6SENG001W Reasoning about Programs

Lecture 3

B's Abstract Machines

&

AMN Programming Language Constructs

Overview of Lecture 3: Introduction B's AMs & AMN

The aim of this lecture is to:

- Outline the *key elements* of a System that need to be specified.
- Describe the B-Method's concept of an *Abstract Machine*:
  - what it is meant to represent,
  - its structure & components – the *"clauses"*,
  - its operations.
- *Abstract Machine Notation* (AMN) *programming language constructs*.
- Example *Abstract Machine*: Club membership.

PART I

Key Elements of a System that Need to be Specified

B-Method Software Development Stages

As we have seen in a previous lecture the software development process using the B-Method can be broken down into several “phases” or “stages”:

The first two stages are:

- *Capture the System Requirements*
- *Design a “B Model” (Abstract Model) from the System Requirements*

The first stage of the actual B development begins with the second of these — *“design of a B model”* from the system requirements.

So before we see how this is achieved by using a B *abstract machine*, we need to look at some of the general properties of the systems we will be specifying.

Types of System States

The first thing we need to consider are

all the different types of possible system states,

these are represented in the following diagram.

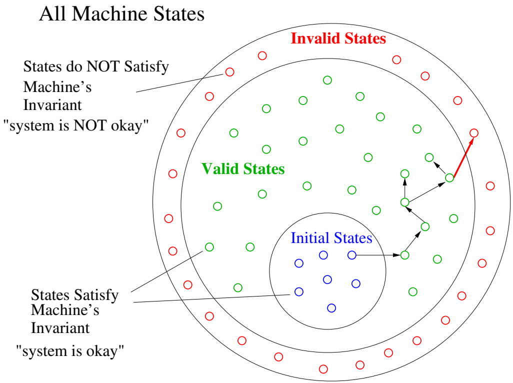

Figure : 3.1 All Possible System States

Categories of System States

Any system can be in one of *three different types of states*:

- **Valid states**
  - those that satisfy defined *constraints* or *properties*.
  - These constraints are known as the system (or state) *"invariant"*.
  - The *"invariant"* is one of the *most important parts* of a system.
  - Because it defines what is a *valid system state*.
  - E.g. for dates – $\mathbf{9^{th}}$ **October 2021**, but not $\mathbf{30^{th}}$ **February 2021**
- **Initial states** (or **start states**)
  - one or many **valid states** of the system that are defined as appropriate starting states for the system.
  - E.g. Unix start date – $\mathbf{1^{st}}$ **January 1970**.
- **Invalid states** (or **error states**)
  - states that **do not satisfy** the system *invariant*.
  - E.g. $\mathbf{29^{th}}$ **February 2017**.

Example: Stack States

Consider a *stack* with a maximum size, then the:

- *System invariant* for this stack is that —

  "zero $\leq$ the number of items $\leq$ maximum size".

- *Valid states* of the stack are those that satisfy the system *invariant*, i.e. zero $\leq$ the number of items $\leq$ maximum size.

- *Initial state* would be the empty stack.
  - This also *satisfies* the *system invariant*.
  - This is an *essential property* relating the *initial state* & *system invariant*.

- *Invalid states* of the stack are those that do not satisfy the *system invariant*:
  the number of items $<$ zero,
  or
  the number of items $>$ maximum size.

Examples of a Stack's States

Assume that a stack has a *maximum size* of 10 then Fig. 3.2 illustrates examples of its *Initial*, *Valid* & *Invalid* states.

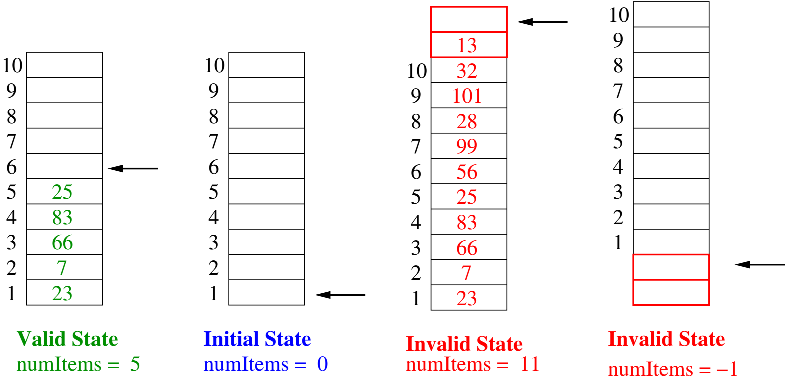

Figure : 3.2 Examples of Initial, Valid & Invalid Stack States

Executing System Operations

An *execution* of the system:

- Starts from one of the *initial states*.
- When an operation is *performed correctly* the system moves from one *valid state* to another *valid state*.
- If an operation is **performed incorrectly** then the system may move from a *valid state* to an *invalid state*.

NOTES

All the states the system passes through by performing operations **MUST SATISFY** the **invariant**.

We shall consider that an *"operation is performed incorrectly"* if it would result in the system ending up in an *invalid state*, i.e. one that does **not satisfy** the *invariant*.

And an *"operation is performed correctly"* if it results in the system ending up in a *valid state*, i.e. one that does **satisfy** the *invariant*.

Example: Executing Stack Operations push & pop

In relation to our *stack* with a maximum size, then the:

- Execution starts from the *empty* stack.
  (An empty stack is a valid state.)

- A *push* operation can be performed successfully if the stack is **not full** before the *push* is attempted.

- A *push* operation moves from a *valid state* to an *invalid state* if the **stack is full** before the *push* is attempted.
  Results in the number of items $>$ maximum size.
  A state that does **not satisfy** the *invariant*.

- A *pop* operation moves from a *valid state* to an *invalid state* if the **stack is empty** before the *pop* is attempted.
  Results in the number of items $<$ zero.
  A state that does **not satisfy** the *invariant*.

Ensuring an Operation is *"Performed Correctly"*

As we have seen, it is essential to ensure an operation is *"performed correctly"*, i.e. the system moves from one *valid state* to another *valid state*.

When defining an operation it is necessary to determine the states of the system in which the operation can be performed correctly, i.e. *before* states.

These states are characterised by means of the *"precondition"* of the operation.

In addition it is necessary to specify the effect of the operation on the states of the system, i.e. *after* states.

These states are characterised by means of the *"post-condition"* of the operation.

It is also essential that these *after states satisfy the invariant*, i.e. are *valid state*.

The act of specifying an operation is then a process of specifying the *pre-* & *post-conditions*.

Operations also input data via parameters & output data as results.

Specifying an Operation

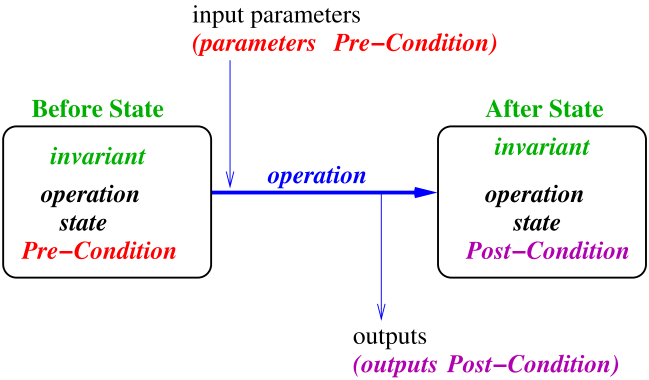

Figure : 3.3 Specifying an operation

To be able to *specify an operation* like the one above, then we need to be able to formalise each of the above components of an operation.

Questions for Specifying an Operation

The questions that have to be answered when *specifying an operation* are:

Pre-Condition components:

- What are the *data input parameters* to the operation?
- What are the *preconditions* the *parameters* must satisfy?
- What are the *preconditions* the *state* must satisfy?

Post-Condition components:

- What part of the *state is altered & how*?
- What part of the *state remains the same*?
- What are the *data outputs* from the operation?
- What *output report* should be made – none, success or error?

Specifying a System in B using an Abstract Machine

As we have just seen the systems we are going to specify in B have the following components:

- *Information* that makes up the *state* of the system.
- *Properties* the information must satisfy, i.e. the state *invariant*.
- *Operations* that transform the state, defined in terms of *pre-* & *post-conditions*.

We are going to specify these systems by constructing a *B-model* of the system.

A B model is a *"high level"*, *"abstract" specification* of:

- *what a system should be like* or *how it should behave*;
- rather than *how it should be implemented*.

Where a B specification takes the form of a B *abstract machine*, e.g. the `PaperRound`.

PART II

The B-Method's Abstract Machines

Overview of an Abstract Machine (AM)

An *abstract machine* is the basic structure for writing a specification in the B-method.

In the sense that an *abstract machine* is similar to a *class definition* in an OO language, e.g. Java, C++.

The main parts of an *abstract machine* are:

- a *name*,
- it can take *parameters*,
- a *local state* represented by *encapsulated state variables*,
- *state invariant* that the state variables must satisfy at all times,
- *initialisation* of the state,
- a collection of *operations* that allow the state variables to be accessed & manipulated,
- **all operations MUST maintaining the state invariant**.

An *abstract machine* is written in *AMN* & uses a lot of mathematical notation as well as some programming style notation.

General Structure of an Abstract Machine

Name

```
MACHINE Name( mparam1, ..., mparamN )

  CONSTRAINTS          CP1 & CP2 & ..... & CPn

  SETS                 SS1 ; SS2 ; ... ; SSn
  CONSTANTS            C1, C2, ... , Cn
  PROPERTIES           PR1 & PR2 & ... & PRn

  VARIABLES            var1, var2, ..., varn
  INVARIANT            INV1 & INV2 & ... & INVn
  INITIALISATION       var1, ..., varn  :=  val1, ..., valn

  OPERATIONS

  output1, .., outputM <-- operation_1( param1, .., paramN )
     =  PRE   PreCondition
        THEN
                Substitution
        END

   ...
END
```

“Clauses” of an Abstract Machine

An *abstract machine* is divided up into a number of *“clauses”*.

Each *clause* is concerned with specifying a particular part of the abstract machine, i.e. the system being specified.

The *clauses* incorporate the *static* (e.g. invariant) & *dynamic* (operations) description of its behaviour.

Not all types of clauses need to be included in an abstract machine, i.e. clauses are optional.

But some clauses must be included as a group.

For example, if a machine wants to *use a variable* then these 3 clauses `VARIABLES`, `INVARIANT` & `INITIALISATION` **must be included**.

Which individual clauses & groups of clauses are used to define an abstract machine is dependant on how complex the system is that the machine is designed to represent.

Purpose of an Abstract Machine's Clauses

The main clauses are:

MACHINE	declaration of the abstract machine's *name* & optional *list of parameters*
CONSTRAINTS	declaration of the *properties* the *machine's parameters* **must satisfy**
SETS	declaration of *deferred* & *enumerated sets*
CONSTANTS	declaration of *constants*
PROPERTIES	declaration of the *properties* the machine's *sets* & *constants* **must satisfy**
VARIABLES	declaration of *variables*
INVARIANT	declaration of *invariant properties* of the variables, **must include every variable's type**
INITIALISATION	*initialisation* of all variables
OPERATIONS	declaration of the *operations* in the form of an *interface* (header) & *body*

Machine Clause Visibility

The visibility & relationships between the machine clauses are given in the following diagram.

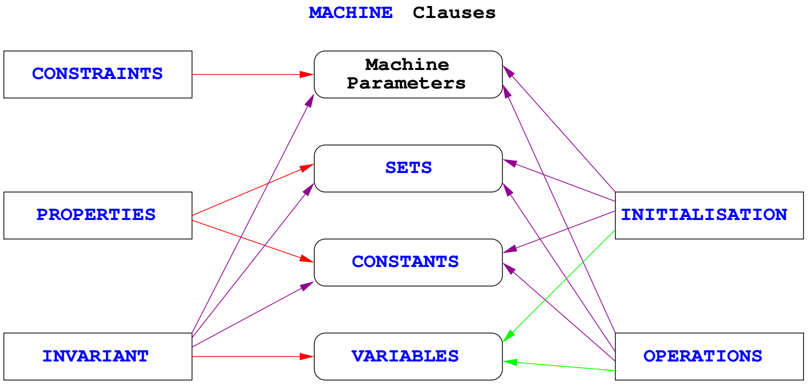

Figure : 3.4 Machine Clause Visibility

Where: "⟵" *"type & properties of"*, "←" *"is visible"* & "←" *"can modify"*.

Diagrammatic View of an Abstract Machine

An *abstract machine* can be represented by a *"Structure Diagram"* that presents an overview of its structure in terms of its:
Data – *sets & constants*, *properties & types of sets & constants*, *state variables*, *state invariant*,
Interface – *operations*.

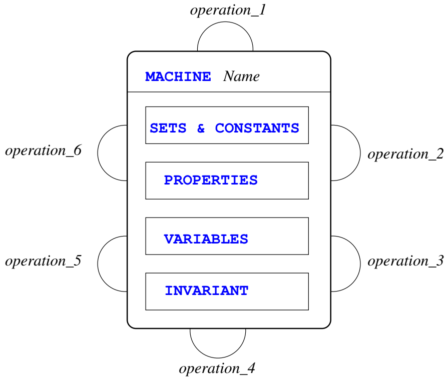

Figure : 3.5 A "Structure Diagram" for an Abstract Machine

Abstract Machine Parameters

An abstract machine can be made *"generic"* by using parameters in its definition.

Sets & *constants* can be provided as parameters of the machine.

- *Sets* must be written all in `UPPER CASE`,
- *Constants* must be written in `lower case`.

For example, for an abstract machine that is used to model student module registration & grades called: `Register`,

Register

```
MACHINE
   Register( GRADE, top, maxreg )
```

The sets that are passed, e.g. `GRADE`, can be used as types within the machine.

`top` & `maxreg` are constants.

Constraints on Machine Parameters

Any *constraints* about a machine's parameters is provided in a `CONSTRAINTS` clause.

This clause must contain *type information* about all of the constants, e.g.

Constraints

```
CONSTRAINTS
  maxreg : NAT1   &   top : GRADE
```

It cannot constrain the *type* of the *sets* being passed as parameters, but it can place *non-type* constraints on them.

For example, `GRADE` must contain at least 2 elements:

Constraints

```
CONSTRAINTS
  card( GRADE ) >= 2
```

Abstract Machine State

The *state* of the machine should correspond to how the machine is to be understood.

It need not correspond to what can be implemented directly on a computer.

The *machine's state* in defined by means of *variables*.

The *variables* can have whatever *type* is most appropriate to the specification.

For example, a machine which keeps track of the members of a club can use a variable that contains the set of members:

Club State

```
VARIABLES
   members
```

Note: in the B tools *variable names must be at least two characters long*.

Abstract Machine Type Information

Types are particular sets.

Each *state variable* must be accompanied by information concerning its *type*.

Typing information about the state variables is contained in the machine's `INVARIANT` clause.

Type information is of the form:

- $var \in SS$ — var is a member of the type SS, or equivalently that var has type SS.
- $var \subseteq SS$ — var is a subset of SS, or equivalently that var has type $\mathbb{P}(SS)$

For example, for the set of members of a club, assuming that `NAME` is the set of possible names, we could define its type using either of the following declarations:

Club Invariant

```
INVARIANT
   members <: NAME
   /* or */
   members : POW( NAME )
```

Abstract Machine Invariant

The machine's *invariant* contains *consistency conditions* about the information contained in the machine, i.e. what is a *valid state*.

The *invariant* is sometimes called the *"static specification"* of the machine: it describes something that **must be true of every state the machine can reach**.

If the *invariant* becomes *false* during an execution of the machine, then an error has occurred.

The *invariant* is a *logical* statement about the *state variables*, it provides:

- *type information*,
- *conditions* that must hold: on *particular variables* & the *relationships* between them.

For example, the paper round manager might require that no more than 75 houses can have papers delivered, so the invariant would be:

PaperRound Invariant

```
INVARIANT
   houseset <: NAT1  &  card(houseset) <= 75
```

Abstract Machine Initialisation

The state must be *initialised*, i.e. the machine's *state variables* must all be assigned an *initial value*.

Initialisation of the machine's variables must *satisfy the invariant*.

This means that the machine's *initial state* is a *valid state*.

The initialisation clause of a machine is specified using the *pseudo-programming language* part of AMN, i.e. using *assignment* (:=).

For example, initially no houses have papers delivered:

PaperRound State Initialisation

```
INITIALISATION
   houseset := {}
```

Abstract Machine Operations

The *operations* describe how the machine can *change its state*.

The operations are sometimes called the *dynamic specification* of the machine, because they specify how it will *behave*.

An operation is a *one-step change of state* & optionally takes *parameters* & produces *output values*.

Operation's can contain a *precondition*: a predicate expressing the conditions necessary to invoke the operation.

Operations & *invariants* are tightly coupled:

- *invariants* must always be preserved by all of the operations,
- provided they are called when their *precondition is true*.

If all of the operations preserve the *invariant*, then the machine can never reach an *invalid state*. (See Figure 3.1.)

Operations listed in a machine description are *separated* by semi-colons "**;**", but the last operation **does not have a semi-colon**.

Components of an Operation

In general, specifications of operations require the following information:

- *"Operation Interface"* – is defined as the:
  - *Name*.
  - Optional list of *Input parameters* – values input to the operation. If the operation has no parameters, just give its name i.e. use "`op_no_params`" not "`op_no_params()`".
  - Optional list of *Output variables* – the results of the operation are assigned to these in the *body*.
- *Precondition* – constraints on the *input parameters* & the *machine's state variables*.
  If this is **not true** then the operation *cannot be called & executed*.
- *Body* of the operation – describes its effect, i.e. what the operation does.
  - modify the *state variables* &
  - assigns results to the *output variables*.

Examples, see `PaperRound` & `Club`.

Operation Preconditions

An operation is usually described using the AMN `PRE`-`THEN`-`END` construct:

Precondition

```
PRE  PC
THEN Substitution
END
```

Where `PC` is the *precondition* on the operation.

It describes restrictions on the *parameters* & on the *state* of the machine.

The operation should only be called when the *precondition is true*.

The precondition must give the *type* of all *input parameters*.

It can also include other constraints on the machine's *state variables*.

NOTE: if an *operation has parameters* then the *types of the parameters* **MUST** be given & this can **ONLY** be done in the *precondition* part of a `PRE`-`THEN`-`END` command.

Example Operation Preconditions

Examples of preconditions:

PaperRound ``add'' Precondition

```
new : NAT1

new : NAT1 & new < 163
```

If the *precondition* is just *"true"* then we do not need to use the `PRE` construct, but can just use a code block: `BEGIN` – `END`.

Operation Body or "Substitution"

The *body* of the operation (`Substitution`) describes its effect, it must describe how the *machine's state* is updated & the output to be provided.

An operation's body is called a *"substitution"* because the *new state variable values* produced by the operation are *substituted* for the *existing state variable values*.

The body is defined by an *assignment* (:=)

- *State variables* may optionally be assigned, if they are not, the operation does not alter them.
- All *output variables* must be assigned.

Paper round example:

PaperRound ``getspapers'' Operation

```
ans <-- getspapers( housenumber ) =
        PRE  housenumber : NAT1 & housenumber : houseset
        THEN
            ans := yes
        END ;
```

PART III

AMN's Programming Language Constructs

B-Method's Abstract Machine Notation (AMN)

B-Method specifications are written in its formal language – *Abstract Machine Notation* (AMN), it consists of the following elements:

- Predicate logic: used to express properties relating to all the data of a component.
  E.g. *"invariants"* properties of variables,
  "preconditions" under which an operation can be called,
  "proof obligations".

- Expressions: these are formula describing data, i.e. the data's type & value. (Using numbers, sets, relations, functions & sequences.)

- Substitutions: mathematical notation used to describe the *dynamic* aspect of B components – "*state changes*".
  This is the definition of the *body* of an operation, using AMN's programming language constructs.

- Components: Abstract Machines, Refinements, Implementations.

AMN's Programming Language Constructs

These are used to define the following components of a B machine:

- the initialisation of the *state variables*,
- the definition of the *body of an operation*.

AMN includes the following *programming language constructs*:

- *assignment* & *multiple assignment*: "`:=`"
- *conditional* constructs:
  - several `IF`- `THEN`- `END` variants,
  - `CASE`
- the *"do nothing"* command: `skip`,
- *block structure*: `BEGIN S END`, similar to Java's block structuring construct "`{ .. }`".
- *parallel execution*: "`||`"

AMN: Assignment

An *assignment* has the form

```
xx := E
```

It calculates the value of the expression `E`, and then overwrites the value of variable `xx` so it now contains this value.

A *multiple assignment* has the form

$$var_1,\ var_2,\ ...,\ var_n \ := \ exp_1,\ exp_2,\ ...,\ exp_n$$

All of the expressions: $exp_1,\ exp_2,\ ...,\ exp_n$ are first of all evaluated, & then the result of each is *simultaneously assigned* to the corresponding variable: $var_1,\ var_2,\ ...,\ var_n$, overwriting their previous contents.

For example:

```
xx, yy := yy, xx
houseset, num := houseset \/ {new}, card(houseset \/ {new})
```

AMN: `IF` Conditional

The `IF` statement provides for usually single or binary choice based on the value of an expression.

The expression **must be a boolean** value.

The AMN `IF` statement is similar to Java's `if` statement.

The conditional construct has several forms in AMN:

IF Conditionals

```
IF ( B ) THEN S END

IF ( B ) THEN S ELSE T END

IF ( B1 ) THEN S ELSIF ( B2 ) THEN T ELSE U END
```

Where `B, B1, B2` are boolean expressions.

Both the `ELSE` & `ELSIF` clauses are optional.

Execution of the `IF` statement

The AMN `IF` statement is executed in exactly the same way as a standard programming language `if` statement.

Note that in particular, the *state of the machine remains unchanged* in the following circumstances:

1. The `ELSE` clause is *omitted* & the `IF` *condition is false*.
2. The `ELSE` clause is *omitted* & the `IF` *condition is false* & all of the `ELSIF` *conditions are false*.

Conditional Examples

IF Conditionals

```
IF  ( xx < yy )
THEN
      xx, yy := yy, xx
END


IF  ( xx < yy )
THEN
      largest := yy
ELSE
      largest := xx
END


IF  ( xx < yy )
THEN
      smallestXY := xx
ELSIF ( yy < zz )
    THEN
          smallestYZ := yy
    ELSE
          smallestYZ := zz
END
```

AMN: CASE statement

The CASE statement provides for multiple choice based on the value of an expression.

The expression does not have to be a *boolean* value, as in the case of the IF statement.

The AMN CASE statement is similar to Java's `switch` statement.

The form of the AMN CASE statement is as follows:

CASE

```
CASE E OF
  EITHER
         val1 THEN S1
  OR
         val2 THEN S2
  OR
         val3 THEN S3
  ...
  ELSE Sm
END
```

The ELSE part is optional.

Execution of the `CASE` statement

The `CASE` statement is executed as follows:

1. The expression `E` is evaluate.
2. If `E` evaluates to `val1` then statement `S1` is executed.
3. If `E` evaluates to `val2` then statement `S2` is executed.
4. Etc.
5. If `E` **does not** evaluate to any of the listed case values `val1, ..., valn` then:
   5.1 If the `ELSE` part is **present** then statement `Sm` is executed.
   5.2 If the `ELSE` part is **not present** then nothing is executed & the *state remains unchanged*.

AMN: “Do Nothing”

`skip` is the *“do nothing”* or *“no-operation”* command.

It is the command that has *no effect on the state of the machine*.

In other words, it *does not change the state of the machine*.

This may seem like a completely pointless programming language construct but it can be extremely useful to be able to *explicitly state* that at some point you *do not want anything to happen*, in particular you *do not want the state to change*.

This is similar to why we have the number *zero* & its numeral *”0”* in mathematics, there are occasions when we want to say that we have nothing, i.e. *0 things*.

For example, when dividing by 0:

skip

```
IF ( yy /= 0 )
THEN  result := xx div yy
ELSE  skip
END
```

AMN: Parallel Execution

Statements can also be specified in *parallel*:

```
S || T
```

The variables **updated** by `S` and by `T` **must be distinct**, i.e. the same variable **cannot** be updated by both `S` & `T`.

Examples: these two are equivalent.

```
xx := 4   ||   yy := 7
xx, yy  :=  4, 7
```

The following swaps `xx` and `yy`, and assigns `7` to `zz`.

```
xx := yy   ||   yy, zz := xx, 7
```

The components of a parallel composition can be any statement, not only assignment:

Parallel Execution

```
xx := 4  ||  IF  (yy < xx )  THEN  yy, zz := xx-1, yy  END
```

PART IV

Example: Club Membership

Example: `Club` Membership

This example B machine specifies a simple club membership system.

The club is parameterised on a set of `NAME`s for its members & the maximum `capacity` for members.

Before a person can join the club, i.e. be on the `members` list, they must first be on the `waiting` list.

Both lists have a maximum size & no one can be on both.

There are five club operations:

- `join` – a person joins the club by moving from the `waiting` list to the `members` list.
- `join_queue` – a person joins the club's `waiting` list.
- `remove` – a member leaves the club.
- `semi_reset` – resets the club lists, resulting in all members being moved to the `waiting` list.
- `is_member` – checks whether someone is a member of the club.

Example: Club B-Machine State

Club State

```
MACHINE   Club(NAME, capacity)

  CONSTRAINTS        capacity : NAT1 & 5 <= capacity &
                     capacity < card(NAME)

  SETS               ANSWER = { yes, no }

  CONSTANTS          queuetotal

  PROPERTIES         queuetotal : NAT1 & queuetotal > 2

  VARIABLES          members, waiting

  INVARIANT          queuetotal < capacity
                     & members <: NAME
                     & waiting <: NAME
                     & members /\ waiting = {}
                     & card(members) <= capacity
                     & card(waiting) <= queuetotal

  INITIALISATION     members := {}  ||  waiting := {}
```

Example: Club Operations: join, join_queue, remove

Club Operations (1)

```
OPERATIONS

  join( newmember ) =
    PRE  newmember : waiting & card(members) < capacity
    THEN
             members := members \/ { newmember }
          || waiting := waiting - { newmember }
    END ;

  join_queue( newmember ) =
    PRE  newmember : NAME  &  newmember /: members  &
         newmember /: waiting  &  card(waiting) < queuetotal
    THEN
           waiting := waiting \/ { newmember }
    END ;

  remove( member ) =
    PRE  member : members
    THEN
           members := members - { member }
    END ;
```

Example: Club Operations: semi_reset, is_member

Club Operations (2)

```
  semi_reset =
    BEGIN
           members, waiting := {}, members
    END ;


  ans <-- is_member( member ) =
    PRE   member : NAME
    THEN
          IF  ( member : members )
          THEN
                ans := yes
          ELSE
                ans := no
          END
    END

END
```

Structure Diagram for `Club` Abstract Machine

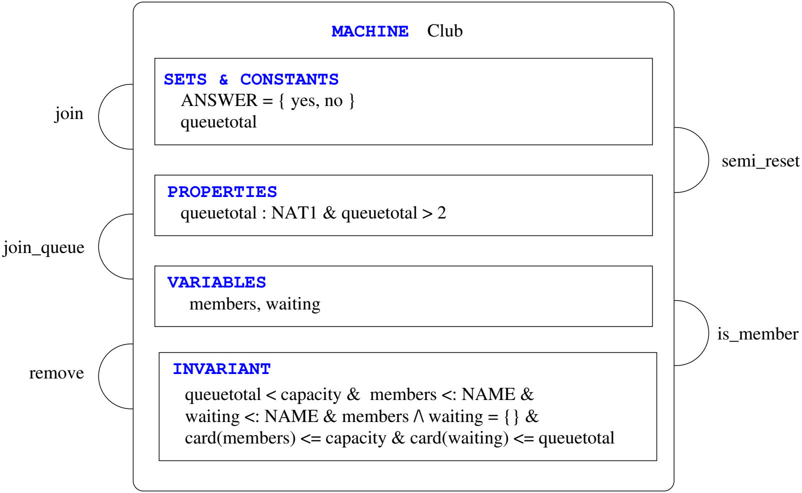

Figure : 3.6 Club Structure Diagram