Using B to verify UML Transformations

K. Lano

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK

Abstract. This paper describes the use of the B formal method to verify semantic properties of UML graphical models, and the correctness of transformations on these models.

1 Introduction

UML is a large and complex notation, in which many aspects of the semantics remain incomplete or are only expressed in an operational manner, ill-suited for analysis using proof tools. Specific problems include:

1. Complex use of undefined and null values within OCL, and missing/inexpressible axioms of OCL. For example the $s \rightarrow at(i)$ operation does not have a formal semantics in UML 2.0 [22].
2. Lack of semantic consistency properties for individual models and between models of the same system [13]. For example, it is necessary that the state invariants of a state machine defining the classifier behaviour of a particular class should be consistent with the invariants of that class.
3. Lack of grounded interpretation for UML concepts, independent of UML [21].

Many of these problems are due to the lack of an objective semantics [8] for UML, and the desire by the UML community to leave certain semantic aspects open, to support a wider use of the notation across different domains. At the same time, an objective semantics is essential to support reuse (if we don't know what a diagram means, how can we reuse it and its corresponding developed code?) and to support verification (the diagrams should have a mathematical semantics because they are abstract descriptions of mathematically precise artifacts – programs).

We solve some of these problems by defining a subset, UML-RSDS (Reactive system development support), of UML, which has a precise semantics based on ZF set theory and classical predicate calculus, which is also the foundation for the Z [25] and B [1] specification languages. This semantics is entirely independent of UML. B provides a notation in which the semantics of models can be expressed and used for verification and validation of the models. B provides an integration of class diagram and state machine models in single components (B machines).

Figure 1 shows the overall development process supported by UML-RSDS and its accompanying toolset. A developer can construct PIM or PSM class diagrams and state machines using the tool, transform models to improve their

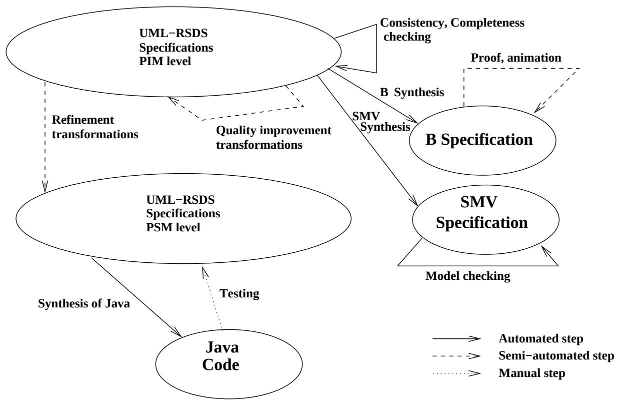

Fig. 1. UML-RSDS Development Steps

quality or refine them, translate to B [1] or SMV [12] for semantic analysis, and generate Java code from a Java PSM.

The translation from UML to B closely corresponds to the semantics of UML-RSDS defined in [19]: both the mathematical elements used to interpret classes, associations, etc, and the structuring/interrelationships of machines parallels the formalisation of models as theories, and the relations of inclusion/extension between these theories. However there are some differences, such as the absence of real numbers in B, and the absence of internal concurrency in B.

2 Specification in UML-RSDS

UML-RSDS specifications consist of:

1. A UML class diagram, including constraints attached to operations, classes and (sets of) associations;
2. A use-case diagram, defining the operations of the system;
3. State machine models attached to classes or operations in the class diagram, or to use cases.

Class attributes can be stereotyped as *input*, *internal*, *derived* or *output*: Derived attributes are prefixed by / as usual. The prefix ? indicates an input attribute and ! an output. These stereotypes are applicable for many different kinds of system, for example, an input field on a web page, or a sensor in a process control system, could both be represented as input attributes.

2.1 Specification Example

An example of a UML-RSDS specification, of part of a robot control system from the production cell case study [20], is shown in Figure 2.

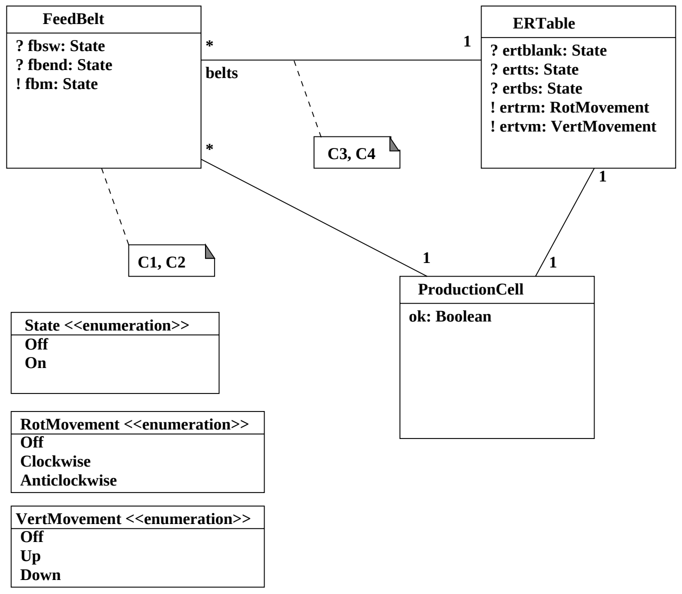

Fig. 2. Class Diagram of Production Cell System

The *FeedBelt* class represents feed belts, which move work pieces (such as car bodies) into the robot production cell. These have a motor *fbm* to move the belt, a switch *fbsw* to switch the belt on and off, and a sensor *fbend* to detect if a piece has reached the end of the belt, ready for unloading into the next component of the cell.

One such component is an 'elevating rotating table', represented by the *ERTable* class. These tables have two motors *ertvm* for vertical movement and *ertrm* for rotary movement, and two sensors *ertts* and *ertbs* to detect if the table is at its top or bottom position, respectively. The sensor *ertblank* detects if there is a work piece in the table.

Normally one or more feed belts may feed blanks into a given table. *belts* gives, for each table, the set of belts that feed that table.

Some example constraints in this system are:

- $C1$ "If the belt switch is off, the motor is off":

$$fbsw = Off \;\Rightarrow\; fbm = Off$$

- $C2$ "If there is no blank at the end, the belt keeps moving":

$$fbsw = On \;\&\; fbend = Off \;\Rightarrow\; fbm = On$$

These are local invariants of the *FeedBelt* class.

A constraint which links the feed belt and table classes is:

– $C3$ “If a belt is ready to unload, and its table is ready to receive a blank, then unloading may proceed”:

$$\begin{array}{l} \mathit{fbsw} = \mathit{On}\ \&\ \mathit{fbend} = \mathit{On}\ \& \\ \mathit{ertblank} = \mathit{Off}\ \&\ \mathit{ertts} = \mathit{On} \ \Rightarrow\ \mathit{fbm} = \mathit{On} \end{array}$$

$C3$ is a constraint on the association between *FeedBelt* and *ERTable*: it specifies that, for any pair of feed belt and table objects linked by this association, that the given invariant must hold true. In this system the association represents the physical connection between the robot system components: that the belt is positioned to feed blanks to the table.

2.2 UML-RSDS Constraints

One significant feature of UML-RSDS class diagrams is that constraints may be attached to associations, these represent an implicit universal quantification over all the objects linked by the associations.

Table 1 shows the syntax of constraints 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: +, −, ∗, /, *div*, *mod*, \/, /\ (also written as ∪ and ∩), or ⌢. A comparator operator *op*2 is one of =, /=, <, >, <=, >=, :, <:, /:, / <:. A logical operator *op*3 is one of &, *or*. Identifiers

< *value* >	::=	< *ident* > |	Variable expression.
		< *number* > | < *string* > |	Primitive literal
		< *boolean* >	expressions.
< *objectref* >	::=	< *ident* > |	
		< *objectref* >`.`< *ident* > |	Navigation call expression.
		< *objectref* > `|(` < *expression* > `)`	Select expression.
< *arrayref* >	::=	< *objectref* > |	
		< *objectref* >`[`< *value* >`]`	At expression.
< *factor* >	::=	< *value* > |	
		{ < *valueseq* > } |	Collection literal
		`Sequence`{ < *valueseq* > } |	expressions.
		< *arrayref* > |	
		< *factor* > op1 < *factor* >	Infix binary operation call (1)
< *expression*1 >	::=	< *factor* > op2 < *factor* >	Infix binary operation call (2)
< *expression* >	::=	< *expression*1 > |	
		`(` < *expression* > `)` |	
		< *expression*1 > op3 < *expression* >	Infix binary operation call (3)
< *invariant* >	::=	< *expression* > |	
		< *expression* > `=>` < *expression* >	

Table 1. UML-RSDS Constraint Syntax

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 notation $objs \mid (predicate)$ denotes the select operator, and evaluates to the set of elements of *objs* which satisfy *predicate*.

3 The UML-RSDS Tools

A large toolset has been developed to support UML-RSDS. The tool facilities include:

1. Diagram creation and editing for class diagrams and state machines.
2. Syntactic and semantic checks on diagram correctness, including consistency and completeness of constraints.
3. Transformations on UML models.
4. Automated translations from UML-RSDS specifications into SMV, the B notation, and Java.

The translation and diagram checking operations are fully automated. Transformations are also automatically applied, but must be selected manually by the tool user.

In addition, there is a tool UML2Web for the creation of web applications.

4 Translation from UML-RSDS to B

To semantically analyse UML models, and to animate (test using symbolic execution) models, we use a translation to the B notation [1]. B is an established formal method which has been extensively used in industry, particularly in the European railway industry [9]. It has comprehensive tool support, the B Toolkit [11], Atelier B [7] and B4Free.

The translation from UML-RSDS into B essentially represents the axiomatic UML-RSDS semantics [10, 17, 19] of models in the B language. Each class E is represented by a variable es (the set of instances of E currently existing) and a type E_OBJ with $es \subseteq E\_OBJ$. Each instance attribute att of type T is represented by a map

$$att : es \rightarrow T'$$

where T' is the representation of T in B. Associations are also represented as maps, Table 2 shows the most common cases.

Ordered associations are represented in a similar manner, except that the range type of the B representation is $seq(bs)$ instead of $\mathbb{F}(bs)$. Table 3 shows the interpretation of some basic expressions.

The B Toolkit can then be used to check if a UML specification has a model, non-trivial models, or to animate the specification. It can also be used to compare

Association	*B role type*	*B invariants*
$\mathrm{A}_{*}{-}_{*}^{r}\mathrm{B}$	$r : as \rightarrow \mathbb{F}(bs)$	
$\mathrm{A}_{0..1}{-}_{*}^{r}\mathrm{B}$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall\, a.(a \in as \Rightarrow r(a) \cap union(r[as - \{a\}]) = \varnothing)$
$\mathrm{A}_{1}{-}_{*}^{r}\mathrm{B}$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall\, a.(a \in as \Rightarrow r(a) \cap union(r[as - \{a\}]) = \varnothing)$
		$union(r[as]) = bs$
$\mathrm{A}_{*}{-}_{1}^{r}\mathrm{B}$	$r : as \rightarrow bs$	
$\mathrm{A}_{0..1}{-}_{1}^{r}\mathrm{B}$	$r : as \rightarrowtail bs$	

Table 2. Representation of Associations in B

OCL	*Semantic Representation in B*
Variable or constant x, primitive or string value x	x
Attribute of single-valued expression $obj.att$	$att(obj)$
Attribute of set-valued expression $s.att$	$att[s]$
Role of single-valued expression $obj.role$	$role(obj)$
Multiplicity ONE role of set-valued expression $s.role$	$role[s]$
Non-ONE role of set-valued expression $s.role$	$union(role[s])$

Table 3. The Interpretation of OCL expressions in B

two models to verify that one is a refinement of another, ie, that all functional properties of one model are also true in a proposed refining model.

Proof obligations for internal consistency of a module in B are:

1. That there is some state which satisfies the module constraints and the typing constraints.
2. That all the constraints are true in the initial state.
3. That each operation, if executed within its precondition, maintains the truth of each constraint.

These correspond directly to similar properties of the UML-RSDS class or subsystem from which the B module was derived. Together they ensure that the constraints are always true, for each object of the class, at time points where no operation is executing on the object provided that operations are only executed within their preconditions (the latter becomes a proof obligation for callers of the operations). Condition 3 ensures that each transition into a state of a state machine attached to a class establishes the invariant of that state.

Animation can be used to check that state invariants of a class are consistent with the class invariants: it should be possible to enter each state of the state machine while satisfying the class invariants.

In the translation to B, the effect of an input event is made explicit: the changes to all objects affected by the event are defined in the B operation which represents the event. The semantics of inheritance, state machines and dynamic binding are also made explicit in the B translation.

The translation to B uses a pragmatic approach which attempts to make the resulting B specification as modular as possible, to enable a close correspondence between the B and UML, and to improve the feasibility of proof. Classes A and B are translated to separate B machines A and B unless:

1. A and B are linked by inheritance: all descendents of a class E without ancestors are grouped into a single machine E.
   If instead we require A and B to be represented by separate machines, the transformation 'replace inheritance by association' can be applied before translation to B.
2. A and B are members of a cycle of (directed) associations: an association $E \rightarrow_r F$ is represented as a variable r of E which refers to a variable of F, so that machine E USES machine F. Cycles are not permitted in the USES relationship, so if there are dependencies in both directions the machines for A and B must be extended by a third machine S representing the complete subsystem of A, B and their linking associations together. The variables representing the associations and the operations on these are placed in S.

Figure 3 shows an example of how UML structures are represented in B.

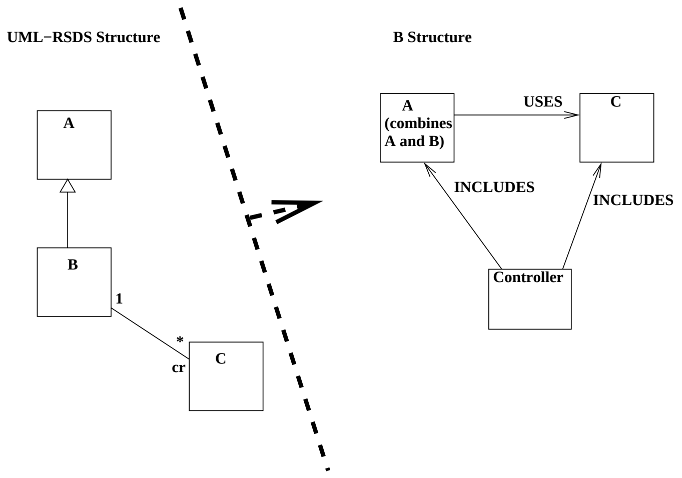

Fig. 3. Structure of B Translation of UML

For the production cell there are no inheritances or cyclic dependencies of classes, so all classes can be specified in separate B machines. A *SystemTypes* machine encapsulates the type definitions of the system:

```
MACHINE SystemTypes
SETS State = {Off, On}; FeedBelt_OBJ; ERTable_OBJ
END
```

The *FeedBelt* machine gives the semantic representation of the *FeedBelt* class, with attributes expressed as maps from the set of existing *FeedBelt* objects (*feedbelts*) to their type sets, and operations synthesised to maintain the class invariants:

```
MACHINE FeedBelt
```

```
SEES SystemTypes
VARIABLES feedbelts, fbsw, fbend, fbm
INVARIANT (feedbelts <: FeedBelt_OBJ) &
  (fbsw : feedbelts --> State) &
  (fbend : feedbelts --> State) &
  (fbm : feedbelts --> State) &
  /* C1: */
  (!feedbeltx.(feedbeltx : feedbelts =>
      ( fbsw(feedbeltx) = Off => fbm(feedbeltx) = Off ))) &
  /* C2: */
  (!feedbeltx.(feedbeltx : feedbelts =>
      ( fbsw(feedbeltx) = On &
        fbend(feedbeltx) = Off => fbm(feedbeltx) = On )))
INITIALISATION feedbelts := {} ||
  fbsw := {} || fbend := {} || fbm := {}
OPERATIONS
    oo <-- new_FeedBelt(fbswx,fbendx,fbmx) =
    PRE feedbelts /= FeedBelt_OBJ & fbswx : State &
      fbendx : State & fbmx : State &
      ( ( fbswx = Off => fbmx = Off ) &
        ( fbswx = On & fbendx = Off => fbmx = On ) )
    THEN
      ANY feedbeltx
      WHERE feedbeltx : FeedBelt_OBJ - feedbelts
      THEN feedbelts := feedbelts \/ {feedbeltx} ||
        fbsw(feedbeltx) := fbswx ||
        fbend(feedbeltx) := fbendx ||
        fbm(feedbeltx) := fbmx ||
        oo := feedbeltx
      END
    END;

    setfbsw(feedbeltx,fbswxx) =
    PRE feedbeltx : feedbelts & fbswxx : State
    THEN fbsw(feedbeltx) := fbswxx ||
      IF fbswxx = Off
      THEN fbm(feedbeltx) := Off   /* Derived from C1 */
      ELSE IF fbswxx = On & fbend(feedbeltx) = Off
      THEN fbm(feedbeltx) := On   /* Derived from C2 */
      END
      END
    END;

  ...
END
```

The controller machine defines all externally-available operations of the system, and manages the global invariants.

For each operation such as *setfbsw* there is both a local version *setfbsw* of the operation, defined in the machine representing the owning class of the operation, and a global version *set_fbsw*, defined in the *Controller* machine. The local version carries out those updates of local features due to the operation, whilst the global version carries out updates of non-local features.

The specifications of B operations are generated from UML-RSDS constraints. These specifications are derived from three parts of the UML-RSDS models:

1. The invariant constraints of the system.
2. The pre and post constraints of the operation, together with its declaration in its owning class.
3. The statemachine of the owning class.

Operations which update features may affect the truth of invariant constraints, both local and global. Therefore it may be necessary to define additional effects for the operation, to maintain these constraints.

In general there are five stages in deriving operation code from invariant constraints:

– For each individual constraint:
    1. Identify if the operation can invalidate the constraint, and therefore if new code needs to be added to the operation to ensure that the constraint is not invalidated.
    2. Identify what set of objects throughout the system can be affected by the operation.
    3. Identify what updates are required on each affected object to maintain constraints.
    4. Convert the update and the conditions under which it applies into B notation.
– Integrate the B derived from each individual constraint into an overall effect for the operation.

For example, in the case of *setfbsw*(*fbswxx*), this operation can violate both $C1$ and $C2$. The new updates which need to be added are:

```
fbswxx = Off  =>  AX(fbm = Off)
```

in the first case, and

```
fbswxx = On & fbend = Off  =>  AX(fbm = On)
```

in the second. No further objects in the system are affected by these actions, so the required updates are purely local to the feedbelt *feedbeltx*. The updates become

```
IF fbswxx = OFF THEN fbm(feedbeltx) := Off END
```

and

```
IF fbswxx = On & fbend(feedbeltx) = Off
THEN fbm(feedbeltx) := On
END
```

in B notation. These can be integrated into a single operation using an IF THEN ELSE structure as their conditions are mutually exclusive.

Operation postconditions can modify local features of a class. These updates are specified in the same manner as in constraint succedents, with the addition that the value of a modified attribute *att* at the start of the operation can be referred to as *att*@*pre*.

Underspecified postcondition constraints such as

```
post: f > f@pre
```

can be formalised using the ANY construct of B.

Behavioural statemachines can be attached to a class C, to define how the operations of that class change the state of the class. The transitions of the statemachine can modify local features of the class and also invoke operations of supplier objects. In the translation to B, the local updates are carried out in the local version of the operation, and the non-local are carried out in the *Controller* version.

The correctness of the translation can itself be verified by providing a common semantics for B and UML, and demonstrating that the B translation $T(e)$ of any UML element e has the same semantics as e [2].

The close correspondence between the UML and the B translation permits analysis on the generated B to be interpreted directly in terms of the model it is derived from.

5 Verification of Model Transformations

Transformations on UML models include:

1. Quality improvements, such as removing redundant classes or associations
2. PIM to PSM transformations, such as the replacement of many-many associations by many-one associations (for implementation of a data model in a relational database).
3. Introduction of detailed design elements, such as a design pattern.

A large number of UML model transformations are known in the modelling community, and some, such as transformations of class diagrams to relational database ER diagrams, have been embedded in commercial tools. We also provide a wide range of transformations in the UML-RSDS tools.

However, developers may need to apply variations of known model transformations, or devise new transformations, and the correctness of these should be shown, so that properties of the original system are preserved in the transformed system.

The translation from UML to B described in the previous section can be used for such verification, by using the B concept of formal refinement. Figure 4 shows the approach adopted. The models on the LHS can be combinations of class diagrams and state machines, as in transformations which introduce the State pattern. Transformations between different modelling languages could also

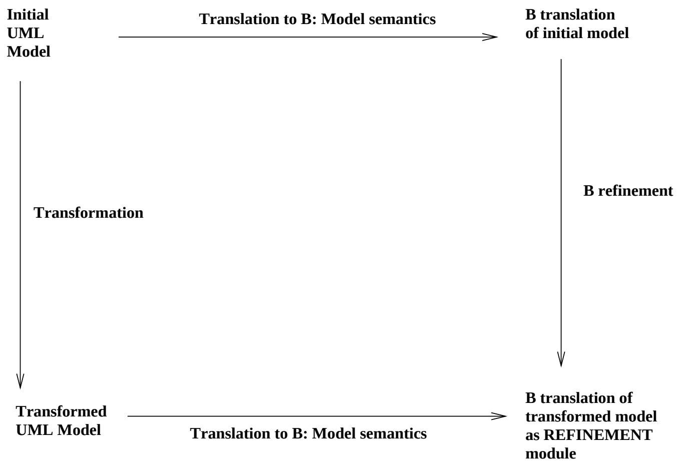
Initial UML Model
Translation to B: Model semantics
B translation of initial model
Transformation
B refinement
Transformed UML Model
Translation to B: Model semantics
B translation of transformed model as REFINEMENT module

Fig. 4. Transformation Verification

be proved correct, provided both languages have a semantics expressible using B.

A model transformation to be verified is expressed in a general form, and both the original model and the transformed model are translated to B, which expresses their semantics (the translation to B is performed automatically by the UML-RSDS tool). The transformed model is defined in a B module which is declared as a REFINEMENT of the B module which expresses the semantics of the original model. The B proof obligations for refinement can then be generated using a tool for B. These obligations are:

1. That the static invariants of the original module remain true (under the data transformation) in the refined module.
2. That the possible initialisations of the refined module correspond to possible initialisations of the original module.
3. That for each operation *op* of the refined module, its behaviour as defined in the refined module is consistent with its behaviour as defined in the original module. More precisely, each possible execution of the refined version of *op* corresponds under the data transformation to a possible execution of the original version.

The refinement proof in B establishes that all pre-post properties of operations and that all invariant properties of the original UML model are also valid in the transformed model.

Each such proof verifies a general family of model transformations. For example, consider the transformation 'replace many-many association by two many-one associations' shown in Figure 5.

The B module representing the semantics of the original model is:

```
MACHINE Model1
```

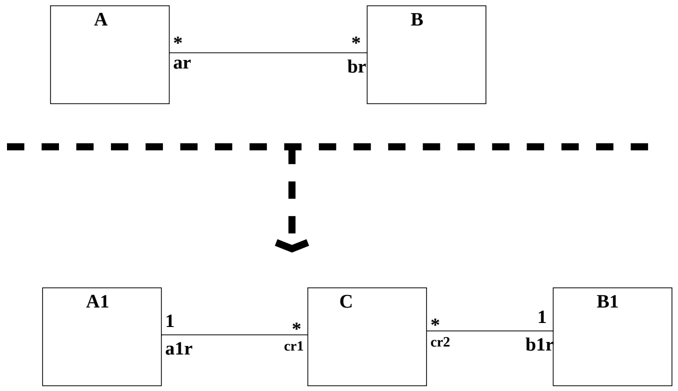

Fig. 5. ‘Replace many-many association’ Transformation

```
SETS A_OBJ; B_OBJ
VARIABLES as, bs, ar, br
INVARIANT (as <: A_OBJ) & (bs <: B_OBJ) &
  (ar: bs --> FIN(as)) &
  (br: as --> FIN(bs)) &
  !ax.(ax : as =>
       !bx.(bx : bs & bx : br(ax) => ax : ar(bx))) &
  !ax.(ax : as =>
       !bx.(bx : bs & ax : ar(bx) => bx : br(ax)))
INITIALISATION
  as := {} || bs := {} || br := {} || ar := {}
OPERATIONS
  addbr(ax,bx) =
    PRE ax: as & bx: bs
    THEN
      br(ax) := br(ax) \/ {bx} ||
      ar(bx) := ar(bx) \/ {ax}
    END:
  ...
END
```

There are also operations to create A and B instances, and to remove elements from the association, etc.

The model of the new system has the formalisation:

```
REFINEMENT Model2
REFINES Model1
SETS C_OBJ
VARIABLES a1s, b1s, a1r, b1r
INVARIANT (a1s <: A_OBJ) & (b1s <: B_OBJ) & (cx <: C_OBJ) &
  (a1r: cs --> a1s) & (b1r: cs --> b1s) &
  (cr1: as --> FIN(cs)) & (cr2: bs --> FIN(cs)) &
  !ax.(ax : a1s =>
       !cx.(cx : cs & cx : cr1(ax) => ax = a1r(cx))) &
  !ax.(ax : a1s =>
       !cx.(cx : cs & ax = a1r(cx) => cx : cr1(ax))) &
  !bx.(bx : b1s =>
       !cx.(cx : cs & cx : cr2(bx) => bx = b1r(cx))) &
  !bx.(bx : b1s =>
       !cx.(cx : cs & bx = b1r(cx) => cx : cr2(bx))) &

  a1s = as & b1s = bs &
  !ax.(ax : a1s => br(ax) = b1r[cr1(ax)]) &
  !bx.(bx : b1s => ar(bx) = a1r[cr2(bx)])
INITIALISATION
  a1s := {} || b1s := {} || cs := {} ||
  b1r := {} || a1r := {} || cr1 := {} || cr2 := {}
OPERATIONS
  addbr(ax,bx) =
    PRE ax: a1s & bx: b1s
    THEN
      IF bx : b1r[cr1(ax)]
      THEN skip
      ELSE
        ANY cx WHERE cx : C_OBJ - cs
        THEN
          cs := cs \/ {cx} ||
          a1r(cx) := ax ||
          b1r(cx) := bx ||
          cr1(ax) := cr1(ax) \/ {cx} ||
          cr2(bx) := cr2(bx) \/ {cx}
        END
      END
    END;
  ...
END
```

The last four invariant conjuncts describe the refinement relation corresponding to the data transformation, and they define how the data of the original model is interpreted in terms of the new model.

The invariants of the original model must be proved correct for these interpretations, for example the property that ar and br are inverse roles:

```
  !ax.(ax : as =>
       !bx.(bx : bs & bx : br(ax) => ax : ar(bx)))
```

must hold in the form:

```
  !ax.(ax : a1s =>
       !bx.(bx : b1s & bx : b1r[cr1(ax)] => ax : a1r[cr2(bx)]))
```

This is proved by using the corresponding properties of the pairs of inverse roles $a1r$ and $cr1$ and $b1r$ and $cr2$.

For each operation, each execution of the operation according to the *Model*2 definition must satisfy the *Model*1 specification of the operation, under the interpretation of *Model*1 data in *Model*2. Informally this is clear for *addbr*, since if there is not already a *cx* with

$$ax = a1r(cx) \ \& \ bx = b1r(cx)$$

then such a *cx* is created and results in *bx* being added to $b1r[cr1(ax)]$, and *ax* to $a1r[cr2(bx)]$ as required. Formal proof of the transformation requires precise assumptions (which might be neglected in informal definitions of UML transformations). In this case we require that no memory problems occur, and that it is always possible to allocate a new *cx* object as required in the new definition of *addbr*. We ensure this by fixing C_OBJ as isomorphic to $A\_OBJ * B\_OBJ$, and only permitting at most one *cx* object to be linked to a particular pair (ax, bx) of *A* and *B* elements.

6 Related Work

Related work on UML is the U2B tool of Butler [24] as part of the RODIN project [23], and translations [14] from UML to Object-Z. Constraints which need to be manually specified in U2B are provided automatically for the developer by UML-RSDS, such as preconditions for *addrole* operations on injective associations [4].

Verification of UML transformations is also treated in [15], using algebraic interpretations, however this has limitations (simple patterns such as Value Object cannot be treated, for example) which our approach avoids. Modelling transforma tions in OCL is another alternative [6], however there are no proof tools available for OCL comparable to the tools available for B. Likewise, the approaches of [5] and [26], using abstract machines (ASMs) and graph transformations, respectively, are limited by the lack of proof support for these representations. B is a more semantically transparent (closer to ZF set theory) representation than ASM. We also provide direct support for models enhanced with OCL constraints, which these last two approaches do not.

Our approach to UML development is similar to that of [3], which carries out performance analysis of a system specified in UML, by means of a translation to a process algebra and analysis tools for this algebra. However this translation is manual, which increases the cost and the risk of introducing errors, compared to automated translations.

References

1. J-R Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Androutsopoulos, K., *Verification of Reactive System Specifications using Model Checking*, PhD thesis, King's College, 2004.

3. A. Bennett, A. Field, *Performance Engineering with the UML Profile for Schedulability, Performance and Time: A Case Study*, Proc. IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Volendam, Netherlands, October 2004.
4. Butler, B., Leuschel, M., Snook, C., *Tools for system validation with B abstract machines*, ASM 2005 - International Workshop on Abstract State Machines, Paris, 2005.
5. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E., *Semantic Anchoring with Model Transformations*, ECMDA 2005.
6. Clark T., Evans A., Girish M., Sammut P., Willans J., *Modelling Language Transformations*, L'Objet. Vol. 9, No 4, pp. 31–51, 2003.
7. ClearSy System Engineering, *Atelier B*, http://www.atelierb.societe.com/, 2004.
8. S Cook. UML semantics. MSDN Weblog, December 2004.
9. Patrick Behm, Paul Benoit, Alain Faivre, Jean-Marc Meynadier, *Meteor: A Successful Application of B in a Large Project*, in Proceedings of FM'99: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Jeannette Wing, Jim Woodcock and Jim Davies (Editors).
10. Bicarregui, J., Lano, K., Maibaum, T., *Objects, Associations and Subsystems: a hierarchical approach to encapsulation*, ECOOP 97, LNCS, 1997.
11. B-Core UK Ltd., The BToolkit, 2005.
12. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, J., *Symbolic Model Checking: 10^{20} States and Beyond*, Proceedings of the Fifth Annual Symposium on Logic in Computer Science, 1990.
13. M Glinz. Problems and deficiencies of UML as a requirements specification language. In *Proceedings of 10th International Workshop on Software Specification and Design (IWSSD-10)*, pages 11–22, 2000.
14. Kim, S., Carrington, D., *A Formal Mapping Between UML Models and Object-Z Specifications*, in ZB2000, LNCS Vol. 1878, Springer-Verlag, 2000.
15. P. Kosiuczenko, *Redesign of UML Class Diagrams: a formal approach*, Munich University, 2003.
16. Lano, K., *Logical Specification of Reactive and Real-Time Systems*, *Journal of Logic and Computation*, Vol. 8, No. 5, pp 679–711, 1998.
17. Lano, K., Clark, D., Androutsopoulos, K., *Formalising Inter-model Consistency of the UML*, UML 2002, Workshop on Consistency Analysis of UML.
18. K. Lano, D. Clark, K. Androutsopoulos, *RSDS: A Subset of UML with Precise Semantics*, L'Objet, Vol. 9, No. 4, 2003, pp. 53–73.
19. K. Lano, *A Compositional Semantics of UML-RSDS*, submitted to SoSyM, 2006.
20. C. Lewerentz, T. Lindner, *Formal Development of Reactive Systems. Case Study Production Cell*, LNCS Vol. 891. Springer-Verlag, 1995.
21. A. Naumenko, A. Wegmann, *Triune Continuum Paradigm and Problems of UML Semantics*, icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200344.pdf
22. OMG, *UML OCL 2.0 Specification, ptc/2005-06-06*, http://www.omg.org/uml/, 2005.
23. RODIN FP6 IST project, http://rodin.cs.ncl.ac.uk, 2006.
24. Snook, C., Butler, M., *U2B – A tool for translating UML-B models into B*, in Mermet J., (Ed.), *UML-B Specification for Proven Embedded Systems Design*, Chapter 6. Springer-Verlag, 2004.
25. J Spivey. *The Z Notation*. Prentice Hall, 1990.
26. Varro, D., Pataricza, A., *Automated Formal Verification of Model Transformations*, CSDUML 2003.