6SENG005W Formal Methods

Week 9

Structuring B Specifications

Overview of Week 9 Lecture: Structuring B Specifications

The aim of this lecture is to outline *how to structure a B specification using several B machines*:

- The reason for & types of techniques used to *structure* a multi-machine B specification.
- The types of machine *clauses* that can be used to Structure B Machines:
  - `INCLUDES`
  - `PROMOTES`
  - `EXTENDS`
  - `SEES`
  - `USES`
- Two examples of Structuring B specifications using *multiple B machines*:
  - A Bank's Safes Specification using 4 B machines:
    `Doors`, **`Keys`**, **`Locks`** & **`Safes`**.
  - A Births, Deaths & Marriages Registrar Specification using 3 B machines:
    `Life`, **`Marriage`** & **`Registrar`**.

PART I

Introduction to B Specifications Composed of Several B Machines

Complex System States

In almost all systems the state of the system consists of many *components* or *parts*, i.e. *sub-states*, *sub-sub-states*, etc; as depicted in Figure 9.1.

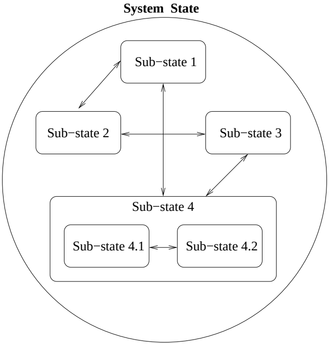

Figure : 9.1 Complex System State

If this is the case then it is good *software engineering practice* to use *modularisation* to decompose it into a number of *sub-systems* each representing one of the state's components.

B Specifications & Complex System States

If we apply good software engineering practice when designing a B specification for a system with a complex state (Figure 9.1), then we need to model the system state as a *collection of B machines*.

This is achieved in the B-Method by modelling each component (sub-state) of the system by a different *B machine*, see Figure 9.2.

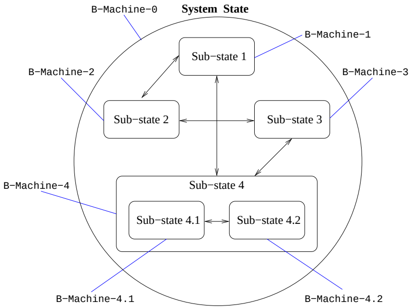

Figure : 9.2 Complex System State & Multiple B Machines

`B-Machine-0` models the *System State*, `B-Machine-1` models *Sub-state 1*, etc.

Advantages of using Structure Techniques

Modularity & *Abstraction* are the principal weapons in the battle against complexity because they provide:

Modularity:

- Separation of concerns (into different state components)
- Separation of proof obligations
- Aids understanding
- Allows module reuse

Abstraction:

- Information hiding
- Separation of specification from implementation concerns
- Reasoning at appropriate level

These are standard approaches used throughout *Software Engineering*.

For example, in Object Oriented languages like Java, C++, C♯; where programs are structured using collections of classes that are linked together via sub-classing, etc.

Multiple B Machine Specifications

In this lecture we will cover the facilities that the B-Method provides for achieving this *"structuring"* of B Specifications.

In general a B *specification* is a collection of *Abstract Machines (AM)*.

B allows software developers to specify a complex system by:

> *defining a collection of abstract B machines that can be related to & interact with each other in various ways.*

This *collection of B machines* are able to *interact* in various ways, such as one B machine can:

- *control* another B machine by – directly accessing its data & use its operations to modify its state;
- *use* another B machine by – directly accessing its data & using its enquiry operations, but is **not** allowed to use its operations to modify its state variables.

Structuring Specifications: Combining Machines

A B specification *"component"* can be an: *abstract machine*, a *refinement* or an *implementation*.

Components possess *"clauses"* incorporating the static (data) & dynamic (operations) description of behaviour.

There are two main ways in which one machine can use another machine:

- One way is to use it as a *totally controlled machine*, being able to change its state:
  - *Including machines*, using the `INCLUDES` clause.
  - *Promoting operations* of an included machine, using the `PROMOTES` clause.
  - *Extending machines*, using the `EXTENDS` clause.
- The other is to use it to answer queries & perform functions, but without being able to change its state:
  - *Using machines*, using the `USES` clause.
  - *Seeing machines*, using the `SEES` clause.

PART II

Structuring B Specifications using: INCLUDES, PROMOTES *&* EXTENDS

B Machine "Structuring Clauses" (I)

The *structuring clauses* that allow one B machine to use another machine *with control over it* are:

`INCLUDES`: when an abstract machine *includes* another abstract machine it integrates the data of the included machine & can use its operations, but **does not** make its *operations* part of its *interface*, i.e they are **not "visible"**.

`PROMOTES`: when an abstract machine *includes* another abstract machine it integrates the data of the included machine.
And can use the `PROMOTES` clause to **selectively** add any of its *operations* to its interface, by *promoting them*, i.e. making them *"visible"*.

`EXTENDS`: when an abstract machine *extends* another abstract machine it integrates the data of the included machine & makes *all of its operations part of its interface*, i.e. *"visible"*.

Accessible Components of a B Machine

If B machine `M2` uses one of the structuring clauses: `INCLUDES`, `EXTENDS`, `SEES` or `USES`, on another B machine `M1`.

Then `M2` can *access* &/or *use* some or all of the following component clauses of machine `M1`.

`SETS` & `CONSTANTS`	declaration of the machine's sets & constants.
`PROPERTIES`	declaration of the *types* & *properties* of the sets & constants.
`VARIABLES`	declaration of the state variables.
`INVARIANT`	declaration of invariant properties of the variables
`OPERATIONS`	declaration of the operations that change its state & enquiries that do not.

Which of the specific components listed above can be accessed & how depends on which of the structuring clauses is used.

For example, `INCLUDES` has the most access & control, whereas `SEES` has the least.

The INCLUDES Clause

As an example consider one machine including another:

```
M2 INCLUDES M1
```

this is illustrated by the Structure diagram in Figure 9.3.

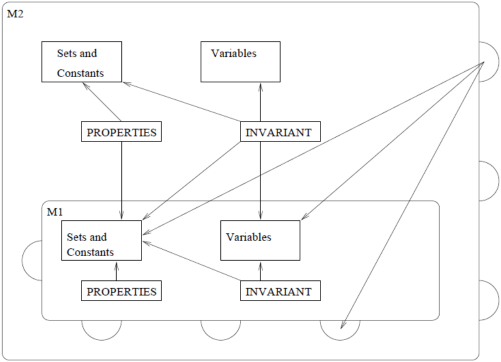

Figure : 9.3 Structure Diagram of M2 INCLUDES M1

Properties of: `M2 INCLUDES M1`

`M1` is *defined independently* of any machines (`M2`) that *include* it, via an `INCLUDES` clause.

So `M1` *cannot refer to any of the information contained in* `M2`.

`M2`'s *invariant* & *operations* have access to the state & operations of `M1`.

`M2` can *change the state of* `M1` *but only through* `M1`'s *own operations*.

In particular, `M2` **cannot** make an explicit assignment to one of `M1`'s variables in one of its own operations, e.g. the following is **illegal**:

```
M2_operation = BEGIN
                    M1_var := M2_var + 10    /* ILLEGAL */
               END
```

Restriction on the use of `INCLUDES`:

A machine may be included in AT MOST ONE MACHINE.

This prevents machines from violating each other's invariant.

See the **`Safes`** example where the **`Safes`** machine includes two machines: **`Keys`** & **`Locks`**, (& **`Locks`** includes **`Doors`**).

Machine “Inheritance” & Restrictions

An abstract machine may be structured by use of an `INCLUDES` clause, which lists a number of abstract machines (& supplies their parameters) which are incorporated into the extending machine.

In `M2 INCLUDES M1`:

- The *“native”* variables of `M2` are those defined in `M2`’s `VARIABLES` clause.
- Similarly, the *native* sets & constants are those defines in `M2`’s `SETS` & `CONSTANTS` clauses.
- The *included* variables of `M2` are the native & included variables of `M1`.
- Similarly for sets & constants.

The PROMOTES Clause

In some situations it is desirable to *promote* some operations of an *included machine* to become operations of the *including* machine, but **not** to promote all of them.

The PROMOTES clause lists all of those *operations* which should be promoted to the *interface* of the *including machine*.

Since these operations are now operations of the including machine, they **MUST** *be shown to preserve that machine's invariant*.

For example:

Safes

```
MACHINE Safes

  INCLUDES
    Keys, Locks

  PROMOTES
    opendoor, lockdoor   /* promoted from "Locks" */
    closedoor            /* promoted from "Doors" */
```

PROMOTES Clause

In M2 the use of the following PROMOTES clause results in the Structure diagram given in Figure 9.4.

```
MACHINE M2
  INCLUDES M1
  PROMOTES M1_Op1, M1_Op2    /* But NOT M1_Op3 or M1_Op4 */
```

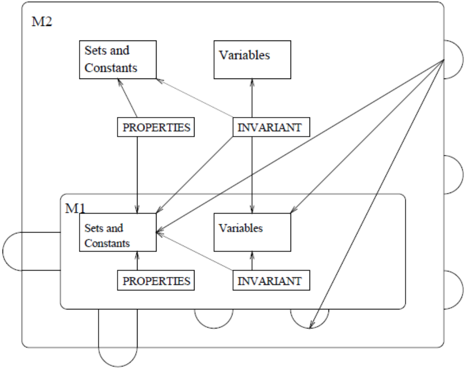

Figure : 9.4 Structure Diagram of M2 PROMOTES M1_Op1, M1_Op2

The `EXTENDS` Clause

`EXTENDS` is a special case of `INCLUDES`.

In `M2` `EXTENDS` `M1`:

- All operations of `M1` are *promoted* to operations of `M2`, i.e. are visible.
- All included variables of `M1` may be *directly read-accessed* in operation definitions of `M2`.
- Included variables of `M1` may be *updated* in `M2` **only** by using `M1`'s *native* operations.
- `M1`'s operations can be used in the definition of an `M2` operation.
- The *invariants* of `M1` becomes part of the *invariant* of `M2`.
- Any clause of `M2`'s invariant (inherited or defined explicitly in `M2`) may refer to included variables.

The EXTENDS Clause

The result of using

```
MACHINE M2
   EXTENDS M1
```

is that *all of the operations* of M1 (M1_Op1 to M1_Op4) are *promoted to operations of* M2, i.e. become part of M2's *interface*.

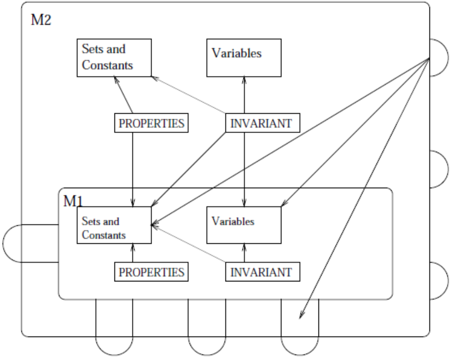

Figure : 9.5 Structure Diagram of M2 EXTENDS M1

Using an Operation of an Included Machine

Operations of a machine that has been included (via `INCLUDES`) may be *used in operation definitions of the including machine*.

When using, i.e. *"calling"*, these operations any *formal parameters* must be replaced with the *actual parameters*.

For example, the **Locks** operation

```
opendoor( dd1 ) =                               /* Locks operation */
  PRE
       dd1 : DOOR  &  status(dd1) = unlocked
  THEN
        opening( dd1 )                          /* Doors operation */
  END
```

uses the following operation from **Doors** using `dd1` for `dd0`:

```
opening( dd0 ) =                                /* Doors operation */
  PRE
        dd0 : DOOR                              /* Precondition */
  THEN
        position( dd0 ) := open
  END
```

Preconditions of Included Operation

The *precondition* of an included operation is included in its expansion.

Hence it will have to be *true* whenever the operation is called in order for consistency to be obtained.

Hence the including machine:

> *Must guarantee that the precondition is true whenever the operation is called.*

For example, in the previous example of **`Locks opendoor`** operation calling **`Doors opening`** operation, it **must ensure** that **`opening`**'s *precondition*:

```
dd0 : DOOR    /* opening's Precondition. */
```

is true, at the point where it is called, with the actual parameter `dd1` replacing the formal parameter `dd0`.

So this must be true:

```
dd1 : DOOR    /* Precondition with parameter dd1. */
```

Problem of Included or Extended Operations
Invalidating a Machine's INVARIANT

See the multiple machine example in the tutorial using 6 machines: M0 to M5.

In particular, the M2 machine:

- M2 *includes* M0 & *promotes* 2 of its operations, &
- M2's *invariant* relates its variable & M0's variable.

```
MACHINE M2
  INCLUDES     M0
  PROMOTES     set_M0_var_equal_M4_var,  M0_modify_M0_var

  INVARIANT    M2_var : NAT  &  M0_var <= M2_var
```

But both of M0's promoted operations **could invalidate** M2's INVARIANT.

For example, the M0 operation:

```
  M0_modify_M0_var = BEGIN
                         M0_var := M0_var + 100
                     END
```

could easily invalidate M2's INVARIANT.

Ensuring Included or Extended Operations **Satisfy** a Machine's `INVARIANT`

So to ensure that `M0`'s `M0_modify_M0_var` operation **does not invalidate** `M2`'s `INVARIANT`: **`M0_var <= M2_var`**

The operation should **not be promoted** via `PROMOTES`, since its uncontrolled execution can invalidate `M2`'s `INVARIANT`.

Solution: *embed it inside* an `M2` operation that ensures that this **cannot** happen by checking that `M2`'s `INVARIANT` will be true *after it is executed*.

M2_SAFE_M0_modify_M0_var

```
report <-- M2_SAFE_M0_modify_M0_var =
   BEGIN
       IF ( M0_var + 100 <= M2_var )   /* M2's INVARIANT */
       THEN
             M0_modify_M0_var ||       /* M0's Operation */
             report := M2_INVARIANT_MAINTAINED
       ELSE
             report := WOULD_INVALIDATE_M2_INVARIANT
       END
   END
```

Summary: `TopLevel` Machine

Top_Level

```
MACHINE  Top_Level

  EXTENDS     Machine_One

  INCLUDES    Machine_Two,  Machine_Three

  PROMOTES
    M_Two_Op_1,                       /* But NOT M_Two_Op2 */
    M_Three_Op_3, M_Three_Op_4        /* But NOT M_Three_Op1 or
                                                 M_Three_Op2 */

  VARIABLES
    var_1, var_2
        ⋮
END
```

Then *no other machine may use* `EXTENDS` or `INCLUDES` on `Machine_One`, `Machine_Two` or `Machine_Three`.

However, `TopLevel` *may be included* in some other machine.

TopLevel Machine Structure Diagram

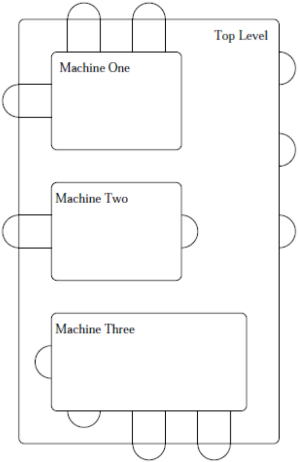

Figure : 9.6 TopLevel Machine Structure Diagram

PART III

Example 1: `Safes` B Specification

- *4 B machines*,
- *INCLUDES & PROMOTES clauses.*

Example 1: `Safes` Specification

This example is taken from Chapter 10 of Steve Schneider's book.

It specifies the actions of locking & unlocking a collection of Bank safes, in terms of doors, locks & keys.

The specification is constructed from four B machines:

- **`Keys`** – records which *keys* are in a door & operations to *insert* & *extract* a key from a door.
- **`Doors`** – records which *doors* are open/closed & operations to *open* & *close* a door.
- **`Locks`** – records which *doors* are locked/unlocked & operations to *open*, *lock* & *unlock* a door.
  It *"includes"* the **`Doors`** machine.
- **`Safes`** – combines the above machines & records which *key* unlocks which *door*; & operations to *insert* & *extract* a key into a door; *unlocking* a door.
  `Safes` *includes* the **`Locks`** & **`Keys`** machines & promotes 3 operations.

Structure of **Safes** Specification

The **Safes** specification is defined by means of a collection of *four* B machines: **Safes**, **Keys**, **Locks** & **Doors**.

The following diagram shows the INCLUDES relationship between the four machines: **Safes**, **Keys**, **Locks** & **Doors**.

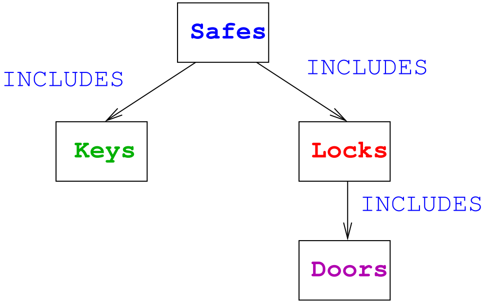

Figure : 9.7 INCLUDES relationship for **Safes** Specification

ProB's **`Safes`** Machine Hierarchy

Using **ProB**'s *Visualize* features a more detailed **`Safes`** Machine Hierarchy can be produced.

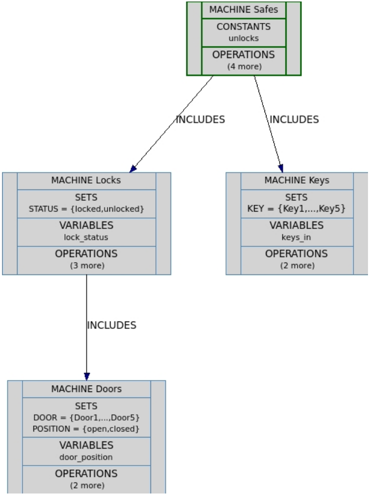

Figure : 9.8 **ProB**'s generated **`Safes`** Machine Hierarchy

Doors Machines' State

Doors State

```
MACHINE Doors

  SETS
    DOOR ;
    POSITION = { open, closed }

  VARIABLES
    door_position

  INVARIANT
    door_position : DOOR --> POSITION  /* Total fn */

  INITIALISATION
    door_position := DOOR * { closed } /* All Doors closed */
```

- There are a set of doors.
- A door's position is either `open` or `closed`.
- Initially, all doors are `closed`.

Doors Machine's Operations

Doors Operations

```
  OPERATIONS

    opening( dd0 ) =
      PRE
          dd0 : DOOR
      THEN
           door_position(dd0) := open
      END ;

    closedoor( dd0 ) =
      PRE
          dd0 : DOOR
      THEN
           door_position(dd0) := closed
      END

END  /* Doors */
```

- `opening( dd0 )` – open a door.
- `closedoor( dd0 )` – close a door.

Keys Machine's State

Keys

```
MACHINE  Keys

  SETS                 KEY

  VARIABLES            keys_in           /* Set of keys in doors */
  INVARIANT            keys_in <: KEY    /* Set of keys */
  INITIALISATION       keys_in := {}     /* No key is in a door */

  OPERATIONS

    insertkey( kk0 ) =
      PRE   kk0 : KEY
      THEN
            keys_in := keys_in \/ { kk0 }
      END ;

    removekey( kk0 ) =
       PRE   kk0 : KEY
       THEN
             keys_in := keys_in - { kk0 }
       END
END  /* Keys */
```

Locks Machine's State

Locks

```
MACHINE  Locks

  INCLUDES
    Doors          /* Includes Doors machine */

  PROMOTES
    closedoor      /* Promotes Doors's closedoor operation,
                      but NOT its opendoor operation. */
  SETS
    STATUS = { locked, unlocked }

  VARIABLES
    lock_status

  INVARIANT
    lock_status : DOOR --> STATUS &
    door_position~[ { open } ] <: lock_status~[ { unlocked } ]

  INITIALISATION
    lock_status := DOOR * { locked }
```

Notes on the **Locks** Machine's State

- The **Locks** machine *"includes"* the **Doors** machine, i.e. it has access to its state & its operations.

- The **Locks** machine *"promotes"* the **Doors** machine's "**closedoor**" operation, i.e. it is added to **Locks** interface.
  But it does not promote **Doors**'s "**opendoor**" operation, i.e. it is **not** added to **Locks** interface.

- The **Locks** machine keeps track of which doors are *locked* & *unlocked* by means of the **lock_status** total function.
  Note: that the **DOOR** type set is used as its *domain* & is defined in the included **Doors** machine.

- The predicate in the **Locks** `INVARIANT` clause:

  ```
  door_position~[ {open} ] <: lock_status~[ {unlocked} ]
  ```

  means that the *"open" doors* are a subset or equal to the *"unlocked" doors*.
  In other words if a door is open then it must have been unlocked or alternatively, that only an unlocked door can be open.

Locks Machine's Operations

Locks Operations

```
  OPERATIONS

    opendoor( dd1 ) =
      PRE   dd1 : DOOR  &  lock_status(dd1) = unlocked
      THEN
            opening(dd1)  /* Locks "calls" Doors operation */
      END ;

    unlockdoor( dd1 ) =
      PRE   dd1 : DOOR
      THEN
            lock_status(dd1) := unlocked
      END ;

    lockdoor( dd1 ) =
      PRE   dd1 : DOOR  &  door_position(dd1) = closed
      THEN
            lock_status(dd1) := locked
      END
END  /* Locks */
```

Safes Machine

Safes

```
MACHINE   Safes

  INCLUDES     Locks, Keys

  PROMOTES     opendoor, lockdoor,     /* Locks Ops */
               closedoor               /* Doors Op */

  CONSTANTS    unlocks
  PROPERTIES   unlocks : KEY >->> DOOR  /* Bijection */

  INVARIANT
    lock_status~[ { unlocked } ] <: unlocks[ keys_in ]
```

- The **Safes** machine *"includes"* the **Locks** & **Keys** machines, i.e. it has access to their state & operations.
- **Safes** *"promotes"* – **Locks**' operations: "**opendoor**", "**lockdoor**" & **Doors**' operation: "**closedoor**".
- The invariant means that the *"unlocked"* doors, i.e. `lock_status~[{unlocked}]`, are a subset or equal to the doors that *"have their key currently inserted"*, i.e. `unlocks[keys_in]`.

Safes Machine's Operations (I)

Safes Operations (I)

```
OPERATIONS

  insert( kk, dd ) =
    PRE
         kk : KEY & dd : DOOR & unlocks(kk) = dd
    THEN
         insertkey(kk)   /* Safes "calls" Keys operation */
    END ;

  extract( kk, dd ) =
    PRE
         kk : KEY             &   dd : DOOR   &
         unlocks(kk) = dd     &   lock_status(dd) = locked
    THEN
         removekey(kk)   /* Safes "calls" Keys operation */
    END ;
```

- **`insert`**`( kk, dd )` – insert the key in the door if it unlocks it.
- **`extract`**`( kk, dd )` – extract the key from the door if it unlocks it & the door is locked.

Safes Machine's Operations (II)

Safes Operations (II)

```
    unlock( dd ) =
      PRE
          dd : DOOR  &  unlocks~(dd) : keys_in
      THEN
          unlockdoor(dd)
      END ;

    quicklock( dd ) =
      PRE
          dd : DOOR  &  door_position(dd) = closed
      THEN
          lockdoor( dd )            ||
          removekey( unlocks~(dd) )
      END

END  /* Safes */
```

- **unlock**(dd) – if the key is in the door's lock then unlock it.
- **quicklock**(dd) – if the door is closed then lock it & remove the key.

ProB's `Safes` Initial State

Using **ProB**'s *Visualize* features the `Safes` system's initial state can be produced.

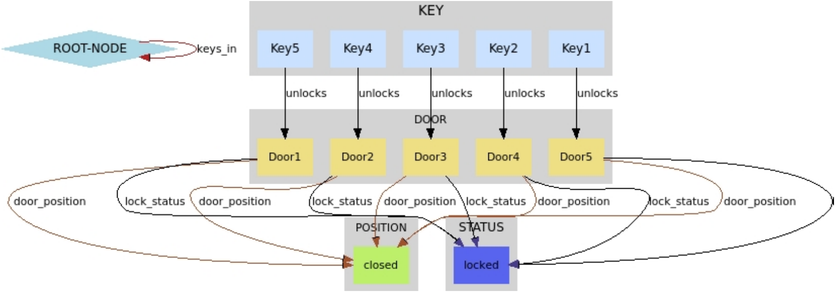

Figure : 9.8 **ProB**'s generated `Safes` Initial State

Structure Diagram for **Safes** Machine

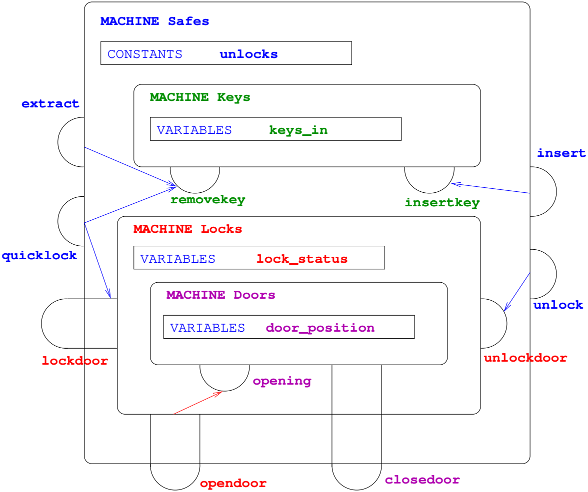

Figure : 9.10 Structure Diagram of **Safes** Machine

PART IV

Structuring a B Specification using: SEES *&* USES

B Machine "Structuring Clauses"(II)

There are two *structuring clauses* that allow one B machine `M2` to use another machine `M1` *without control over it* – `SEES` & `USES`.

`SEES` Clause:

```
MACHINE M2
   SEES M1
```

- Machine `M2` has *"read access"* of `M1`'s state, i.e. its variables.
- **BUT** `M2` **cannot alter** `M1`'s state.
- **BUT** `M2` **cannot be dependent on** `M1`'s state.

`USES` Clause:

```
MACHINE M2
   USES M1
```

- Same relationship as the `SEES` clause.
- **EXCEPT** that machine `M2`'s state **can be dependent on** `M1`'s state.

B Machine "`M2 SEES M1`" Clause

The features of the `SEES` clause are:

- When a machine `M2` *sees* another machine `M1`, it *refers* to this machine & has *read access* to its data: sets, constants & variables.
- `M1`'s variables may be read directly in `M2`'s *initialisation* & *operation definitions*, but `M2` **cannot** refer to them in its *invariant*.
- This means that the *"seeing"* machine `M2`'s state **cannot be dependent on** the *"seen"* machine `M1`'s state.
- Also the *"seeing"* machine can **ONLY** use (call) the operations which **do not modify** the *"seen"* machine's state, i.e. `M2` can only use `M1`'s *enquiry* operations in the definition of its operations.
- The *"seen"* machine is **not** under the control of the *"seeing"* machine, unlike with `INCLUDES` & `EXTENDS`.
- `SEES` is often used to *allow many machines* within a specification to *"share"*/*"see"* (refer to) a particular component (machine).

SEES Diagram

M2 SEES M1

Note there is no link between M2's INVARIANT & M1's VARIABLES, i.e. M2's state **cannot** be related to M1's state.

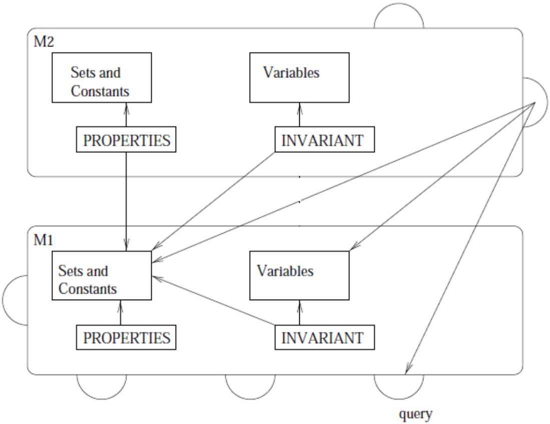

Figure : 9.12 M2 SEES M1 Structure Diagram

B Machine "USES" Clause

The features of the USES clause are:

- It is a generalisation of the SEES relationship & the *"using"* machine has the same access to the *"used"* machine.
- **IN ADDITION**, unlike with the SEES clause, the *"using"* machine *can* refer to the *"used"* machine's *state variables* in its *invariant*.
- This means that USES is used to capture a *relationship between the states of two different machines.*
- That is, the *"using"* machine's state is **related to or dependant on** the *"used"* machine's *state*.

USES Diagram

M2 USES M1

Note there is a **link** between M2's INVARIANT & M1's VARIABLES, i.e. M2's state **can** be related to M1's state.

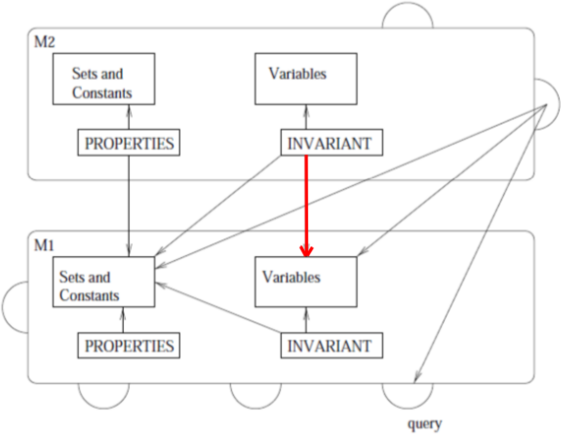

Figure : 9.12 M2 USES M1 Structure Diagram

The `USES` Clause

When `M2` `USES` `M1` this means that:

- `M2` has *read-access* to the variables of `M1`.
- `M2` may call the enquiry operations of `M1`, those operations that **do not change its state**.
  (This may be statically checked.)
- `M2` **cannot** call the operations that *change the state* of `M1`.
- `M2` may also refer to the state variables of `M1` in its `INVARIANT`.
  (Represented by the **arrow** linking `M2`'s `INVARIANT` to `M1`'s `VARIABLES` in Figure 9.12.)
- This means that even though `M2` does not have control over the state of `M1` it still requires a *particular relationship between* the variables of the two machines.
- `M1`'s variables become "*used variables*" of `M2`.
  This means they may appear in the `INVARIANT` of `M2` & on the right hand side of assignments in initialisation & operation definitions.
- A machine may appear in a number of `USES` clauses of other machines.

SEES vs. USES

The differences between "M2 SEES M1" & "M2 USES M1":

- In M2 SEES M1 the variables of M1 **cannot appear in the invariant of** M2.
- This means that there are **no parts of the invariant** of M1 that need to be "passed up" the hierarchy of abstract machines to M2.
- **But**, in M2 USES M1 the variables of M1 **can appear in the invariant of** M2.
- This means that there are **parts of the invariant** of M1 that do need to be "passed up" the hierarchy of abstract machines to M2.
- In the M2 SEES M1 Structure diagram there **is no link** between M2's INVARIANT & M1's VARIABLES.
- **But**, in the M2 USES M1 Structure diagram there **is a link** between M2's INVARIANT & M1's VARIABLES.

PART V

Example 2: Registrar B Specification

- *3 B machines*,
- *Uses clauses:* `USES`, `EXTENDS`, `INCLUDES` *&* `PROMOTES`.

Example 2: **Registrar** Machines

This example is taken from Steve Schneider's book pages 168–171.

It specifies the actions of a *Registrar*: registering *births*, *deaths* & *marriages*.

The **Registrar** specification is defined by means of a collection of *three* B machines: **Life**, **Marriage** & **Registrar**.

- The **Life** machine deals with *births* & *deaths*.
- The **Marriage** machine deals with *marriages*, *divorce* & *who's married to who*.
  It *"uses"* the **Life** machine, i.e. it USES **Life**, so it has *read access* to its state.
- The **Registrar** machine combines the above machines & defines a *"dies"* operation that takes into account marriage.
  Registrar *includes* the **Life** machine & *extends* the **Marriage** machine, i.e. INCLUDES **Life**, EXTENDS **Marriage**.

Figure 9.13 shows the EXTENDS, INCLUDES & USES relationship between the three machines.

ProB's `Registrar` Machine Hierarchy

Using **ProB**'s *Visualize* features a more detailed `Safes` Machine Hierarchy can be produced.

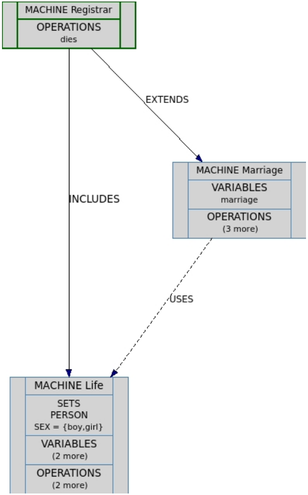

Figure : 9.13 **ProB**'s generated `Registrar` Machine Hierarchy

Life Machine's State

Life

```
MACHINE  Life

  SETS
    PERSON ;
    SEX = { boy, girl }

  VARIABLES
    male, female

  INVARIANT
    male <: PERSON & female <: PERSON &
    male /\ female = {}

  INITIALISATION
    male := {} || female := {}
```

The **Life** machine records the *births* & *deaths* of *males* & *females*.

The *invariant* means that no one is both male & female.

Life Machine's Operations

Life Operations

```
OPERATIONS

  born( pp, sex ) =
    PRE  pp : PERSON & pp /: (male \/ female) & sex : SEX
    THEN
        IF   ( sex = boy )
        THEN   male   := male   \/ { pp }
        ELSE   female := female \/ { pp }
        END
    END ;

  die( pp ) =
    PRE  pp : PERSON  &  pp : ( male \/ female )
    THEN
        IF  ( pp : male )
        THEN   male   := male   - { pp }
        ELSE   female := female - { pp }
        END
    END

END  /* Life */
```

Marriage Machine's State

Marriage

```
MACHINE  Marriage

  USES             Life    /* "read" access to state */

  VARIABLES        marriage
  INVARIANT        marriage : male >+> female /* Injective */
  INITIALISATION   marriage := {}

  OPERATIONS

    wedding( mm, ff ) =
      PRE   mm : male    &  mm /: dom(marriage)   &
            ff : female  &  ff /: ran(marriage)
      THEN
            marriage(mm) := ff
      END ;
```

- **Marriage** machine *"uses"* the **Life** machine, i.e. it has *read access* to **male** & **female**, but it does *not control it*, i.e. it **cannot** alter their values.
- The machine's *invariant* means that a person can only be married to one person at a time.

Marriage Machine's Operations

Marriage Operations

```
part( mm, ff ) =
  PRE   mm : male   &   ff : female   &
        mm |-> ff : marriage
  THEN
        marriage := marriage - { mm |-> ff }
  END ;

pp <-- partner( pp ) =
    PRE  pp : PERSON   &
         pp : dom(marriage) \/ ran(marriage)
    THEN
        IF ( pp : dom(marriage) ) /* pp is husband */
        THEN
             pp := marriage(pp)    /* wife */
        ELSE
             pp := marriage~(pp)   /* husband */
        END
      END

END  /* Marriage */
```

Registrar Machine

Registrar

```
MACHINE  Registrar

 EXTENDS    Marriage   /* All operations promoted */
 INCLUDES   Life
 PROMOTES   born       /* Promotes Life's "born" operation,
                          but not "die" operation. */

 OPERATIONS

   dies( pp ) =
   PRE  pp : PERSON  &  pp : male \/ female
   THEN
         die( pp )                       /* Calls Life's "die" */
     ||
        IF   ( pp : dom(marriage) )      /* husband died */
        THEN
            part( pp, marriage(pp) )     /* Marriage's "part" */
        ELSIF ( pp : ran(marriage) )     /* wife died */
        THEN
            part( marriage~(pp), pp )    /* Marriage's "part" */
        END
   END
END /* Registrar */
```

ProB's Example `Registrar` State

Using **ProB**'s *Visualize* features an example of a `Registrar` state after: birth of 2 males & 2 females, and 2 marriages.

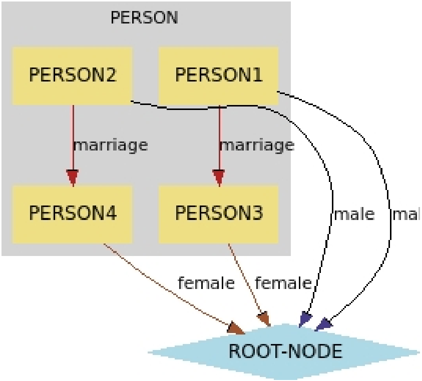

Figure : 9.14 **ProB**'s generated `Registrar` Example State

Structure Diagram for **Registrar** Machine

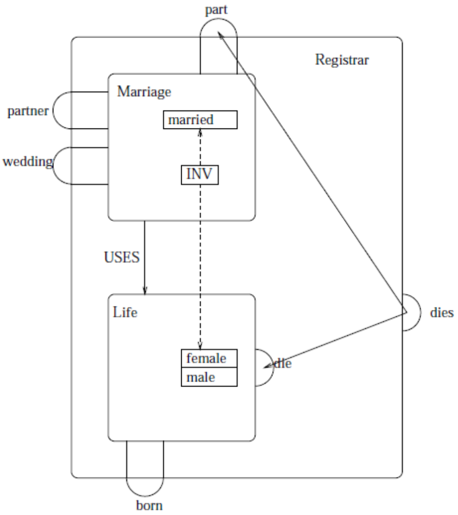

Figure : 9.15 **Registrar**'s Structure Diagram