UML To B: Formal Verification of Object-oriented Models

K. Lano, D. Clark, K. Androutsopoulos

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK.
{ kcl, david, kelly } @dcs.kcl.ac.uk Phone: 0207 848 2832, Fax: 0207 848 2851

Abstract. The integration of UML and formal methods such as B and SMV provides a bridge between graphical specification techniques usable by mainstream software engineers, and precise analysis and verification techniques, essential for the development of high integrity and critical systems. In this paper we define a translation from UML class diagrams into B, which is used to verify the consistency of UML models and to verify that expected properties of these models hold.

Keywords: UML, B, UML-RSDS, Graphical Specifications.

1 Introduction

In the RSDS method [10], a small subset of UML statechart diagrams was used as a specification and design notation for reactive systems. ¿From such specifications, executable code could be synthesised, and B [1] and SMV descriptions produced which allowed the analysis of static and temporal properties of the system. This approach is adequate for small control systems without dynamic reconfiguration. However for more general control systems, and for other types of critical system such as e-commerce, the expressibility of a larger subset of UML is required.

This paper therefore extends RSDS by:

1. Translating general UML class diagram structures into B, including diagrams involving inheritance.
2. Translating a subset of OCL into B.
3. Synthesising the code of methods from OCL constraints.

The extended RSDS notation incorporating UML class diagrams is referred to as UML-RSDS in the following.

2 UML-RSDS Specifications

UML-RSDS specifications consist of:

1. A UML class diagram, including constraints attached to operations, classes and (sets of) associations;

2. Statemachines attached to attributes of classes in the class diagram.

Attributes are stereotyped as *sensor*, *internal*, *derived* or *actuator*: state changes on sensor attributes are used as input events to the system, these then lead to changes to internal and derived attributes of the same and/or different classes, and then to actuator attribute changes, which effect the output of the system. The decoration ? indicates a sensor attribute and ! an actuator, the conventional / decoration is used for derived attributes. These stereotypes are used to guide the translation of classes, as in the approach of [20]. For general software systems, sensors represent inputs to the system (eg, HTTP requests in a web-based system) and actuators outputs from the system (eg, generated web pages).

The constraint language we use, LOCA (Logic of objects, constraints and associations) is a notational varient of OCL 2.0 [15]. Table 1 gives the formal syntax of single-state invariants in LOCA. A $< \mathit{valueseq} >$ is a comma-separated

$< \mathit{value} >$	$::= < \mathit{ident} > \mid < \mathit{number} > \mid < \mathit{string} > \mid$ $< \mathit{boolean} >$
$< \mathit{objectref} >$	$::= < \mathit{ident} > \mid < \mathit{objectref} > . < \mathit{ident} > \mid$ $< \mathit{objectref} > \mid (< \mathit{expression} >)$
$< \mathit{factor} >$	$::= < \mathit{value} > \mid \{ < \mathit{valueseq} > \} \mid < \mathit{objectref} > \mid$ $< \mathit{factor} > \mathrm{op1} < \mathit{factor} >$
$< \mathit{expression1} >$	$::= < \mathit{factor} > \mathrm{op2} < \mathit{factor} >$
$< \mathit{expression} >$	$::= < \mathit{expression1} > \mid < \mathit{expression1} > \mathrm{op3} < \mathit{expression1} >$
$< \mathit{stateinvariant} >$	$::= < \mathit{expression} > \mid < \mathit{expression} > => < \mathit{expression} >$

Table 1. Syntax of LOCA Expressions

list of $< \mathit{value} >$s. An $op1$ is one of $+$, $-$, $*$, $/$, $\backslash/$, $/\backslash$. An $op2$ is one of $=$, $/=$, $<$, $>$, $<=$, $>=$, $:$, $<:$, $/:$, $/<:$. An $op3$ is one of &, *or*. Identifiers are either class names, function names, class features (attribute 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.

There are the following omissions from OCL and specific interpretations:

- When used in constraints attached to associations, the *context* of the LOCA formula is the set of all object pairs linked by these associations. This means that, provided attributes of the connected classes have distinct names, in many formulas we can omit any quantifiers or reference to specific objects completely. For example the constraint $C1$:

  $$locn = occupied \;\Rightarrow\; rte = occupied$$

  in Figure 2 represents the inter-class invariant

  $$\forall\, l : Location;\; r : Route \cdot (r, l) \in Route\_Location \;\wedge$$
  $$l.locn = occupied \;\Rightarrow\; r.rte = occupied$$

It could also be expressed as an invariant of *Route*:

$$occupied : path.locn \;\Rightarrow\; rte = occupied$$

This version is more biased towards a particular implementation where *Route* is responsible for maintaining the constraint, so the first version is preferred in UML-RSDS.

- Metamodel features of OCL are omitted: *oclIsTypeOf*, *oclIsKindOf*, *oclIsNew*, *oclAsType*, *allInstances*, *OclType*, *oclInState*, *OclState*, *OclAny*, *OclExpression*. $x.oclIsKindOf(t)$ is expressible as $x : t$ for class names t. $x.oclInState(s)$ is expressible by $x.att = s$ where s is a state of the statemachine attached to attribute *att*. $C.allInstances()$ is expressed by the name C by itself.
- The procedural operator *iterate* is omitted.

OCL navigation expressions, including implicit flattening of collections, can be used in LOCA.

Some concepts from B have been adopted, for example, the use of set equality to avoid the use of quantifiers:

$$r.path = l$$

for a specific route r and location l in Figure 2 has semantics $path(r) = \{l\}$.

The result is a pure logical language which allows more concise expression of inter-class constraints than conventional OCL, where constraints are normally only attached to operation pre and post conditions and to individual classes. Such inter-class constraints are essential in abstract implicit specifications of system behaviour.

2.1 UML-RSDS Specification of Railway Signalling System

An example of a UML-RSDS specification for a railway signalling system [5, 11] is shown in Figure 2. This system supports dispatchers in organising the set-up and cancellation of routes and the blocking of parts of the network (eg, for maintenance) whilst enforcing safety rules such that at most one train can occupy a given route.

The railway network consists of a set of *locations*, each of which has an occupancy detector, and may also have a signal and/or a switch (Figure 1). Each switch may be in either the normal (straight through) position or reverse (directing the train to the side) position, or neither. A train can only travel over the point if it is in the normal or reverse position. Each switch has sensors *swn*, *swr* to detect its position, and motor *swset* to change its position. Each signal may be set to clear (allowing full speed) or half, or stop. The network is divided into *routes*, which consist of a sequence of adjacent track locations extending from a location containing a signal, the *lead signal* of the route, to a location immediately before the next signal in the same direction. In Figure 1 the four routes have been indicated.

The dispatchers can issue commands of the form *cmd locn*, where *locn* is the number of the location the command is intended for. *cmd* may be one of the following:

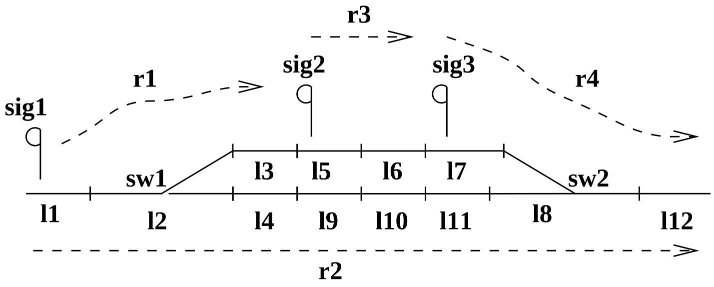


Fig. 1. Example Track Layout

- *sigclr* – set the signal clear;
- *sigstp* – set the signal to stop;
- *sigblk* – set the signal blocked;
- *sigubk* – unblock the signal;
- *swrev* – set the switch to reverse;
- *swnor* – set the switch to normal;
- *swblk* – block switch;
- *swubk* – unblock switch;
- *tkblk* – block track;
- *tkubk* – unblock track.

There may be checks required before a command can be accepted, for example a *sigclr* command can only be obeyed if the route associated with the signal is unoccupied, not blocked, none of its locations are used in another active route, none of the switches in the route are blocked, and they are all in a reverse or normal state, and the signal is not blocked. Alarms are raised if an invalid command is issued.

Figure 2 shows the abstract analysis model of the system as a UML-RSDS class diagram.

Some example constraints are:

1. $C1$ "If any location in a route is occupied, the route is occupied":

   $$locn = occupied \;\Rightarrow\; rte = occupied$$

2. $C2$ "If any location in a route is blocked, so is the route":

   $$locstatus = blocked \;\Rightarrow\; routestatus = blocked$$

   Similarly if any signal in the route is blocked, or if any switch in the route is blocked.
3. $C3$ "If a route is not ready (some switch is in neither the normal or reverse position) then it is not traversable":

   $$ready = false \;\Rightarrow\; traversable = false$$

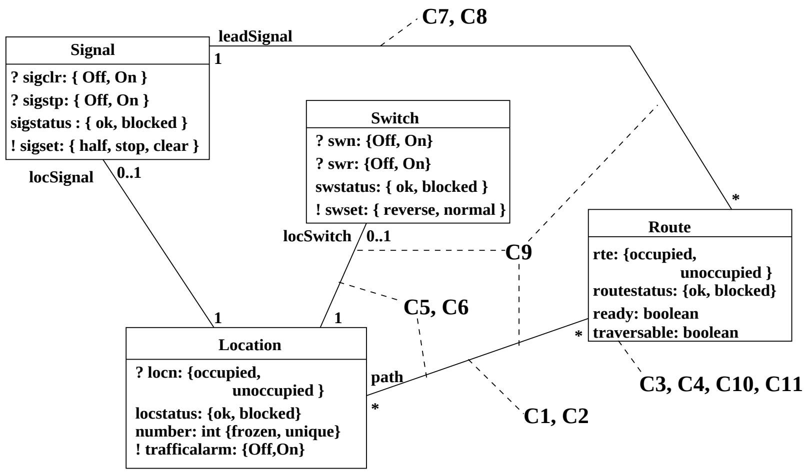


Fig. 2. Abstract Analysis Class Diagram

4. $C5$ "A route cannot be entered if some switch is not set to either normal or reverse":

$$swr = Off \ \& \ swn = Off \ \Rightarrow \ ready = false$$
$$ready = false \ \Rightarrow \ leadSignal.sigset = stop$$

5. $C7$ "If a route is occupied, its signal must be set to stop":

$$rte = occupied \ \Rightarrow \ sigset = stop$$

6. $C8$ "If a route is blocked, its signal must be set to stop":

$$routestatus = blocked \ \Rightarrow \ sigset = stop$$

Some operation specifications describing the effect of operator commands are as follows, where $AX(P)$ denotes that P holds in the next state, ie, at completion of the reaction to the event. This corresponds to P being a postcondition constraint of the operation in standard OCL notation.

– *sigstp*:

$$sigstp \ \& \ sigstatus = ok \ \Rightarrow \ AX(sigset = stop)$$

– *swnor*:

$$swnor \ \& \ rte = unoccupied \ \& \ swstatus = ok \ \&$$
$$routestatus = ok \ \Rightarrow \ AX(swset = normal)$$

– The blocking and unblocking commands have no validation checks, and simply modify the values of the *locstatus*, *swstatus* and *sigstatus* variables as appropriate (and therefore also possibly the values of *routestatus* and the state of signals).

Finally we can express environmental constraints whose failure will trigger the creation of a traffic alarm for the location concerned, using the RSDS fault detection architecture [10].

- A switch is failed if its *swn* and *swr* sensors are both set:

$$swn = On \;\Rightarrow\; swr = Off$$

- Similarly for signals:

$$sigclr = On \;\Rightarrow\; sigstp = Off$$

- A location should not become occupied if it is blocked:

$$locnoccupied \;\Rightarrow\; locstatus = ok$$

- A route should only be entered by its initial location:

$$locnoccupied \;\&\; rte = unoccupied \;\Rightarrow\; locSignal = leadSignal$$

The usual rules of $\Rightarrow$-transitivity and contraposition apply to LOCA constraints: if $A \;\Rightarrow\; B$ is a system constraint on the set r of associations, and $B \;\Rightarrow\; C$ is on the set s of associations, so are $A \;\Rightarrow\; C$ on set $r \cup s$, and $not(B) \;\Rightarrow\; not(A)$ on set r, provided that the names of all attributes in the classes are unique in the set of connected classes.

3 Translation from UML-RSDS to B

We use the B language [9] as a formal notation for expressing the precise semantics of a UML-RSDS model and for reasoning about this model. The translation makes explicit the meaning of the LOCA constraints of the model, and respects the structure of the model, with individual classes being represented by distinct machines in the translation.

3.1 Translation of Types

An enumerated type $T = \{v_1, \ldots, v_n\}$ is translated to a corresponding SET in B. The enumerated types of attributes in the class diagram (corresponding to the states of their statemachines) are defined in the *SystemTypes* B machine, this is then accessed via *SEES* by all other generated B components.

The OCL *Boolean* type is interpreted by $BOOL$ (defined as the set $\{TRUE, FALSE\}$ in the $Bool\_TYPE$ machine) in B. *String* is interpreted by $STRING$ in B. The OCL *Integer* type is translated to B's INT. This translation is inexact as the OCL type is the mathematical set of integers whilst the B type is bounded.

The OCL *Real* type is not translatable to B.

A class type C is translated to a type C_OBJ in *SystemTypes* representing all possible instances of C, and a variable $cs \subseteq C\_OBJ$ representing the set of existing instances of C. This variable will be located in the B machine that represents C.

3.2 Translation of Classes and Associations

For a class C with attributes $att_1 : T_1$ to $att_n : T_n$ and roles $role_1$ of cardinality c_1 to class D_1, ..., $role_m$ of cardinality c_m to class D_m (Figure 3), we define a machine $\mathcal{B}(C)$ using the usual translation [9] of attributes and associations to maps:

```
MACHINE C
SEES SystemTypes
USES D1, ..., Dm
VARIABLES cs, att1, ..., attn, role1, ..., rolem
INVARIANT cs <: C_OBJ &
  att1: cs --> T1 & ... & attn: cs --> Tn &
  role1: cs --> S1 & ... & rolem: cs --> Sm
INITIALISATION
  cs := {} || att1 := {} || ... || attn := {} ||
  role1 := {} || ... || rolem := {}
OPERATIONS
  cx <-- new_C(att1x,...,attnx,rle1x,...,rlekx) =
    PRE att1x: T1 & ... & attnx: Tn &
        rle1x: P1 & ... & rlekx: Pk & cs /= C_OBJ
    THEN
      ANY oox WHERE oox: C_OBJ - cs
      THEN
        cs := cs \/ { oox } ||
        att1(oox) := att1x ||
        ...
        attn(oox) := attnx ||
        rle1(oox) := rle1x || /* for cardinality 1 roles */
        ...
        rlek(oox) := rlekx ||
        role(oox) := {} || /* for roles with cardinality /= 1 */
        cx := oox
      END
    END;

  att1x <-- getatt1(cx) =
    PRE cx: cs THEN att1x := att1(cx) END;

  ...

  attnx <-- getattn(cx) =
    PRE cx: cs THEN attnx := attn(cx) END;

  role1x <-- getrole1(cx) =
    PRE cx: cs THEN role1x := role1(cx) END;

  ...

  rolemx <-- getrolem(cx) =
    PRE cx: cs THEN rolemx := rolem(cx) END;
```

```
  setatt1(cx,val1) =
    PRE cx: cs & val1: T1 THEN att1(cx) := val1 END;

  ...

  setattn(cx,valn) =
    PRE cx: cs & valn: Tn THEN attn(cx) := valn END;

  setrole1(cx,role1x) =
    PRE cx: cs & role1x: S1 THEN role1(cx) := role1x END;

  ...

  setrolem(cx,rolemx) =
    PRE cx: cs & rolemx: Sm THEN rolem(cx) := rolemx END
END
```

The machine $\mathcal{B}(C)$ may need to SEE $Bool\_TYPE$, Int_TYPE or $String\_TYPE$ if the corresponding data types are used as one of the T_i. If an initial value is

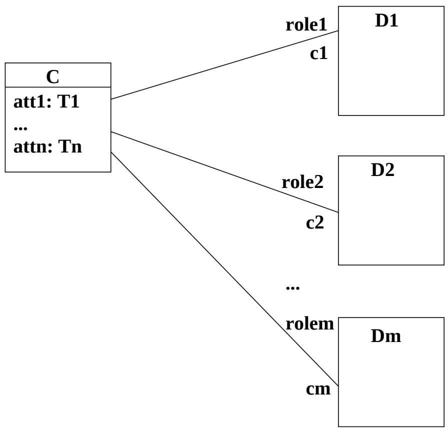


Fig. 3. Class with Attributes and Associations

specified in the class diagram for an attribute, or there is a statemachine with an initial state for the attribute, then this value is used instead of a parameter in new_C. If $role_j$ has ONE cardinality: $c_j = 1..1$, then S_j is simply the set d_js of existing instances of D_j, and there is a parameter $role_jx$ and assignment $role_j(oox) := role_jx$ in the constructor operation. Otherwise, S_j is $\mathbb{F}(d_js)$ and the constructor assignment is $role_j(oox) := \{\}$, there is no parameter[1]. The

[1] Provided c_j has a lower bound of zero.

rle$_1$, ..., *rle*$_k$ above are all the cardinality ONE roles, *role*$_l$, with P_i being the corresponding $d_l s$.

For roles with a MANY (ie, $a..b$ with $a < b$, $a..*$ or $*$) cardinality there are also $addrole(cx, obj)$, $removerole(cx, obj)$ operations which add/remove obj from $role(cx)$, and $unionrole(cx, objs)$ and $subtractrole(cx, objs)$ which add and remove sets *objs* of objects from *role*. Apart from $*$ cardinality roles, these have preconditions to ensure cardinality constraints.

Operations $setAllfeat(oos, val)$, $addAllrole(oos, obj)$, $removeAllrole(oos, obj)$, $unionAllrole(oos, objs)$, $subtractAllrole(oos, objs)$ which perform multiple updates are also included, so they can be used by code synthesised from constraints.

For each association, invariants are derived from its role cardinalities. For example if *role* has target cardinality 1 .. 2 then the invariant

```
!cx.(cx: cs => card(role2(cx)): {1,2})
```

is included in the machine for C. ! denotes the universal quantifier in B notation.

Table 2 shows the particular function types and extra invariants for the most common combinations of role cardinalities.

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

Table 2. Representation of Associations in B

If a UML-RSDS specification is purely declarative, that is, all functionality is specified implicitly via class and inter-class constraints, with no user-defined methods, then an automatically generated *Controller* machine is produced, which INCLUDES all other machines, none of which need to modify each others state and so are only related via USES, which permits (acyclic) dependency structures. An example of this strategy is the B specification generated for the railway signalling system, which is structured as shown in Figure 4.

3.3 Inheritance

Inheritance poses a particular problem in translation of UML to B, since a subclass is dependent on its superclass, and this is an operation dependence (implying the use of INCLUDES in B) since creation of a subclass instance requires invocation of its superclass constructor. However in B, at most one machine can be includes-dependent on a given machine. Thus multiple subclassing of a class in UML cannot be represented using separate machines in B. It is possible to represent linear chains of inheritance using separate machines [22], whereby the

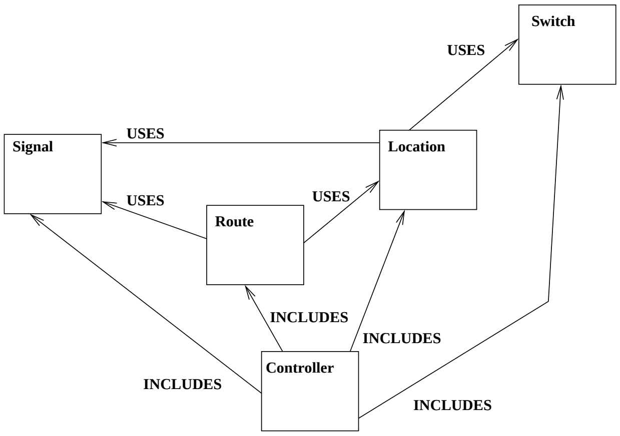
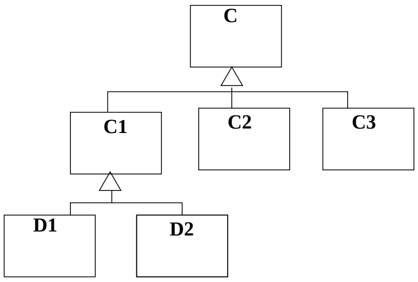


Fig. 4. Structure of Generated B for Signalling System

subclass machine selects the new object to be created and then invokes superclass machine operations to add the instance to their respective instance sets and initialise their local attributes for it, but we prefer to use a consistent approach and always gather together all classes that are related in one inheritance tree and formalise them in a single B machine [9], this is the machine that represents the superclass of all the classes in the tree (Figure 5).




```
MACHINE C
VARIABLES
  cs, c1s, c2s, c3s,
  d1s, d2s
INVARIANT
  cs <: C_OBJ &
  c1s <: cs & c2s <: cs &
  c3s <: cs &
  d1s <: c1s &
  d2s <: c1s
```

Fig. 5. Representation of Inheritance in B

If class D is an immediate subclass of class C then the axiom $ds \subseteq cs$ is adjoined to the machine M representing them. If D and E are distinct direct subclasses of the same class C then the axiom $ds \cap es = \varnothing$ is also adjoined, and if C is an abstract superclass of a complete set $C_1, \ldots, C_n$ of subclasses, then the axiom $c_1 s \cup \ldots \cup c_n s = cs$ is included. Alternatives, such as *overlapping* subclasses, could also be formalised in the same manner. The B constructor *new_D* for an immediate subclass D of a class C must select (using ANY) an instance not a member of any class in the hierarchy that includes C and D, and then add this instance oox to each ancestor instance set es of D as well as to ds. It must also set initial values of all the attributes and ONE roles of each ancestor class.

Method polymorphism can be modelled in B as follows. Assume that method $m(p : T) : R$ is defined in class C and also given definitions in subclasses C_1, C_2, ..., C_n, then the B definition of m is an IF statement in which the case of a more specific class is considered before the case of its immediate superclass, etc. For example if C_1 and C_2 are subclasses of C_3, and C_1 and C_3 define m, the B interpretation would be:

```
mresult <-- m(cx,p) =
  PRE cx: cs & p: T & PreCm
  THEN
    IF cx: c1s THEN DefC1(cx,p)
    ELSE IF cx: c3s THEN DefC3(cx,p)
    ELSE DefC(cx,p) END END
  END
```

where *DefD* is the definition of m in D. The precondition of m in C should imply the preconditions of m in any subclass, so these are omitted.

3.4 Translation of Constraints

There are three separate categories of constraint:

- *local class constraints* – constraints attached to a single class and involving only features of that class. These are translated into invariants of the B machine representing the class, and update code to preserve these invariants is added within the operations of the machine.
- *non-local class constraints* – other constraints attached to a single class. These are translated into invariants of the *Controller* B machine, and update code to preserve these invariants, written within the operations of the *Controller*.
- *association constraints* – constraints attached to one or more associations. These are also translated into invariants and code in the *Controller*.

Table 3 shows the correspondence of some expressions between OCL, LOCA and B. Sequences (such as ordered associations) and operators such as *sum* also have direct equivalents in B. In general for every OCL expression that is in the LOCA subset (wrt change of syntax) there is a direct corresponding B expression.

There are a number of standard UML constraints on model elements [14]: *frozen* (same as *final* in Java), *classifier* scope (same as *static* in Java), *unique* for attributes, *leaf* (same as *final* in Java) and *abstract* for classes, and *classifier* scope, *abstract* and *query* for operations. Associations may also be *frozen*.

Frozen attributes have no *set* operation, otherwise their interpretation in B is as for other attributes. Frozen associations must be initialised in the constructor, regardless of their cardinality, and have no *set* or other update operations.

OCL	*LOCA*	*B*
r1.max(r2)	{ r1, r2 }.max	max({ r1, r2 })
set->union(set2)	set \/ set2	set \/ set2
set->intersection(set2)	set /\ set2	set /\ set2
set->includes(elem)	elem : set	elem : set
set->including(elem)	set \/ { elem }	set \/ { elem }
set->excludes(elem)	elem /: set	elem /: set
set->excluding(elem)	set - { elem }	set - { elem }
set->isEmpty()	set = {}	set = {}
set->select(P)	set | (P)	$\{x \mid x : set \; \& \; P(x)\}$
C.allInstances()	C	cs

Table 3. Correspondence of Expressions from OCL to B

Classifier scope attributes $att : T$ are represented as variables $att : T$ in B, not as functions. Unique attributes of class C are represented using an injective function type $att : cs \rightarrowtail T$, and an extra precondition $val \notin att[cs - \{cx\}]$ is required for $setatt(cx, val)$.

Leaf classes cannot have subclasses. Abstract classes have extent the union of the extents of their direct subclasses.

Classifier scope operations omit the object reference parameter cx. Query operations can only modify their *result* variable, and they cannot use the @*pre* pre-state qualifier in their specification. Abstract operations must be specified in each concrete subclass of the abstract class in which they are declared.

3.5 Translation of Class Invariants

Constraints attached to a class C in UML are interpreted as machine invariants of the machine $\mathcal{B}(C)$. The translation makes explicit the OCL 'flattening' of sets but otherwise is direct. Table 4 shows the translation of attributes and roles. e' denotes the interpretation of expression e into B. Operators :, =, etc in LOCA

LOCA	*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 $obj.role$	$role(obj')$
Cardinality ONE role of set-valued expression $s.role$	$role[s']$
Non-ONE role of set-valued expression $s.role$	$union(role[s'])$

Table 4. Interpretation of Navigation Expressions in B

map to corresponding operators in B. However if one side of an equality $a = b$ is set-valued and the other is not, the non-set, say b, must be expressed as $\{b\}$ in the translation. Similarly for other operators ($x : y$ becomes $x \subseteq y$ if x is set-valued, etc).

Each free variable v occuring in a LOCA formula F must have a determinable type T that can be established from the formula. The result of the B translation then universally quantifies v over the translated formula BF:

```
!v.(v: T => BF)
```

Finally the formula is universally quantified by an object cx of the class concerned.

An example of this translation in the case study is from the constraint

```
locn = occupied  =>  in.leadSignal.sigset = stop
```

to the B invariant

```
!locationx.(locationx: locations =>
     (locn(locationx) = occupied =>
        sigset[leadSignal[in(locationx)]] = {stop}))
```

If there was a many role *in* from *Location* to *Route*.

The B translation of class invariants are placed in the machine representing the class *provided* they only refer to local features of that class. Other invariants are made the responsibility of the system controller.

The constructor new_C in $\mathcal{B}(C)$ must check that the local class invariants hold for the initial values to be assigned to the attributes and roles. Each *set* operation must also do this for the new values assigned to the attribute or role. Code is generated to preserve local invariants involving two or more non-frozen attributes, eg: the operation $setready(obj, readyx)$ must also set $traversable(obj) := false$ if $readyx = false$.

3.6 Translation of Inter-class Constraints

Constraints may be attached to several associations, and therefore also implicitly to the classes related by these associations. In the translation to B, we invent a variable cx for each class C in the context of the constraint, and universally quantify over these variables. Table 5 shows the quantifier range set of the cx variables ranging over the instances of each class in each case of an association $E1^{role1} \longrightarrow^{role2} E2$. Then in the quantified formula F we interpret a reference *feat* to an attribute or role of class E by $feat(vare)$ where $vare$ is the quantified variable created for E. The rules given in Table 4 are also applied if F involves navigation expressions.

As an example of the mapping, the constraint

```
locn = occupied => rte = occupied
```

becomes the B invariant

```
!locationx.(locationx: locations =>
  !routex.(routex: routes & locationx: path(routex) =>
               (locn(locationx) = occupied =>
                         rte(routex) = occupied)))
```

of the *Controller* machine for the signalling system.

LOCA	*B*
Class 2 at role 2 (ONE) end of association where class 1 quantifier is already known, on $var1$	$var2 = role2(var1)$
Class 2 at role 2 (non-ONE) end of association where class 1 quantifier is known, on $var1$	$var2 : role2(var1)$
Class 1 at role 1 end of association where class 2 quantifier is known, on $var2$, $role2$ is ONE	$var2 = role2(var1)$
Class 1 at role 1 end of association where class 2 quantifier is known, on $var2$, $role2$ non-ONE	$var2 : role2(var1)$

Table 5. Mapping of Inter-class Constraints to B Quantifier Restrictions

3.7 Synthesis of Event Code

An invariant such as

$$locn = occupied \;\Rightarrow\; rte = occupied$$

can be interpreted as an instruction to set *rte* to *occupied* for each *Route* object related to a *Location* object *loc* whenever *loc*'s *locn* is set to *occupied*, ie, as a description of the necessary actions to be taken to preserve the truth of the invariant.

In general, for each constraint C, response code for an event α will need to be generated (to maintain C) if

$$Pre_{\alpha} \wedge C \;\Rightarrow\; [\alpha]C$$

is not true, where $[\alpha]C$ denotes the weakest precondition of C with respect to α, and Pre_{α} is the specified precondition of α. The response code will modify a designated *write frame*: for local invariants this is the set of non-frozen data features of the class, excluding the feature directly modified by α. If it is impossible to produce such code, the precondition must be strengthened instead.

In the case of a local invariant of a class, the response code will be expressed using assignments := and no operation calls, within the code of α in the B machine representing the class. For non-local constraints the code synthesis process generates the updates required on all sets of objects related to a particular object via a constraint. If object *oo* of class C changes state via an event $attval(oo)$ of the system, for example, then there will be a set *dset* of objects of a related class D affected, and the constraint will prescribe how their states should change.

Therefore the translation derives the set *dset* of affected objects for each class D related to the event origin class C via the associations of the constraint. Table 6 shows the set of affected objects of each class in each situation.

The constraint conclusion is translated into B using these sets as the arguments of suitable $setAllatt(objectset, val)$, $setatt(obj, val)$, etc, operations. A feature *feat* of class C is interpreted as $feat[cset]$ where *cset* is the set of affected objects of C, or as $union(feat[cset])$ if *feat* is a non-ONE cardinality role.

A number of simplification rules are applied to reduce the complexity of the resulting formula (Table 7).

$LOCA$	B
Class 2 at role 2 (ONE) end of association; class 1 set of objects affected is $set1$	$role2[set1]$
Class 2 at role 2 (non-ONE) end; class 1 set of objects affected is $set1$	$union(role2[set1])$
Class 1 at role 1 end of association; $set2$ are class 2 objects affected, $role2$ is ONE	$\{c1x \mid c1x : c1s \ \wedge \ role2(c1x) : set2\}$
Class 1 at role 1 end of association; class 2 objects affected are $set2$, $role2$ non-ONE	$\{c1x \mid c1x : c1s \ \wedge \ role2(c1x) \cap set2 \neq \varnothing\}$

Table 6. Sets of Objects Affected by an Event

B Expression	*Rewritten Expression*
$f[\{x\}]$	$\{f(x)\}$
$union(\{x\})$	x
$union(\varnothing)$	$\varnothing$
$v \cap \{y\} \neq \varnothing$	$y : v$
$y : \{x\}$	$y = x$
$\{x\} = \{y\}$	$x = y$
$\{x\} \subseteq y$	$x : y$
$setAllfeat(\{x\}, val)$	$setfeat(x, val)$
$addAllrole(\{x\}, obj)$	$addrole(x, obj)$
$removeAllrole(\{x\}, obj)$	$removerole(x, obj)$

Table 7. Simplification of B Expressions

Applying these rules, the constraint

$$locn = occupied \ \Rightarrow \ rte = occupied$$

together with the transitive composition of $C1$ and $C7$ gives the following code for the controller operation $locnoccupied(oo)$:

```
locnoccupied(oo) =
  PRE oo: locations
  THEN
    setlocn(oo,occupied) ||
    setAllrte({ routex |
                  routex: routes & oo: path(routex) }, occupied) ||
    setAllsigset(
         leadSignal[{ routex |
           routex: routes & oo: path(routex) }], stop)
  END
```

All updates which result from a given sensor event must be performed together (using || and IF statements) in the same controller operation.

The *Controller* machine is responsible for maintaining inter-class invariants and coordinating the response to input events. Its typical form is:

```
MACHINE Controller
```

```
SEES SystemTypes
INCLUDES C1, ..., Cn  /* All machines generated for classes */
PROMOTES new_C1, ...
INVARIANTS /* All inter-class invariants */
OPERATIONS
  attvalue(oo) =
    PRE oo: cs
    THEN
      setatt(oo,val) || synthesised code
    END;

  ....
END
```

There is an operation *attval* for each sensor attribute *att* of each class, and each value *val* (for *att* of enumerated type) of its type.

3.8 Operation Specifications

In OCL and LOCA operation behaviour can be specified using pre/post constraints:

```
context E::op(p: T): R
pre: P
post: Q
```

In Q the initial values of variables at start of the operation are referred to as $var@pre$.

Provided only local features are updated, such an operation specification can be translated into a B operation

```
resultx <-- op(ex,p) =
  PRE p: T' & ex: es & P(ex)
  THEN
    ANY v,result
    WHERE Q[x/x@pre,v/x](ex) & v: Tx & result: R'
    THEN
      x(ex) := v || resultx := result
    END
  END
```

where $P(ex)$ denotes the predicate P with each feature f of E replaced by $f(ex)$, feature application $obj.F$ expressed as $F(obj)$, $F[obj]$ or $union(F[obj])$ as appropriate, and there is a new variable v_f for each feature f of E, the substitution of $x@pre$ by x is carried out for each $x@pre$ in Q. B can then be used to check that the operation specification is consistent with (preserves) the local class invariant. If several classes in the same super/subclass heirarchy also define op, with identical parameter type T, then the modelling of polymorphism described in Section 3.3 can be used, where $DefE$ is the ANY statement.

As an example, the specification

```
context E::op()
pre: x < y
post: x > x@pre & y > x
```

where x and y are *int*-valued attributes of E, becomes:

```
op(ex) =
  PRE ex : es & x(ex) < y(ex)
  THEN
    ANY vx,vy
    WHERE vx > x & vy > vx & vx : INT & vy : INT
    THEN
      x(ex) := vx || y(ex) := vy
    END
  END
```

This can be shown to preserve $!ex.(ex : es \;\Rightarrow\; x(ex) < y(ex))$.

A *query* operation only modifies *result* and has a simpler translation.

In UML-RSDS, action invariants $P \;\Rightarrow\; AX(Q)$ are an alternative means to specify state changes. An OCL pre/post constraint can be expressed as an action invariant

$$v = x \;\&\; P \;\Rightarrow\; AX(Q[v/x@pre])$$

where the v are new variables, one for each x such that $x@pre$ occurs in Q.

4 Specification Verification and Validation

The UML-RSDS tool performs basic consistency and completeness checks on sets of constraints. In addition the translation to B can be used to proof check the specification and identify other errors. This is because the translation has itself been verified, that is, the translation into B of a UML-RSDS model M can be shown to validate each of the axioms of the semantics Γ_M of M [13]. This means that if a property can be derived in M, it will also be true in the B translation. In particular, contradictions will be detectable, in principle, in B.

Incompleteness of the UML-RSDS model can also be detected via animation of the translated B, enabling particular scenarios to be symbolically executed. For example in the railway specification the lack of an invariant to reset a *route* to unoccupied if all its locations are unoccupied can be identified in this way (the required invariant is $path.locn = unoccupied \;\Rightarrow\; rte = unoccupied$). B can also be used to check if there are non-vacuous models of a specification.

As an example of property verification using B, it is easy to write UML class diagrams which are valid according to the UML definitions, but which have only vacuous implementations. In Figure 6 there are two classes with associations to each other. Superficially this seems sensible, however the formal semantics of this situation, expressed by the translation to B, is:

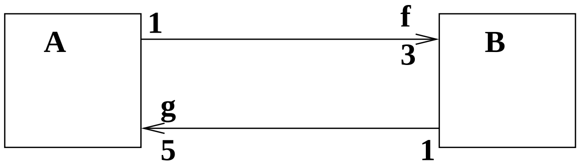


Fig. 6. Vacuous UML Model

$$
\begin{array}{l}
f : as \rightarrow \mathbb{F}(bs) \\
g : bs \rightarrow \mathbb{F}(as) \\
\forall\, a.(a : as \;\Rightarrow\; card(f(a)) = 3) \\
\forall\, b.(b : bs \;\Rightarrow\; card(g(b)) = 5) \\
union(f[as]) = bs \\
union(g[bs]) = as \\
\forall\, a, a'.(a : as \wedge a' : as \wedge a \neq a' \;\Rightarrow\; f(a) \cap f(a') = \varnothing) \\
\forall\, b, b'.(b : bs \wedge b' : bs \wedge b \neq b' \;\Rightarrow\; g(b) \cap g(b') = \varnothing)
\end{array}
$$

This means that the $f(a)$ form a partition of bs, indexed by elements of as, so $card(bs) = 3 * card(as)$, and similarly $card(as) = 5 * card(bs)$. This is only possible if $as = bs = \varnothing$.

The consistency of statechart specifications can also be checked against class diagram invariants: statechart transitions can possess generated actions, which consist of method invocations on supplier objects. For example, a route object could have the generation $leadSignal.setsigset(stop)$ on transitions for $setrte(occupied)$. These explicit commands are asserted to *implement* certain constraints ($C7$ in this case) and their translation into B replaces the B code synthesised from the declarative constraints that they are claimed to implement. Consistency checking the controller machine or machines derived from entities (for local constraints and implementing actions) can be used to verify that the constraints are indeed ensured by the statechart actions.

5 Translations to Java and SMV

A similar translation approach can be used to define an executable implementation of a system in Java [12], and a description in SMV [4] which can be used for automated temporal verification (model checking). The translation into Java is simpler than that to B because of the closer structural relation between Java and UML, and the ability in Java to perform complex updates as a series of individual object updates, whereas in B they must be performed in a single step. Details of the Java translation are given in [12].

The UML-RSDS, B and SMV semantics are very closely related, representing a model or execution of a specification as a sequence of time steps, each step consisting of no events or of a single sensor event and a set of other events (the reaction to the sensor event). Thus the results of analysis in B or SMV can be immediately related to the UML model.

6 Comparison

Related work on UML is the U2B tool of Butler [19], and translations [7] from UML to Object-Z [18]. These translations emphasise the preservation of the UML class diagram structure in the formal notation, as the starting point of a formal development. In both cases the formal notation is central and the UML used only as a diagrammatic front end. In contrast, we envisage that most developers would prefer to use UML as their main notation, and only use B as one analysis tool for property checking and animation of the class diagram. The exact structure of formal modules is therefore less important than the traceability of formal notation elements to the original diagram elements. We have ensured such a traceability is possible by representing each class, attribute and role by a separate variable and each method by a B operation.

Other related tools are: (i) The KeY System. This toolkit [2] provides facilities for verifying object-oriented applications against OCL specifications. In contrast to KeY, UML-RSDS is intended to be used by developers as an OO formal method, and to generate applications from high-level models according to the MDA [16] concepts. We also rely on established predicate logic as the basis of verification (in B) instead of a new dynamic logic. RSDS has also established a modular design methodology including modular verification, which is lacking in KeY.

(ii) The USE Tool. This tool [17] provides validation and verification of UML specifications by means of checking test case object diagrams against the specification. Similar capabilities are provided by UML-RSDS via translation to B, which supports animation and monitoring of preconditions and invariants. Translation to SMV provides additional capabilities of temporal property verification, and identification of sequences of behaviour which are counter-examples to system invariants.

(iii) Alloy. The Alloy constraint language and tools [6] are also intended to provide a lightweight specification notation and semantic error detection capabilities for it. However the UML-RSDS constraint language is closer to OCL than is Alloy, and Alloy retains explicit quantifiers and set comprehension, so requiring a greater level of mathematical experience to use for specification than UML-RSDS. Our analysis approach is based on using established formal methods tools (B and SMV) instead of developing a new tool.

UML-RSDS is similar in intent to pragmatic formal approaches such as SCR [3]. However, instead of inventing a new diagrammatic formal method, we make precise an existing widely-used semi-formal method.

Conclusion

We have shown that a simplified form of OCL can be used for practical formal specification of reactive systems, and that a translation of UML class diagrams to B can be performed together with synthesis of B operation code from constraints.

References

1. J-R. Abrial, *The B Method*, Cambridge University Press, 1996.
2. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hahnle, W. Menzel, and P. H. Schmitt. *The KeY approach: Integrating object oriented design and formal verification.* Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, Jan. 2000.
3. R. Bharadwaj, C. Heitmeyer, *Model checking complete requirements specifications using abstraction.* In *Proceedings of Automated Software Engineering*, 6, 37–68, 1999.
4. J R Burch, E M Clarke, K L McMillan, D L Dill, J Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, Proceedings of the Fifth Annual Symposium on Logic in Computer Science, 1990.
5. CS-RR Inc., *CS-RR Software User Requirements Document*, 1994.
6. Daniel Jackson, *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, Software Design Group, MIT Lab for Computer Science, 2002.
7. S. Kim, D. Carrington, *A Formal Mapping Between UML Models and Object-Z Specifications*, in ZB2000, LNCS Vol. 1878, Springer-Verlag, 2000.
8. K. Lano, D. Clark, K. Androutsopoulos, *Safety and Security Analysis of Object-oriented Models*, Safecomp 2002.
9. K. Lano, H. Haughton, *Specification in B*, Imperial College Press, 1996.
10. K. Lano, J. Fiadeiro, L. Andrade, *Software Design in Java 2*, Palgrave, 2002.
11. K. Lano, D. Clark, K. Androutsopoulos, *Formal Specification and Verification of Railway Systems using UML*, FORMS 2003.
12. K. Lano, D. Clark, K. Androutsopoulos, *Synthesis of Code from UML Specifications*, DCS, King's College, 2003.
13. K. Lano, D. Clark, K. Androutsopoulos, *Extended Axiomatic Semantics of UML Class Diagrams and Statecharts*, DCS, King's College, 2003.
14. OMG, *UML Version 1.5 Specification*, http://www.omg.org/uml/, 2003.
15. OMG, *Response to UML 2.0 OCL RfP*, OMG Document ad/2003-01-07, 2003.
16. OMG, *Model-Driven Architecture*, http://www.omg.org/mda/, 2003.
17. Richters, M. *A UML-based Specification Environment*, http://www.db.informatik.uni-bremen.de/projects/USE, 2001.
18. G. Smith, *The Object-Z Specification Language*, Kluwer, 2000.
19. C. Snook, P. Wheeler, M. Butler, *Preliminary Tool Extensions for Integration of UML and B*, IST-2000-30103 deliverable D4.1.2, 2003.
20. H. Treharne, *Supplementing a UML Development Process with B*, FME '02.
21. J. Warmer, A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1999.
22. P. Zeppo, *From UML to B Specifications*, MSc thesis, Dept. of Computer Science, King's College London, 2002.