{\uparrow}Sequence\{x\}).size \leq n\ \&\ l2{\uparrow}Sequence\{x\} \leq l1$$

so we can propose $P1$ as

$$l2.size < n\ \&\ x \leq l1$$

Likewise, $[Post_2]Invs$ is

$$l2.size \leq n\ \&\ l2 \leq l1{\uparrow}Sequence\{x\}$$

so $P2$ is $l2 \leq x$.

Further, there is a missing case: $not(P1\ or\ P2)$ which is the condition under which the original specification of the operation does not ensure invariant preservation. In the example this predicate is

$$l2.size = n\ \&\ not(l2 \leq x)$$

and we can devise a new postcondition to handle this case (moving the largest element of $l2$ into $l1$ before adding x to $l2$). Alternatively the precondition of the operation (in general) can be strengthened by $P1\ or\ P2$.

Analysis of post and pre conditions is facilitated by the UML-RSDS to B translation supported by the UML-RSDS tools [10]. These map a UML pre/post specification to the schematic B statement

```
PRE Pre
THEN
  ANY v WHERE v : T & Post[v/att,att/att@pre]
  THEN att := v
```

```
  END
END
```

where att is the list of attributes in $wr(Post)$ and v is a list of new variables, one for each of att. T is the list of types of the att.

4.4 Inheritance

If a class D is a subclass of class C, then operations such as $addf$, $removef$ and $setf$ for a feature f of C only need to redefined in D if:

- there are new constraints in D which involve features which are linked (via constraints) to f.

If a new constraint of D replaces (strengthens) a constraint of C, then only the new constraint needs to be considered. If there are no new constraints in D then none of C's operations need to be redefined in D.

The re-definition of an existing operation in D could have the general form:

```
op(pars)
{ if (new preconditions && old preconditions)
  { super.op(pars);
    new updates;
  }
}
```

where 'old preconditions' are the preconditions of the C version of the operation.

5 Related Work

Previous work on refinement patterns includes that of [5], with strategies such as 'replace constant by a variable', 'delete conjunct' and 'generalise equality to inequality' for transforming an operation postcondition into a loop invariant. All of these strategies can be used in our formalism. Strategies for generation and transformation of efficient code were also devised by the KIDS work [14], including transformations to replace recursive definitions of a function by iterative implementations, and automated selection of appropriate algorithms for specific problems. Our framework extends this approach by generating designs appropriate for object-oriented programs.

Many design patterns [4,2] can also be seen as transformations [8], defining how a non-optimal design can be replaced by a functionally equivalent but more modular and flexible design. Finally, program transformations have become an area of much current interest, under the name of 'refactorings' [3].

The approach taken in this paper contrasts with these approaches in that it is 'specification driven': the choice of which code structure to generate is based on the form of the specification model invariants, and does not assume any pre-existing implementation.

6 Conclusion

We have shown that systematic techniques for refining declarative UML specifications to executable code can be defined, and supported by tools. Future work includes the development of a complete catalogue of model transformations and refinement patterns for UML-RSDS, and their incorporation in tools. Transformations specific to web applications, including the introduction of web system design patterns [2], will also be implemented.

References

[1] E. Clarke, W. Heinle, *Modular translation of statecharts to SMV*, technical report, Carnegie Mellon University, 2000.

[2] Alur, D., Crupi, J., Malks, D., *Core J2EE Patterns*, Prentice Hall, 2001.

[3] Fowler, M., *Refactorings – Improving the Design of Existing Code*, Addison Wesley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison Wesley, 1994.

[5] D. Gries, *The Science of Programming*, Springer-Verlag, 1981.

[6] I. Jacobson, *Use Cases and Aspects*, Journal of Object Technology, July/August 2003.

[7] K. Lano, *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag, 1996.

[8] K. Lano, A. Evans, *Rigorous Development in UML*, FASE 1999.

[9] K. Lano, J. Fiadeiro, L. Andrade, *Software Design using Java 2*, MacMillen, 2002.

[10] K. Lano, D. Clark, K. Androutsopoulos, *UML to B: Formal Verification of OO Models*, IFM 2004.

[11] K. Lano, D. Clark, K. Androutsopoulos, *RSDS, a subset of UML with Precise Semantics*, L'Objet, Vol. 9, No. 4, 2003, pp. 53–73.

[12] K. Lano, *Design for Change: Advanced System Design Using Java, UML and MDA*, Elsevier, 2005.

[13] C. Morgan, *Programming from Specifications*, Prentice Hall, 1998.

[14] D. Smith, *KIDS: A Semiautomatic Program Transformation System*, IEEE TSE, Vol. 16, No. 9, 1990.