We consider here a simple system (S0). Its state is represented by two variables xx and input. xx is linked with input such as, when input evolves (event ev1), then xx is likely to evolve too (events ev2 to ev7). We can consider that ev1 represents a kind a stimulus (from the environment) and ev2 to ev7 represent the response to this stimulus. Modalities can be used to write invariant properties restricted to some events (*i.e,*. ev2, ev3, … ev7). In this case, we would like to verify that the response complies with some general law, exhibited in the ESTABLISH clause.

```
SYSTEM
   S0
SETS
   EE = {e0, e1, e2, e3, e4};
   INPUTS = {v1, v2, v3}
VARIABLES
   xx,
   input
INVARIANT
   xx ∈ EE ∧
   input ∈ INPUTS
INITIALISATION
   xx := e0 ||
   input :∈ INPUTS
EVENTS
   ev1 =
      ANY in WHERE in ∈ INPUTS
      THEN
         input := in
      END
   ;
   ev2 = SELECT input=v1 ∧ xx = e0 THEN xx := e2 END;
   ev3 = SELECT input=v1 ∧ xx ≠ e0 THEN xx := e0 END;
   ev4 = SELECT input=v2 ∧ xx = e0 THEN xx := e2 END;
   ev5 = SELECT input=v2 ∧ xx ≠ e0 THEN xx := e3 END;
   ev6 = SELECT input=v3 ∧ xx = e1 THEN xx := e4 END;
   ev7 = SELECT input=v3 ∧ xx ≠ e1 THEN xx := e1 END
MODALITIES
   BEGIN
      ev2, ev3, ev4, ev5, ev6, ev7
   ESTABLISH
    input ∑ xx ∈ {
      (v1 ∑ e0), (v1 ∑ e2), (v1 ∑ e3),
      (v2 ∑ e2), (v2 ∑ e3),
      (v3 ∑ e1), (v3 ∑ e3), (v3 ∑ e4)
      }
   END

END
```

2.3 UML+B Notation

The U2B tool is a prototype tool to convert adapted forms of UML class diagrams and state chart diagrams into specifications in the B language. The aim is to use some of the features of UML diagrams to make the process of writing formal specifications easier, or at least more to the

average programmer. The translation relies on the precise expression of additional behavioural constraints in the specification of class diagram components and in state charts attached to the classes. These constraints are described in an adapted form of the B abstract machine notation. The type of class diagrams that can be converted is restricted in order to comply with constraints of the B-method without making the resultant B unnatural. The resulting UML model is a precise formal specification but in a form which is more friendly to the average programmer, particularly if they use the same UML notation for their program design work.

2.3.1 U2B Class Diagram Translator

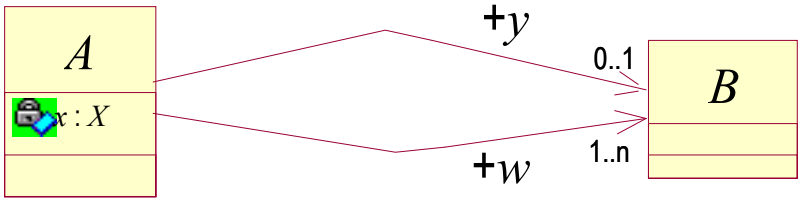
The U2B translator converts Rational Rose[5] UML Class diagrams [Rational00a], including attached state charts, into the B notation. U2B is a script file that runs within Rational Rose and converts the currently open model to B. It is written in the Rational Rose Scripting language, which is an extended version of the Summit Basic Script language [Rational00a, Rational00b]. U2B is configured as a menu option in Rose. U2B uses the object-oriented libraries of the Rose Extensibility Interface to extract information about the classes in the logical diagram of the currently open model. The object model representation of the UML diagram means that information is easily retrieved and the program structure can be based around the logical information in the class rather than a particular textual format. U2B uses Microsoft Word[6] to generate the B Machine files.

2.3.1.1 Translation of Structure and Static Properties

The translation of classes, attributes and operations is derived from proposals for converting OMT (Object Modelling Technique) to B [Meyer99]. However, since our aims are primarily to assist in the creation of a B specification rather than to generate a formal equivalent of a UML specification, our translation simplifies that proposed by [Meyer99]. This is achieved by restricting the translation to a suitable subset of UML models.

A separate machine is created for each class and this contains a set of all possible instances of the class and a variable that represents the subset of current instances of the class. Attributes and (unidirectional) associations are translated into variables whose type is defined as a function from the current instance to the attribute type (as defined in the class diagram) or associated class.

For example consider the following class diagram with classes *A* and *B*, where *A* has an attribute *x* and there is a unidirectional association from *A* to *B* with role *y* and 0 or 1 multiplicity at the target end. A second association, *w,* has a 'many' multiplicity:



[5]Rational Rose is a trademark of the Rational Software Corporation

[6]Microsoft Word97 is a trademark of the Microsoft Corporation.

This will result in the following machine representing all instances of *A*:

```
MACHINE    A
EXTENDS
        B
SETS
        ASET
VARIABLES
        Ainstances,
        x,
        w,
        y
INVARIANT
        Ainstances ⊆ ASET ⇸
        x ∈ Ainstances → X ⇸
        w ∈ Ainstances → POW1(Binstances) ⇸
        y ∈ Ainstances ⇓ Binstances
INITIALISATION
        Ainstances := ∅ ||
        x := ∅ ||
        w := ∅ ||
        y := ∅
```

Note that the multiplicity of the association *w* is handled as a function from instances of class *A* to sets of instances of class *B* using the power set operator (`POW` and `POW1`). The machine is initialised with no instances and hence all attribute and association functions are empty. A separate machine will be generated for class *B*.

Association multiplicities In UML, multiplicity ranges constrain associations. The multiplicities are equivalent to the usual mathematical categorisations of functions: partial, total, injective, surjective and their combinations. Note that the multiplicity at the target end of the association (class *B* in the example above) specifies the number of instances of *B* that instances of the source end, class *A,* can map to. This can be confusing when thinking in terms of functions because the constraint is at the opposite end of the association to the set it is constraining. The multiplicity of an association determines its modelling as shown in Table 2. We use functions to sets of the target class instances (*e.g.*, `POW(B)`) to avoid non-functions. Note that `0..n` is assumed unless otherwise specified in UML.

Association Representations in B for Different Multiplicities		
Ai *and* ***Bi*** *are the current instances sets of class A and B respectively (i.e., Ainstances and Binstances) and* ***f*** *is a function representing the association (i.e., the role name of the association with respect to the source class, A).*		
UML association multiplicity	**Informal description of B representation**	**B invariant**
`0..n` → `0..1`	partial function to *Bi*	$Ai \Downarrow Bi$
`0..n` → `1..1`	total function to *Bi*	$Ai \rightarrow Bi$
`0..n` → `0..n`	total function to subsets of *Bi*	$Ai \rightarrow \mathtt{POW}(Bi)$
`0..n` → `1..n`	total function to non-empty subsets of *Bi*	$Ai \rightarrow \mathtt{POW1}(Bi)$
`0..1` → `0..1`	partial injection to *Bi*	$Ai \rightarrowtail Bi$
`0..1` → `1..1`	total injection to *Bi*	$Ai \odot Bi$
`0..1` → `0..n`	total function to subsets of *Bi* which do not intersect	$Ai \rightarrow \mathtt{POW}(Bi) \twoheadrightarrow \mathtt{inter}(\mathtt{ran}(f)) = \varnothing$
`0..1` → `1..n`	total function to non-empty subsets of *Bi* which do not intersect	$Ai \rightarrow \mathtt{POW1}(Bi) \twoheadrightarrow \mathtt{inter}(\mathtt{ran}(f)) = \varnothing$
`1..n` → `0..1`	partial surjection to *Bi*	Ai □ Bi
`1..n` → `1..1`	total surjection to *Bi*	Ai □ Bi
`1..n` → `0..n`	total function to subsets of *Bi* which cover *Bi*	$Ai \rightarrow \mathtt{POW}(Bi) \twoheadrightarrow \mathtt{union}(\mathtt{ran}(f)) = Bi$
`1..n` → `1..n`	total function to non-empty subsets of *Bi* which cover *Bi*	$Ai \rightarrow \mathtt{POW1}(Bi) \twoheadrightarrow \mathtt{union}(\mathtt{ran}(f)) = Bi$
`1..1` → `0..1`	partial bijection to *Bi*	Ai □ Bi
`1..1` → `1..1`	total bijection to *Bi*	Ai □ Bi
`1..1` → `0..n`	total function to subsets of *Bi* which cover *Bi* without intersecting	$Ai \rightarrow \mathtt{POW}(Bi) \twoheadrightarrow \mathtt{union}(\mathtt{ran}(f)) = Bi \twoheadrightarrow \mathtt{inter}(\mathtt{ran}(f)) = \varnothing$
`1..1` → `1..n`	total function to non-empty subsets of *Bi* which cover *Bi* without intersecting	$Ai \rightarrow \mathtt{POW1}(Bi) \twoheadrightarrow \mathtt{union}(\mathtt{ran}(f)) = Bi \twoheadrightarrow \mathtt{inter}(\mathtt{ran}(f)) = \varnothing$

Table 2: How associations are represented in B for each possible multiplicity constraint.

In Figure 2.10 a mapping represents an association between the classes *A* and *B* with multiplicity `0..n → 0..1`. The representation in the B notation is a partial function. It is not a total function because a_4 doesn't map to anything in B (as indicated by the 0 at the right hand end of `0..n → 0..1`). It is not injective because b_2 is mapped to by both a_2 and a_3 (as indicated by the n at the left hand end of `0..n → 0..1`). It is not surjective because b_3 is not mapped to by anything in *A* (as indicated by the `0` at the left hand end of `0..n → 0..1`).

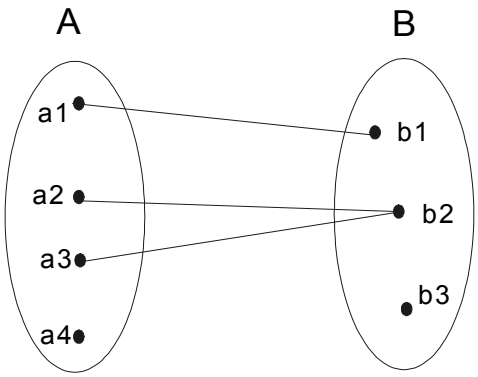
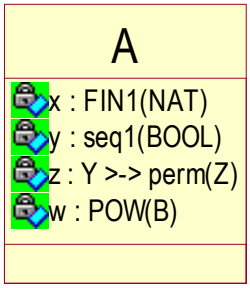
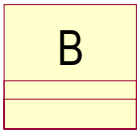


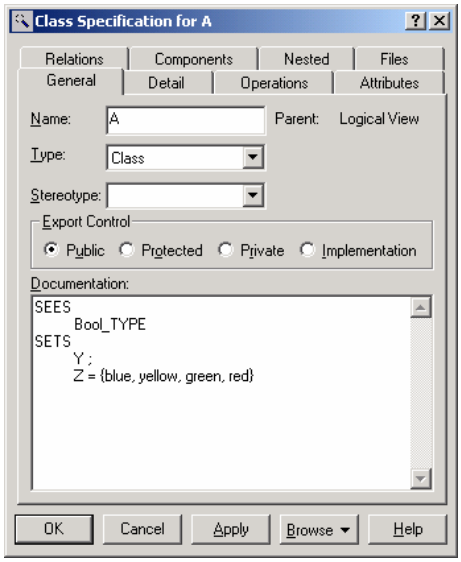
Figure 2.10: Mapping representing a `0..n → 0..1` association.

Attribute types Attribute types may be any valid B expression that defines a set. This includes predefined types such as `NAT`, `NAT1`, `boolean` and `string` (if translating to B-Core B, the appropriate B library machines must be referenced via a `SEES` clause in the class's specification documentation window) functions, sequences, power sets, instances of another class (referenced by the class name) or enumerated or deferred sets defined in the class specification documentation window. If the type involves another class (and there is no unidirectional path of associations to that class) the machine for that class will be referenced in a `USES` clause so that its current instances set can be read. If there is a path of unidirectional associations to the class it will be extended (`EXTENDS`) by this machine in order to represent the association and this will provide access to the instances set. (Note that only unidirectional associations are interpreted as associations. Unspecified or bi-directional associations are ignored and can therefore be used to indicate type dependencies diagrammatically if required). Any references to the class in type definitions of variables or operation arguments will be changed to the current instances set for that class.

For example, the following shows a class that has an attribute `x` of type, non-empty finite subset of natural numbers. It has an attribute `y` that is of type, non-empty sequence of booleans. The library machine `Bool_TYPE` has been referenced via a `SEES` clause in the class's documentation window (this would not be necessary for Atelier-B). It has an attribute `z` that has type, total injection from `Y` to permutations of `Z`. A `SETS` clause has been added to the class's documentation window that defines `Y` as a deferred set and `Z` as an enumerated set.



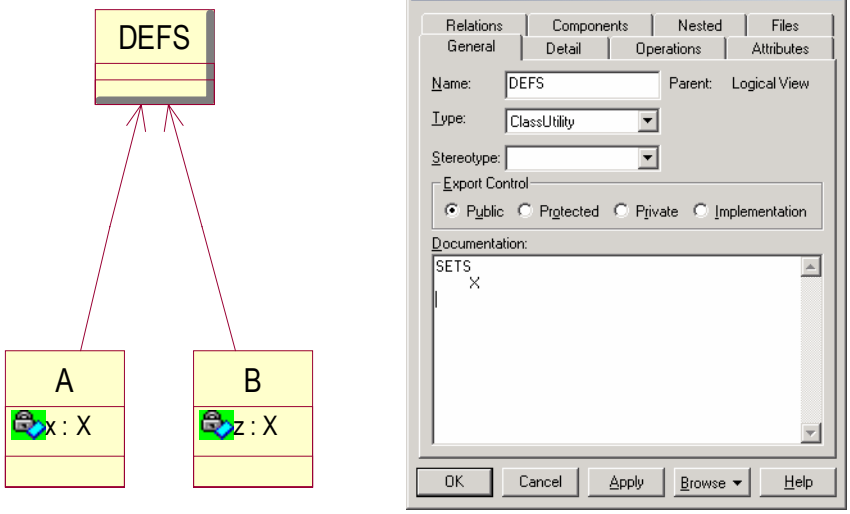




Note that `Export Control` settings in the class specification are not used in the U2B translation. The corresponding B machine for class *A* is shown below.

```
MACHINE   A
SEES
       Bool_TYPE
USES
       B
SETS
       ASET ;
       Y ;
       Z = {blue, yellow, green, red}
VARIABLES
       Ainstances,
       x,
       y,
       z,
       w
INVARIANT
       Ainstances ⊆ ASET ⇸
       x ∈ Ainstances → FIN1(NAT) ⇸
       y ∈ Ainstances → seq1(BOOL) ⇸
       z ∈ Ainstances → Y ⊙ perm(Z) ⇸
       w ∈ Ainstances → POW(Binstances)
```

Global Definitions It is often useful to define types as enumerated or deferred sets for use in many machines. We use *class utilities* for this. In UML, a class utility is a class that has no instances, only static (class-wide) operations and attributes. The U2B translator creates a machine for each class utility and copies any text in the specification documentation window of its class specification into the machine. Hence definitions, sets and constants can be described in B clauses in the documentation window. Any machines that reference items defined in this way must have an association to the class utility. This association will not be interpreted as an association to an ordinary class). In the following example a class utility `DEFS` is used to define a set `X` that is used as a type by two other classes.



The corresponding machine for class utility `DEFS` is:

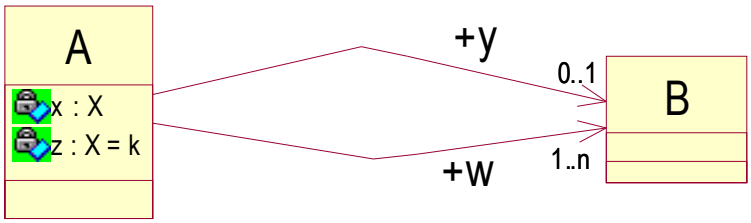
```
MACHINE    DEFS
SETS
          X
END
```

The machines for classes *A* and *B* will reference `DEFS` via a `SEES` clause:

```
SEES
        DEFS
```

Local Definitions Sets can also be defined locally to a class in the class's specification documentation window. In fact, any valid B clause can be added in this window. For example, we use this method to specify invariants for the class. Each clause must be headed by its B clause name in capitals and starting at the beginning of a line, the text that follows that clause, up until the next clause title (if any) will be added to the appropriate clause in the machine. Any text before the first clause is treated as comment and added as such at the top of the machine

Instance Creation and Initialisation of Attributes and Associations. A create operation is automatically provided for each class machine so that new instances can be created. This picks any instance that is not already in use, adds it to the current instances set, and adds a maplet to each of the attribute/association relations mapping the new instance to the appropriate initial value. Note that, according to our definition (via translation) of class diagrams, association means that the source class is able to invoke the methods of the target class. The example below is similar to the first example but class *A* has an additional attribute *z*, that has an initial value *k*.




```
Return ← Acreate =
    PRE
        Ainstances ≠ ASET
    THEN
        ANY new
        WHERE
            new ∈ ASET - Ainstances
        THEN
            Ainstances := Ainstances ∪ {new} ||
            x(new):∈ X ||
            z(new):= k ||
            w(new):∈ POW1(Binstances) ||
            Return := new
        END
    END
```

Note that, because `x` has no initial value specified, it is initialised non-deterministically to any value of the type `X` (`:∈` means any value belonging to). Similarly the association *w* must be initialised to a non empty set because its multiplicity may be greater than one but is definitely greater than zero. (Currently, we have no means of specifying initial values for associations). It is initialised non-deterministically to any non-empty subset of B instances. The association *y* is not initialised because its multiplicity is 0 or 1 and it may, therefore, be undefined. Initially the new instance will have no maplet representing the association *y*.

Singular Classes Often, a B machine models a single generic instance of an entity, rather than an explicit set of instances (in the same way that a class in UML leaves instance referencing implicit). The resulting specification is simpler and clearer for not modelling multiple instances. The U2B translator creates a single-instance machine if the class multiplicity (cardinality) is set to `1..1` in the UML class specification. Note that this can only be done at the top level of a structure since at lower levels the instance set is used for referencing by the higher level. Below is shown the machine representing class *A* from the first example above if the class's multiplicity is set to `1..1`. Note that there is no modelling of instances, the type of attributes is simpler because it is no longer necessary to map from instances to the attribute type. There is no instance creation operation; attributes are initialised in the machine initialisation clause.

```
MACHINE   A
EXTENDS
       B
VARIABLES
       x,
       w,
       y
INVARIANT
       x ∈ X ∧
       w ∈ POW1(Binstances) ∧
       y ∈ Binstances
INITIALISATION
       X :∈ X ||
       w :∈ POW1(Binstances) ||
       y := ∅
END
```

Restrictions The B method imposes some restrictions on the way machines can be composed. These restrictions ensure compositionality of proof. Their impact is that no write sharing is allowed at machine level (*i.e.,* a machine may only be included or extended by one other machine). Also, the inclusion mechanism of B is hierarchical so that, if `M1` includes `M2`, then `M2` cannot, directly or transitively, include `M1`. We reflect these restrictions in the UML form of the specification, which must therefore be tree like in terms of unidirectionally related classes. Non-navigable (and bi-directional) associations are ignored but may be used to illustrate the use of another class as a type (*i.e.,* read access only). However, multiple, parallel associations between the same pair of classes are permitted.

Although we would like to adhere to the UML class diagram rules as much as possible, since our aim is to make B specification more approachable rather than to formalise the UML we are relatively happy to impose restrictions on the diagrams that can be drawn. That is, we only define translations for a subset of UML class diagrams. Other authors [Facon96, Meyer99, Meyer00, Nagui94, Shore96] have suggested ways of dealing with the translation of more general forms of class diagrams. However, the structures of B machines that result from these more general translations can be cumbersome. If the specification were written directly in B, it would be highly unlikely that the resulting B would have this form. Since we also desire a usable B specification we prefer to restrict the types of diagrams that can be drawn.

2.3.1.2 Dynamic Behaviour

The dynamic behaviour modelled on a class diagram that is converted to B by U2B is embodied in the behaviour specification of classes operations and in invariants specified for the classes. UML does not impose any particular notation for these operation and invariant constraint definitions; they could be described in natural language or using UML's Object Constraint Language (OCL). However since we wish to end up with a B specification it makes sense to use bits of B notation to specify these constraints. The constraints are specified in a notation that is close to B notation but has to observe a few conventions in order for it to become valid B within the context of the machine produced by U2B. When writing these portions of B the writer should not need to consider how the translation would represent the features (associations, attributes and operations) of the classes. Also we felt we should follow the more object-oriented conventions of implicit self-referencing and use of the dot notation for explicit instance references. Therefore,

when writing the constraints, a dot notation is used to reference the ownership of features. This is illustrated in examples below.

Invariant Unfortunately there is no dedicated text box for a class invariant in Rational Rose. One suggestion is to put invariant constraints in a note attached to the class [Warmer99], but notes are treated as an annotation on a particular view in Rational Rose and not part of the model. This makes them difficult to access from the translation program and unreliable should we extend the conversion to look at other views. Therefore we include the invariants as a clause in the documentation text box of the class' specification window. The invariants are generally of two kinds, *instance invariants* (describing properties that hold between the attributes and relationships within a single instance) and *class invariants* (describing properties that hold between different instances). To deal with instance invariants, and keeping with the implicit self-reference style of UML, we chose to allow the explicit reference to this instance to be omitted. U2B will add the universal quantification over all instances of the class automatically. For class invariants, the quantification over instances is an integral part of the property and must be given explicitly. Hence, U2B will not need to add instance references.

For example, if `bx ∈ NAT` is an attribute of class *B* then the following invariant could be defined in the documentation box for class *B*:

```
bx < 100 ∧
∀(b1,b2).((b1 ∈ B ∧ b2 ∈ B ∧ b1 ≠ b2) => (b1.bx ≠ b2.bx)
```

This would be translated to:

```
∀(thisB).(thisB ∈ Binstances  ⇒
   bx(thisB) < 100 ∧
   ∀(b1,b2).((b1 ∈ Binstances ∧ b2 ∈ Binstances ∧ b1 ≠ b2)
⇒ (bx(b1) ≠ bx(b2))
)
```

The translation has added a universal quantification, `thisB`, over all instances of *B* and this is used in the first part of the invariant. It is not used in the second part where the invariant already references instances of class *B*. (Note that currently the translator adds one universal quantification for the entire invariant whether or not it is needed).

Operation Semantics Operation preconditions are specified in a textual format attached to the operation within the class. Details of operation behaviour are specified either in a textual format attached to the operation, or in a state chart attached to the class. Operation behaviour may be specified completely by textual annotation, completely by state chart transitions, or by a combination of both composed as simultaneous specification.

Operation Textual behaviour specification In Rational Rose, *Specifications* are provided for operations (as well as many other elements) and these provide text boxes dedicated to writing pre-conditions and semantics for the operation. Although Rational Rose also provides a *post-*

condition text box , this is not currently used, as the *semantics* box suit the pseudo-operational style of B better.

Operations need to know which instance of the class they are to work on. This is implicit in the class diagram. The translation adds a parameter *thisCLASS* of type *CLASSinstances* to each operation. This is used as the instance parameter in each reference to an attribute or association of the class.

In the above example, *set_y* might have the following precondition:

```
i > y.bx
```

and semantics

```
y.b_op(i) ||
IF y.bx <100
THEN
    out := FALSE
ELSE
    out := TRUE
END
```

which would be translated to

```
i > bx(y(thisA))
```

and

```
b_op(y(thisA)) ||
IF bx(y(thisA)) <100
THEN
    out := FALSE
ELSE
    out := TRUE
END
```

Operation Return Type UML operation signatures contain a provision for specifying the type for a value returned by the operation. Since B infers this from the body of the operation we use it instead to name the identifiers that represent operation return values. The string entered in the return type field for the operation will be used as the operation return signature in the B machine representing the class. For example, the *set_y* operation in the above class diagram has its return field set to *out*. The operation signature for `set_y` in the B machine `A` will be :

```
out <--  set_y (thisA,i) =
```

State chart Behavioural Specification For classes that have a strong concept of state change, a state chart representation of behaviour is appropriate. In UML a state chart model can be attached to a class to describe its behaviour. A state chart model consists of a set of states and transitions that represent the state changes that are allowed. If a state chart model is attached to a class the U2B translator combines the behaviour it describes with any operation semantics described in the operation specification semantics windows. Hence operation behaviour can be defined either in the *operation semantics* window or in a state chart model for the class or in a combination of both.

The name of the state chart model is used to define a state variable. (Note that this is not the name of a state chart diagram, several diagrams could be used to draw the state chart model of a class). The collection of states in the state chart model is used to define an enumerated set that is used in the type invariant of the state variable. The state variable is equivalent to an attribute of the class and may be referenced elsewhere in the class and by other classes. State chart transitions define which operation call causes the state variable to change from the source state to the target state, *i.e.*, an operation is only allowed when the state variable equals a state from which there is a transition associated with that operation. To associate a transition with an operation, the transition's name must be given the same name as the operation. Additional guard conditions can be attached to a transition to further constrain when it can take place. All transitions cause the implicit action of changing the state variable from the source state to the target state. (The source and target state may be the same). Additional actions (defined in B) can also be attached to transitions. The translator finds all transitions associated with an operation and compiles a `SELECT` substitution of the following form:

```
SELECT statevar=sourcestate1 ∧ sourcestate1_guards
THEN statevar:=targetstate1 || targetstate1_actions
WHEN statevar=sourcestate2 ∧ sourcestate2_guards
THEN statevar:=targetstate2 || targetstate2_actions
<etc>
END ||
```

This is composed with the operation pre-condition and body (if any) from the textual specification in the operation's `pre-condition` and `semantics` windows:

Let `Popw` be the precondition in the operation `precondition` window, `Sosw` be the operation body from the `operation semantics` window and `Gstc` the `SELECT` substitution for this operation composed from the state chart. Then the translator will produce the following operation:

```
PRE
   Popw
THEN
   Gstc ||
   Sosw
END
```

This can be represented more succinctly in B as:

```
Popw  |  (Gstc || Sosw)
```

Hence the precondition, `Popw`, has precedence and, if false, the operation will abort. If an event B style systems simulation is desired, the specifier should take care not to define pre-conditions that conflict with the transition guards. (For example, if an event only occurs if an attribute `bx` is

positive, and this is modelled by a guarded transition; adding the pre-condition `bx > 0` would change the meaning of the model to represent a system where if the event occurs the operation aborts).

Note that it would be entirely valid (although somewhat obtuse) to write a pre-condition within the *operation semantics* window: `Sosw = Posw | S1osw`. However, preconditions take precedence in simultaneous substitutions, so

```
(Gstc || (Posw | S1osw))  =  Posw | (Gstc || S1osw).
```

Hence, writing the precondition in the `operation semantics` window is equivalent to writing it in the `precondition` window. It has the same precedence and possible conflicts with the operation guards derived from the state chart. We feel that writing the precondition in the `operation semantics` window should be discouraged because the precedence may not be obvious to readers of the specification.

If the pre-condition (`Popw ∧ Posw`) is true, then the guard from `Gstc` takes precedence over the simultaneous substitution `Sosw`. This means that the textual operation body from the operation semantics window, although defined separately from the state chart and not associated with any particular state transition, is only enabled when at least one of the state transitions is enabled. That is, if

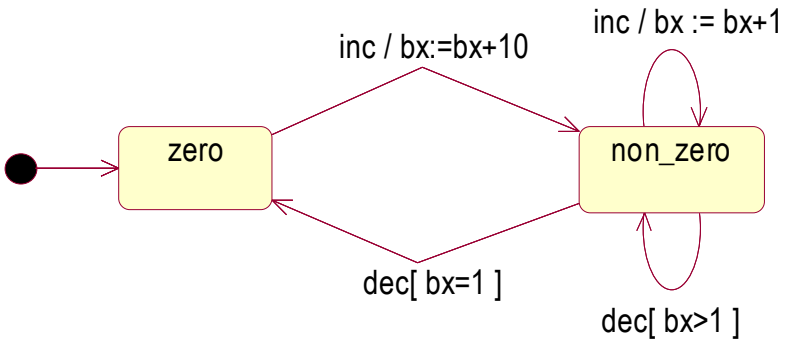
`Gstc` = (G1 ⇒ T1) □ .... □ (Gn ⇒ Tn)

then,

`(Gstc || Sosw) = ( G1 ⇒ ( T1 || Sosw ) ) □ … □ ( G ⇒ ( Tn || Sosw ) )`

where □ represents choice.

Actions should be specified on state transitions when the action is specific to that state transition. Where the action is the same for all that operation's state transitions, it may be specified in the operation semantics window in order to avoid repetition. The chart below illustrates how a state chart can be used to guard operations and define their actions. It also shows how common actions can be defined in the operation semantics window and how a pre-condition could upset the constraints imposed by the state chart.



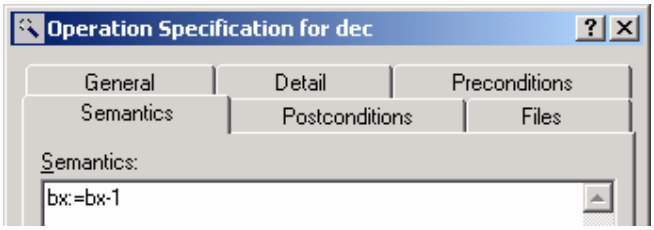
The state chart has two states, *zero* and *non_zero*. The implicit state variable, `b_state` (the name of the state chart model) is treated like an attribute of type `B_STATE = {zero, non_zero}`. An invariant defines the correspondence between the value of the attribute *bx* and the state *zero*. When an instance is created its `b_state` is initialised to `zero` because there is a transition from an initial state to *zero*.

```
MACHINE   B
SETS
       BSET;
       B_STATE={zero,non_zero}
VARIABLES
       Binstances,
       b_state,
       bx
INVARIANT
       Binstances ⊆ BSET ∧
       b_state ∈ Binstances → B_STATE ∧
       bx ∈ Binstances → NAT ∧
       ∀(thisB).(thisB ∈ Binstances ⇒
        (b_state(thisB)=zero) ⇔ (bx(thisB)=0)
        )
INITIALISATION
       Binstances := ∅ ||
       b_state := ∅ ||
       bx := ∅
OPERATIONS
Return  ← Bcreate =
   PRE
       Binstances ≠ BSET
   THEN
       ANY new
       WHERE
          new ∈ BSET - Binstances
       THEN
          Binstances := Binstances ∪ {new } ||
          b_state(new):=zero ||
          bx(new) :∈ NAT ||
          Return := new
       END
   END
;
```

Operation `inc` can occur in either state. Its action is different depending on the starting state and so actions have been defined on the transitions and are combined with the change state action.

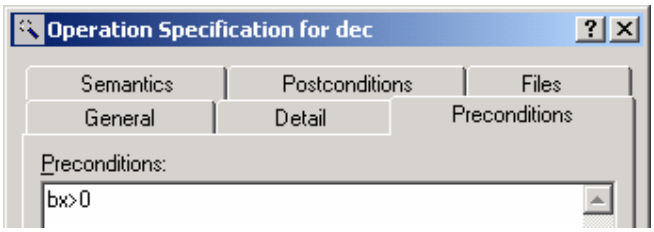
```
inc (thisB) =
   PRE
       thisB ∈ Binstances
   THEN
       SELECT b_state(thisB)=zero
       THEN   b_state(thisB) := non_zero ||
              bx(thisB):=bx(thisB)+10
       WHEN   b_state(thisB)=non_zero
       THEN   bx(thisB) := bx(thisB)+1
       END
   END
```

Operation `dec` has two guarded alternatives when in state `non_zero` but does not occur while in state `zero`. Since the action is the same for both transitions it has been defined in the *operation semantics* window.




```
dec (thisB) =
    PRE
        thisB ∈ Binstances
    THEN
        SELECT b_state(thisB)=non_zero ∧
               bx(thisB)=1
        THEN   b_state(thisB):=zero
        WHEN   b_state(thisB)=non_zero ∧
               bx(thisB)>1
        THEN   skip
        END ||
        bx(thisB):=bx(thisB)-1
    END
END
```

If we had put the pre-condition $bx > 0$ in the *operation specification precondition* window (or even in the *operation semantics* window), the guard would no longer function since the precondition would fail resulting in an abort when $bx = 0$.




```
dec (thisB) =
    PRE
        thisB ∈ Binstances ∧
        bx(thisB)>0
    THEN
        SELECT b_state(thisB)=non_zero ∧
               bx(thisB)=1
        THEN   b_state(thisB):=zero
        WHEN   b_state(thisB)=non_zero ∧
               bx(thisB)>1
        THEN   skip
        END ||
        bx(thisB):=bx(thisB)-1
    END
```

This could be avoided by repeating the precondition and decrement substitution in the action field of each *dec* transition on the state chart in which case the guard would take precedence.