UPPSALA UNIVERSITET

Welcome to

Provably Correct Software

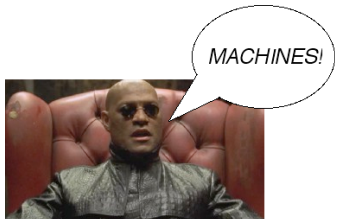
http://www.it.uu.se/
edu/course/homepage/bkp/vt08

Instructor

Lars-Henrik Eriksson

lhe@it.uu.se,
http://www.it.uu.se/katalog/lhe?lang=en

What is the B-method?



What is the B-method (really)?

The *B-method* is a *formal method* used for

- Formal specification of software (using the *Abstract Machine Notation – AMN*)
- Writing executable programs (using the *B0* subset of AMN)
- Proving consistency of specifications and correctness of programs

Characteristics:

- *Model-based specification*
- *Refinement*

The B-method is supported by software tools such as

- Atelier B
- B-Toolkit
- ProB

The software development process

- Requirements capture
- Specification
  Traditionally done using plain language, diagrams, tables ...
- Validation (are we building the right system?)
  Traditionally done by inspection, prototyping ...
- Design
  Specify the architecture and data structures of the software
- Implementation
  Programs written in a programming language
- Verification (are we building the system right?)
  Traditionally done by testing
- Debugging
  Try to find out where the program goes wrong

The role of B in software development

- Requirements capture
- Specification
  Wholly or in part written in AMN.
- Validation (are we building the right system?)
  Proving correctness theorems, animating the specification...
- Design
  Design specifications wholly or in part written in AMN.
- Implementation
  Programs written in the B0 subset of the AMN.
- Verification (are we building the system right?)
  Refinement proof. Testing should not be needed.
- Debugging
  You don't need this (at least not in the traditional sense)

Model-based specification

The specification gives a *mathematical model* of the data the program uses and describes the function of the program in terms of *mathematical operations* on that data.

Consider a *stack*:

- A stack can be modelled as a *sequence of objects*.
- Assume that the top element is always the last element of the sequence.
- Pushing an item onto the stack means the same as adding it to the end of the sequence.
- Popping an object off the stack means removing the last element from the sequence.

A stack in B (simplified)

A stack can be formally specified by the following B *specification machine* (or *abstract* machine) written in AMN.

```
MACHINE Stack
SETS ELEMENTS
VARIABLES stack
INVARIANT stack:seq(ELEMENTS)
INITIALISATION stack := <>
OPERATIONS
   xx <-- get =    xx := last(stack);
   push(xx)   = stack := stack<-xx;
   pop        = stack := front(stack)
END
```

Actually, you would not be able to develop a program conforming to this specification because some important things are missing.
Can you see what? (Think about the Prog. Methodology 1 course...)

B ensures error-free execution

- There must be a size limit for the stack (as computer memory is finite)
- There must be preconditions on the operators to make sure that they are well-defined. (What happens if you pop an empty stack?)

A better specification

```
MACHINE Stack
CONSTANTS maxsize
SETS ELEMENTS
PROPERTIES maxsize:NAT
VARIABLES stack
INVARIANT stack:seq(ELEMENTS) & size(stack)<=maxsize
INITIALISATION stack := <>
OPERATIONS
   xx <-- get = PRE stack /= <>
                THEN xx := last(stack)
                END;
   push(xx)   = PRE xx:ELEMENTS & size(stack)<maxsize
                THEN stack := stack<-xx
                END;
   pop        = PRE stack /= <>
                THEN stack := front(stack)
                END
END
```

`NAT` is the set of *implementable* natural numbers (has upper limit).
B guarantees that preconditions are satisfied when operations are used (*design by contract*).

Implementing Stacks – refinement

The stack specification does not concern itself with implementation details. Stacks are actually implemented by a B *implementation machine*. This machine must be a *refinement* of the specification. Intuitively, a refinement is something which is the same but more *concrete*. For example:

- undetermined things (e.g. maximum stack size) are decided
- algorithms are provided for abstract operations (e.g. a quantified expression can be refined by a loop).
- an operation can be implemented in terms of other simpler operations (stepwise refinement).
- mathematical objects like sequences are replaced by implementable objects like arrays.

A stack implementation

```
IMPLEMENTATION StackI
REFINES Stack
VALUES ELEMENTS = INT; maxsize = 100
CONCRETE_VARIABLES array, currentsize
INVARIANT array:(1..maxsize)-->ELEMENTS &
          currentsize:0..maxsize &
          !ii.(ii:1..currentsize =>
               stack(ii) = array(ii)) &
          currentsize = size(stack)
INITIALISATION array := (1..maxsize)*{0}; currentsize := 0
OPERATIONS
   xx <-- get      = xx := array(currentsize);
   push(xx)        = BEGIN
                       currentsize := currentsize+1;
                       array(currentsize) := xx
                     END;
   pop             = currentsize := currentsize-1
END
```

The stack is stored as an array. When items are pushed on the stack, they are stored in successive array elements.
(If you are curious – identifiers must have at least 2 characters.)

Sequence of refinements

Sometimes the step from specification to implementation is too large. The refinement can then be done as a series of smaller refinements. The intermediate stages are represented by B *refinement machines*.

In the sequence of refinements, the machines get successively more concrete, until the implementation machine is reached.

Algebraic specification

A different specification technique is to give equations that describe what *properties* the *operations* should have. A stack could be specified by the following (in)equations: (assuming `s` ranges over stacks, `e` over elements and `empty` represents the empty stack).

```
get(push(s,e)) = e
pop(push(s,e)) = s
push(s,e) ≠ empty
get(empty) ≠ e
pop(empty) ≠ s
```

B is not intended for algebraic specifications, but you can with difficulty abuse the notation to write such specifications.

Atelier B

We will use the *Atelier B* tool. It can:

- Do syntax and type checking of B machines
- Generate *proof obligations* for the consistency of machines
- Generate proof obligations for refinements
- Prove *most* proof obligations
- Translate implementation machines into C (or C++ or ADA)
- Generate basic documentation of machines
- Manage projects with many developers

See the course web site for instructions on how to run Atelier B!

Atelier B is a commercial product used in industrial software development.

(Unfortunatelty the graphical user interface is somewhat primitive.)

ProB

We will also use the *ProB* tool to validate specifications.
Some things it can do:

- *Animate* B specification machines
- Check internal consistency of a machine by automatic testing

ProB is research software under development.

- It does not implement the full AMN, so not all B specifications can be animated.
- It requires limites ranges of numbers and sizes of sets to work.
- It does give a very clear view of what the B machine is doing.

See the course web site for instructions on how to run ProB!

About the B-method

The B method was developed with practical software development in mind. It brings together ideas from various areas of computer science (and mathematics). Some of them are:

- Axiomatic set theory (Zermelo-Fraenkel)
- Model-based specifications (Z, VDM-SL)
- Pre- and postconditions
- Design by contract
- Invariants
- Guarded commands
- Weakest precondition semantics
- Hoare logic (axiomatic semantics)
- Refinement calculus
- Stepwise refinement

The course

- Lectures outline the material and point out important issues.
- Students study the details from the textbook and other sources (web resources, research papers…)
- Weekly seminars with presentations by students and discussions.
- During the course groups of (2) students carry out a program development project including specification, implementation and proof.
- No proper exam. The seminars can be seen as an ongoing oral exam.

The projects

- Suggest a *small* programming task. Get the instructor's approval.
- Write a B specification machine (or machines)
- Validate it, prove its consistency
- Write a B implementation machine (or machines)
- Prove that it is a refinement (possibly using intermediate refinement machines)
- Generate an executable program and run it. Is it bug-free?
- Write a project report!

Things to consider

- B0 is quite a primitive programming language. The only data structure available is arrays. The data types are restricted to integers, booleans and enumeration types. Other data types (e.g. strings) must be constructed.
- There are some *library machines* that provide slightly more interesting data structures (e.g. sets, sequences)
- The built-in input/output facilities are restricted to terminal i/o ("standard in"/"standard out").

Do *not*

- select a project which needs algorithms requiring complicated mathematical reasoning to show correct!
- select too large a project. You should expect to write the program in traditional programming language in less than a hundred lines.

What to do this week (April 3-4)

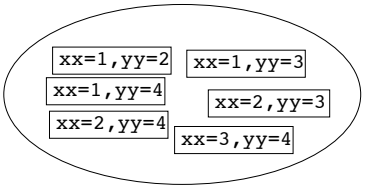
- Read the web site!
- Buy the textbook!
- Read chapters 1–4 of the textbook. Do the self-tests!!
- Form groups of 2 people.
- Start thinking about a project for your group.

- Did I mention that you should read the website?
  `http://www.it.uu.se/edu/course/homepage/bkp/vt08`

State spaces

Each set of variable values is a *state* of the machine.
The machine invariant puts restrictions on the *possible* (combinations of) values the machine variables can have.
The possible set of states is called the *state space* of the machine.

```
VARIABLES xx,yy
INVARIANT xx:1..3 & yy:1..4 & yy>xx
```



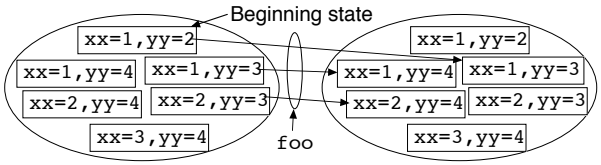
Operations

The AMN specification of an operation (an AMN *statement* or *generalised substitution*) describes how it changes the state.

Initialisations are also statements.

```
INITIALISATION xx,yy:=1,2
OPERATIONS
   foo = PRE yy<4 THEN yy:=yy+1 END
```

This operation is *enabled* only in a state where yy<4.



Consistency

We want to determine that a B machine is *consistent*. The most important things are that

- The initialisation *establishes* the invariant (the initialisation must put the machine in a valid state)
- Each operation *preserves* the invariant (after the operation the machine is still in a valid state)

If these conditions are all met, then the machine will never go outside its state space.

This is what we have to *prove*. How can we do it?

Predicates

A *predicate* is an AMN expression which states that something is true or false. Invariants and preconditions are predicates.

A predicate can be used to *describe a set of states*.

`yy<4` describes the set of states where the value if yy is less than 4.

The predicate P is *stronger than* Q if the set of states described by P is a subset of those described by Q (similarly *weaker*). This is the same as P implies Q (`P=>Q`). E.g. `yy<3` is stronger than `yy<4`.

A predicate which is always true is the weakest possible (describes all states).

A predicate which is always false is the strongest possible (describes no states).

Predicate transformers

Let the state of the machine be in the set of states described by some predicate (e.g. yy<4) when an operation is executed.

After the operation it will *generally* be in a different set described by a *different predicate*. Operations function as *predicate transformers*.

The operation `foo` will transform the predicate `yy<4` to `yy<5`.

The invariant is a predicate describing the entire state space. For a specification to be meaningful, all operations must *preserve* the invariant, that is they must transform the invariant into *itself* or into a *stronger* predicate.

Also the initialisation must transform the true predicate into the invariant (or again a stronger predicate).

Weakest preconditions

A predicate which holds after an operation is called a *postcondition*.

To ensure that an operation established a particular postcondition, the machine must generally be in a particular set of states (described by a *precondition*) before the operation is executed.

(The precondition is transformed into the postcondition.)

If we want the invariant (`xx:1..3&yy:1..4&xx<yy`) to hold after executing `foo`, then the predicate `xx:1..3&yy:0..3&xx<yy+1` (or a *stronger* predicate) must hold before executing `foo`.

`xx:1..3&yy:0..3&xx<yy+1` is called the *weakest precondition for* `foo` *to establish* `xx:1..3&yy:1..4&xx<yy`.

If `S` is an AMN statement and `Q` a postcondition, the weakest precondition is denoted by `[S]Q`.

Assignment

Assignment of a variable is written `x:=E`.
`[x:=E]Q` = `Q[E/x]` (`x` substituted for `E` in `Q`).

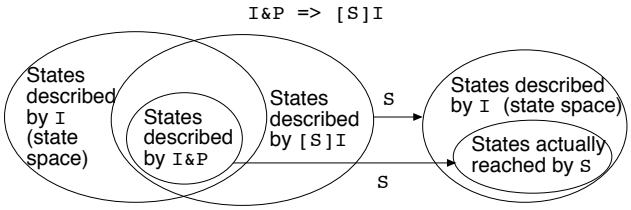
(`x` and `E` are not B variables but *syntactic* variables; they range over B variables/expressions. In that case, names have *one* character.)

Example: `[yy:=yy+1](xx:1..3&yy:1..4&xx<yy)` =
`= (xx:1..3&yy:1..4&xx<yy)[yy+1/yy]` =
`= (xx:1..3&yy+1:1..4&xx<yy+1)` =
`= (xx:1..3&yy:0..3&xx<yy+1)`

For multiple assignment we make simultaneous substitutions.
Example: `[xx,yy:=1,2](xx:1..3&yy:1..4&xx<yy)` =
`= (xx:1..3&yy:1..4&xx<yy)[1,2/xx,yy]` =
`= (1:1..3&2:1..4&1<2)` = *true*

Establishing the invariant

Before an operation is executed, the machine is in a state described by the conjunction `I&P` of the invariant (`I`) and the precondition (`P`) of the operation. To preserve the invariant, this predicate must be stronger than (or the same as) the precondition for the operation to establish `I`. If `S` is the AMN statement of the operation, then:



Proof obligations

In the case of `foo`, `I&P => [S]I` becomes:

```
(xx:1..3&yy:1..4&xx<yy)&yy<4 =>
[yy:=yy+1](xx:1..3&yy:1..4&xx<yy)
```

This is a *proof obligation* – something which we have to prove true – either using a tool such as Atelier B or by hand.

The initialisation also has to establish the invariant. Again, we get a proof obligation which is just `[S]I` where `S` is the initialisation statement.

As seen in a previous slide:

`[xx,yy:=1,2](xx:1..3&yy:1..4&xx<yy)` is always true

Some more AMN constructs

Do nothing: `skip`

`[skip]P = P`

Conditional: `IF E THEN S ELSE T END`

`[IF E THEN S ELSE T END]P = (E&[S]P)or(not(E)&[T]P)`

Note: The weakest precondition of a statement completely determines the effect of that statement. Thus the semantics of AMN — or any imperative programming language — can be defined by giving the weakest preconditions for all language constructs. This is called *weakest precondition semantics*.

What to do next week (April 7-11)

- Read chapters 5–9 of the textbook. Do the self-tests!!
- Try running Atelier B with the ”Hello, world” example from the course web site.
- Decide on a project for your group.

At the seminar on Monday (April 7) you should present your groups and initial thoughts for a project. Also be prepared to discuss chapters 1-4 of the book!

New types

AMN is a *strongly typed* language, although this is well hidden as predicates are used for type declarations (primarily the `:` predicate.)

New types can be introduced as sets using the `SETS` clause.

The elements of the set are either given by explicit enumeration or by the *implementation* (or intermediate refinement). In the latter case, the set is called a *deferred* set. Conditions on a deferred set can be given in a `PROPERTIES` clause, if needed.

```
SETS SS; STATUS = {ok, unknown, failure}  (Note ; !!!)
PROPERTIES card(SS) > 2
```

Here `SS` must be implemented by a set with more than 2 elements.

For consistency with parameters (below), it is a good idea to use names *without lower case letters*.

Don’t know, don’t care…

The specification determines what can and what can not be implemented.

Often, there are things we don’t know or don’t care about when writing a specification. In that case we want to leave things open so that we leave maximum freedom to fill in the gaps when writing implementations or using the specification in a specific situation.

In B this is handled

- for data: using *deferred sets*, *constants* and *parameters*.
- for operations: using *nondeterministic* statements

Constants

The specification can refer to values without knowing or caring exactly what they are.

It is left to the implementation to decide exactly what each constant is. (An intermediate refinement can also do this.)

The constant must be give a type in the `PROPERTIES` clause. Other conditions on the constants can also be given.

```
CONSTANTS numbers, kk
PROPERTIES numbers<:INT & kk:NAT & kk:numbers
```

In this case `numbers` must be eventuelly implemented by a subset of the (implementable) integers while `kk` must be implemented by a (implementable) natural number which is also a member of `numbers`.

Parameters

When a B specification machine is used as *part* (module) of a larger specification, the *larger specification* can define undetermined sets or scalar values. (Compare with *polymorphism* in prog. languages.)

Such sets/values are called *parameters* of the specification machine and are given in an ”argument list” after the machine name. Scalar value names must have at least one lower case letter – set names must not.

Any conditions on the parameters are given in a `CONSTRAINTS` clause. There must at least be a typing of scalar values.

```
MACHINE Foo(ELEMENTS,kk,maximum)
CONSTRAINTS kk:ELEMENTS & maximum:NAT
```

We’ll see later how parameters are defined.

Example

Part of a specification machine to handle course result registrations:

```
MACHINE Register(GRADE,top,limit)
CONSTRAINTS card(GRADE)>=2 & top:GRADE &
            limit:NAT1 & limit>2
SETS STUDENT; REPORT={OK,ERROR}; COURSE
CONSTANTS pm1, spp, maxreg
PROPERTIES pm1:COURSE & spp:COURSE &
           pm1/=spp & maxreg:NAT1 &
           card(COURSE) <= limit
```

(Note that `pm1/=spp` is necessary if you want `pm1` and `spp` to be different values!)

Proof obligations

Parameters, constants, defined and deferred sets give rise to new and changed proof obligations for machine consistency.

Let `p` be the parameters, `St` sets from the `SETS` clause, `k` constants from the `CONSTANTS` clause, `C` the constraints, `B` the properties and `I` the invariant. (`#` is the existential quantifier in ASCII notation.)

- `#p.C` (The constraints must be satisfiable)
- `C => #(St,k).B` (The properties must be satisfiable)
- `B&C => #v.I` (The invariant must be satisfiable)

Let T be the initialisation statement

- `B&C => [T]I` (The initialisation must establish the invariant)

Let `P` be the precondition and `S` the statement of an operation

- `B&C&I&P => [S]I` (The operation must preserve the invariant)

Nondeterminism

An operation which allows an arbitrary choice of different behaviours is called *nondeterministic*. A specification can be nondeterministic, but an implementation must always settle on *some* specific behaviour among the possible ones.

Nondeterministic statements:

- `ANY` (arbitary choice of *value*)
- `SELECT` (arbitrary choice of *statement*)
- `::` (nondeterministic assignment – special case of `ANY`)
- `CHOICE` (special case of `SELECT`)

Also, the `PRE` statement is in some sense an nondeterministic statement since an implementation is permitted an arbitrary behaviour if the precondition is not true.

`ANY` statement

```
ANY ee WHERE ee:SS THEN xx:=ee END
```

Picks an arbitrary `ee` such that `ee:SS` and executes `xx:=ee`.

(This particular `ANY` statement is so common that it has a shorthand: `xx::SS` – nondeterministic assignment)

Weakest precondition:

```
[ANY x WHERE Q THEN T END]P = !x.(Q=>[T]P)
```


`SELECT` statement

```
SELECT xx>1 THEN xx:=xx-1
  WHEN xx<4 THEN xx:=xx+1
  WHEN yy>1 THEN yy:=yy-1
  WHEN yy<4 THEN yy:=yy+1
END
```

Executes an arbitrary *branch* (statement after `THEN`) for which the *guard* (predicate before `THEN`) is true. An optional final `ELSE` branch is taken if no guards are true (refer to textbook for details.

(A `SELECT` with all guards true is so common that it has a shorthand: the `CHOICE` statement .)

Weakest precondition:

```
[SELECT Q1 THEN T1 WHEN Q2 THEN T2 …]P =
(Q1=>[T1]P) & (Q2=>[T2]P) & …
```


Relations and functions

Relations and *functions* are the main data structures of B machines.

They should be understood in the *mathematical* sense, i.e. as *sets of pairs* – in the case of relations pairs of related values, in the case of functions argument-value pairs. E.g.

`{(1,1),(2,4)}` or, alternatively, for functions `{1|->1,2|->4}`

A B function is *not* like a ”function” in a programming language which *computes* a value.

Constructing and using relations in B is a bit like construction and using a relational database.

Lambda abstractions

Function constants can also be written using *lambda abstraction*:

E.g. `%xx.(xx:NATURAL|xx*xx)` is the square function for natural numbers. (`%` is the ASCII notation for λ.)

A lambda abstraction *denotes* a (*possibly infinite*) *set* of pairs. E.g. `(9|->81):%xx.(xx:NATURAL|xx*xx)`. This means that definition by cases can be done by set union.
E.g. the absolute value function for integers can be expressed as:

```
%xx.(xx:INTEGER & xx>=0|xx) \/
 %xx.(xx:INTEGER & xx<0|-xx)
```

Recursion has to be done using *fixpoints* – tricky!
Note again that AMN is not a functional programming language, so unlike functional values (”anonymous functions”) in languages like ML, Haskell… AMN lambda abstractions are *not programs*.

Functional (relational) over-riding

A function (or general relation) can be *updated* by the overriding operator:

`f<+g` is the function (relation) obtained by taking the union of `g` and the parts of `f` which are outside the domain of `g`. I.e.

```
f<+g = (dom(f) <<|g)\/g
```

Example: `{1|->10,2|->20}<+{1|->30} = {1|->30,2|->20}`

The overriding operator is frequently used to update function (relation) variables.

Arrays

An array `aa` of 5 natural numbers with indices 1 through 5

1	2	3	4	5
18	12	5	12	10

can be naturally represented as the *function*

`aa: 1..5-->NAT`

(Actually, in a *specification*, the function need not be total. In an *implementation* it must be.)

Array indexing becomes a function call.

Array assignment is done by function overriding – to set element 3 to 2 use the AMN statement `aa:=aa<+{3|->2}`. This form is so common that it has a shorthand: `aa(3):=2`.

Traps with array assignment

Multiple assignment of arrays using the shorthand is not possible:

`aa(3),aa(4):=2,5` would mean

`aa,aa:=aa<+{3|->2},aa<+{4|->5}` which is not allowed

(conflicting assignment of the same variable).

Write `aa:=aa<+{3|->2,4|->5}` instead.

`[a(i):=e]P` is *not* `P[e/a(i)]`!
Witness: `[aa(3):=xx](aa(ii)=2)=(aa(ii)=2)` What if `ii=3`?

`[a(i):=e]P` *is* `P[a<+{i|->e}/a]`!
Witness: `[aa(3):=xx](aa(ii)=2)`
`= (aa<+{3|->xx}(ii)=2)`
`= (ii=3 & xx=2 or ii/=3 & aa(ii)=2))`

Sequences (lists)

A *sequence* (or *list*) `ss` of `n` values `x1,x2,x3,…,xn` where every element is in the set `E`, is written `[x1,x2,x3,…,xn]`.
(Note: In ASCII notation, the empty sequence is written <> !!)

The sequence is represented as a total function `1..n-->E`:

```
{1|->x1,2|->x2,3|->x3,…,n|->xn}
```

This means that a sequence behaves like an array – individual elements can be accessed and modified like array elements.

Sequence variables are usually declared to be members of the set `seq(E)` which denotes all total functions from `1..n` to the the elements in `E`, for every natural number `n`.

B has a large number of operations on sequences, including the traditional operations on lists – see the textbook.

What to do next week (April 14-18)

- Read chapters 10-11 of the textbook. Do the self-tests!!
- Start working on the specification for your project. Try out the specification in Atelier B (and ProB) early (as soon as you have something written).

At the seminar on Monday you should present your suggestion for a project. Also be prepared to discuss chapters 5–9 of the book!

Structuring machines

Just like programs, specifications are usually broken down into modules.

Advantages include

- Smaller units – easier to write and comprehend
- Reusing parts of specifications
- Simpler proof obligations as submachines can be proven independently of machines that use them.

Including machines

```
MACHINE M2
INCLUDES M1
.....
```

If M1 has parameters, actual values for these parameters must be given, e.g. `INCLUDES M1(NAT, 42)`.

Sets, constants and variables of M1 are available to M2 *as if* they were declared in M2. M2 can use the operations of M1.

The state of M1 becomes *part of* the state of M2. M1 essentially becomes *part of* M2. M1 can be said to be ”under the control” of M2.

Exception: M2 can *update* variables in M1 only using operations of M1. M1 still has its own invariant. Direct update of the variables of M1 could break it, while the operations of M1 are guaranteed not to.

Machine instances and renaming

As the state of M1 becomes *part of* the state of M2 when it is included by M2, a machine can only be included *once* by *one* other machine.

However, different *instances* (or copies) of a machine can be created by *renaming* and included in different places.

```
MACHINE M2
INCLUDES xx.M1,yy.M1
.....
```

Two instances of M1 called xx.M1 and yy.M1 are included by M2.

Other instances could be included by other machines.

Variables and operations of the instances are also renamed and are used by prefixing them with the instance prefix and a dot.

```
..... xx.counter > limit .....
```


Promotion

When M2 includes M1, operations of M1 can be used by M2 but do *not themselves* become operations of M2. The invariant of M2 could break if an operation of M1 was called without the knowledge of M1.

An operation of M1 can be *promoted* to *also* be an operation of M2. This adds a proof obligation to check that this operation preserves the invariant of M2.

```
MACHINE M1
OPERATIONS op = .....
```

```
MACHINE M2
INCLUDES M1
PROMOTES op
.....
```

All operations of M1 can be automatically promoted by writing `EXTENDS M1` rather than `INCLUDES M1` in M2.

`INCLUDE` proof obligations

When M2 includes M1, the p.o.’s of M2 will be changed as the properties and invariant of M1 can be taken to hold *beside* those of M2.

E.g. the p.o. that an operation `PRE P THEN S END` in M2 preserves the invariant becomes `C1&C2&B1&B2&I1&I2&P => [S]I2`, where `Bn`,`Cn`,`In` are the properties, constraints and invariants of Mn.

There is a *new* p.o. for M2, that any parameters passed to M1 satisfies the constraints of M1: `C2&B2 => C1`.

A statement in M2, calling an operation `op = PRE P THEN S END` in M1 has weakest precondition `[op]T = P&[S]T` (*design by contract*).

(It is assumed above that formal parameters in the `MACHINE` header and operation definitions of M1 are *replaced* by actual parameters in the `INCLUDES` clause or operation calls of M2.)

Parallell composition

In a specification machine, sequencing of statement (;) is *not permitted*. If several statements are needed for an operation (initialisation), they must be done in parallell instead: `S1||S2` (read ”S1 with S2”).

Note that this puts some rather strong constraints on how to express things in a specification.

```
op(xx) = PRE xx:NAT THEN yy:=yy+xx; zz:=yy END
```

is not allowed. Write

```
op(xx) = PRE xx:NAT THEN yy:=yy+xx || zz:=yy+xx END
```

instead. Or break out the common expression:

```
op(xx) = PRE xx:NAT THEN
         LET vv BE vv=yy+xx IN yy,zz:=vv,vv END
         END
```


Limitations on parallell composition

Two statements can not be done in parallell if they would change the same variable(s). (The same restriction as multiple assignment.)

Similarly, two operations of the same included machine can not be done in parallell as that could break the invariant of the included machine. (Even if they would not change the same variable!)

Example. What if `xx=1` and `yy=0`?

```
MACHINE M1
VARIABLES xx,yy
INVARIANT xx:NATURAL&yy:NATURAL& xx>=yy & xx<=yy+2
OPERATIONS op1 = xx:=yy+2;
          op2 = yy:=xx-2

MACHINE M2
INCLUDES M1
OPERATIONS foo = op1||op2
```

Weakest precondition of ||

`[S1||S2]T` can not be defined in terms of `[S1]T` and `[S2]T` (it is not *compositional*).

Instead it is defined in terms of rewrite rules which moves || *inwards* in S1 and S2, e.g.

```
(IF E THEN S11 ELSE S12)||S2 =
                    = IF E THEN S11||S2 ELSE S12||S2
(ANY X WHERE E THEN S1 END)||S2 =
                    = (ANY X WHERE E THEN S1||S2)
```

until only assignments are composed in parallell, then

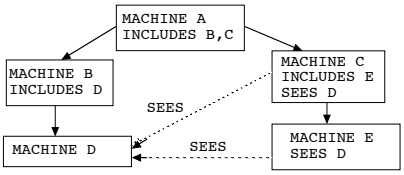
`[x1:=E1||x2:=E2]T = [x1,x2:=E1,E2]T = T[E1,E2/x1,x2]`

SEES

`INCLUDES` introduces a strict hierarchy among machines.

Often there is a need to use information in a machine outside the hierarchy, e.g. to keep definitions in one place.

The `SEES` relation between machines provides such access.



A and B can use information from D through the `INCLUDES` hierarchy. C and E must ”see” D to use that information.

What a machine can see

The `SEES` relation between machine provides less information than `INCLUDES`. If M2 sees M1, then M2 has access to:

- Sets and constants of M1 (*not* parameters).
- Variables (*except* in the invariant of M2).

The reason variables of M1 can not be used in the invariant of M2 is that M1 is not ”under the control” of M2 – the state of M1 can change independently of M2 – thus M2 can not ensure that an invariant using variables from M1 is maintained.

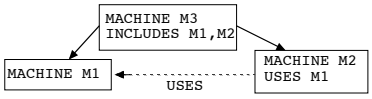
M1 is not considered a part of M2, so machines including M2 do not have access to M1 without themselves seeing M2.

When M2 sees M1, the proof obligations of M2 are changed in a similar way as for includes – except that some information is not available.

USES

Exceptionally, there is a need to let the invariant of a machine depend on the state of another machine which is not included. The `USES` relation is an extension of `SEES` which does allow this.

If M2 uses M1, then M2 can use variables of M1 in its own invariant. Some machine higher up in the includes hierarchy (e.g. M3 below) must ensure that the invariant of M2 is maintained. This becomes an extra proof obligation of M3!



The machines are static

A B machine has some similarity to a class/object of an object-oriented language:

- It has ”methods” (operations) and a local state.
- It can ”inherit” some data from another (seen/included) machine.
- Several instances of a machine can be created.

However, this similarity does not carry far

- There is no dynamic mechanism for creating machines (instances) The exact number and identity of machine instances is known in advance.
- There is no actual concept of inheritance.

Specification and implementation structuring

The structuring of the specification is independent of that of the implementation.

Each machine in the specification hierarchy can be separately implemented.

The whole or a part of the specification hierarchy can be implemented by a single implementation machine.

An implementation machine can itself use submachines in a hierarchy different from that of the specification.

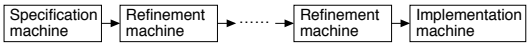
What to do this week (April 21-25)

- Read chapters 12-14 of the textbook. Do the self-tests!!
- Work on the specification for your project. Try out the specification in Atelier B (and ProB) early (as soon as you have something written).

At the seminar on Monday you should give a status report on your project (10 minutes per group or so). Also be prepared to discuss chapters 10-11 of the book!

Refinement

A specification is transformed into an implementation by a (sequence of) *refinement step(s)*.



Intuitively, each machine in the sequence ”does the same thing”, but each is more concrete than the previous one.

The number of refinement steps depend on the size of the ”conceptual gap” between the specification and implementation as well as the effort one is prepared to make to prove each individual refinement step.

An implementation machine is a special case of a refinement machine, which can be directly translated into a traditional programming language. In simple cases, the specification to be refined to an implementation in a single step.

New AMN constructs for refinements

`S;T`

Do S and T in sequence.

Weakest precondition: `[S;T]P = [S][T]P`

`VAR X IN S END`

`X` is a local variable in `S` (or a list of local variables).

Weakest precondition: `[VAR X IN S END]P = !X.[S]P`

`BEGIN S END`

Do `S`. The `BEGIN` and `END` simply work as ”parentheses” for `S`.

Weakest precondition: The same as for `S`.

These constructs can *not* be used in specification machines.

What is changed in a refinement?

Both data and operations can be refined.

The data representation can be changed to e.g.

- better suit a particular algorithm to be used in the implementation. (e.g. replace a sequence by an sorted sequence for binary search).
- be more ”implementable” (e.g. replace a finite set by a sequence or a sequence by an array).
- remove unused or redundant information (e.g. replace a finite set by its size if only the size is needed).

The operations can be changed to e.g.

- replace nondeterministic statements with deterministic ones.
- replace complex operations by sequences of simpler ones.
- replace operations over sets (e.g. quantification) with *loops* (only in implementation machines).

What is a refinement, precisely?

The refine*ment* machine (`R`) is a B machine with *exactly* the same operations as the refine*d* machine (`M`).

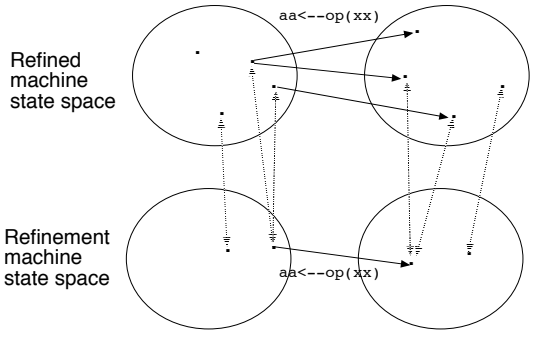
Every (valid) state of `R` must be *linked to* (at least) one (valid) state of `M`.

If the machines are in linked states, any operation of `R` must give the *same result* as the same operation of `M` in the sense that:

- The operations must return the *same* value.
- The machines must be in linked states *after* the operation.

If `M` is non-deterministic, it is sufficient that it can make *some* choice so that these conditions hold.
If several states of `M` are linked to a single state of `R`, then the conditions must hold for *all* those states.

What is a refinement (cont’d)?



How do we express refinement?

The *invariant* of the refinement machine (R) can make use of the state variables (and sets and constants) of the refined machine (M) to express the linking of states — the *linking invariant*.

The operation definitions of R should be independent of M, so the invariant is the *only* place where R can access the variables of M.

Example, the stack:

```
!ii.(ii:1..currentsize => stack(ii) = array(ii)) &
currentsize = size(stack)
```

A state in Stack with a particular sequence stack is linked to a state in StackI where the first currentsize number of elements in array are the same as those of stack. In this case, there are generally several states in StackI corresponding to a state in Stack.

How do we establish refinement?

```
MACHINE M                    REFINEMENT R
                             REFINES M
                             INVARIANT J
foo = PRE P THEN S END       foo = S1
```

The proof obligation of foo (and other operations) in R establish that R is a correct refinement of M.

The p.o. needs to state that

IF the states are linked and the precondition is satisfied (J&P holds)
AND S1 changes the state of R
THEN S must put M in *some* state so that J is again satified.

How do we express this?

The proof obligation for refined operations

```
MACHINE M                    REFINEMENT R
                             REFINES M
                             INVARIANT J
foo = PRE P THEN S END       foo = S1
```

[S]¬J characterises states from which it is certain that S will *not* put M into a state linked to the state of R (characterised by ¬J)

¬[S]¬J characterises states from which it is *not* certain that S will *not* put M into a linked state. S *can* put M into a linked state, somehow.

[S1]¬[S]¬J characterises states from which it is *certain* that S1 will put R in a state from which S *can* put M into a linked state.

This must hold assuming that we start in a linked state and that the precondition holds:

```
P&J => [S1]¬[S]¬J
```


Proof obligation example:

```
MACHINE Stack
   push(xx) = PRE xx:ELEMENTS & size(stack)<maxsize
              THEN stack := stack<-xx END;
IMPLEMENTATION StackI
REFINES Stack
INVARIANT ……currentsize = size(stack)
   push(xx) = BEGIN currentsize := currentsize+1;
                    array(currentsize) := xx END;
P&J => [currentsize:=currentsize+1;array(currentsize):=xx]
       ¬[stack := stack<-xx]¬currentsize=size(stack)
P&J => [currentsize:=currentsize+1;array(currentsize):=xx]
       ¬¬currentsize=size(stack<-xx)
P&J => [currentsize:=currentsize+1;array(currentsize):=xx]
       currentsize=size(stack<-xx)
P&J => [currentsize:=currentsize+1][array(currentsize):=xx]
       currentsize=size(stack<-xx)
P&J => [currentsize:=currentsize+1]currentsize=size(stack)+1
P&J => currentsize+1=size(stack<-xx)
P&…currentsize=size(stack)… => currentsize+1=size(stack<-xx)
```

This p.o. is true since size(stack<-xx)=size(stack)+1.

Refining nondeterminism

```
MACHINE M                    REFINEMENT R
                             REFINES M
INVARIANT xx:NAT             INVARIANT xx=yy
foo = ANY ii WHERE ii:1..5
      THEN xx:=xx+ii END     foo = yy:=yy+1
```

```
xx=yy=>[yy:=yy+1]¬[ANY…]¬xx=yy
xx=yy=>[yy:=yy+1]¬!ii.(ii:1..5=>[xx:=xx+ii]¬xx=yy)
xx=yy=>[yy:=yy+1]¬!ii.(ii:1..5=>¬xx+ii=yy)
xx=yy=>¬!ii.(ii:1..5=>¬xx+ii=yy+1)
```

The p.o. is true because:

```
xx=yy=>#ii.(ii:1..5&xx+ii=yy+1)
#ii.(xx=yy=>ii:1..5&xx+ii=yy+1)
xx=yy=>1:1..5&xx+1=yy+1            …which is true
```

Note that if R incorrectly did yy:=yy+6, then the p.o. would *not* be true.

What if the operation has output?

```
MACHINE M                    REFINEMENT R
                             REFINES M
                             INVARIANT J
xx<--foo = PRE P THEN S END  yy<--foo = S1
```

Add to J that xx=yy: P&J => [S1]¬[S]¬(J&xx=yy)

If the output variables have the same name in both machines, change the name in one of them to avoid conflict!

Loose ends…

```
MACHINE M                    REFINEMENT R
                             REFINES M
                             INVARIANT J
INITIALISATIONS T            INITIALISATIONS T1
```

The proof obligation of the initialisation works like an operation, although it we can not make any assumptions that the states are linked beforehand: `[T1]¬[T]¬J`

As usual, the proof obligations include the constraints and properties of `R` – and of `M` also, as the parameters, sets and constants of `M` are available to `R` – much like they would be if `R` had a `SEES M` clause.

You can not include (extend), see (use) a refinement – only a specification machine.

The example p.o. in Atelier B

```
"`Valuations'" &
maxsize = 100 & ELEMENTS = INT &
"`Previous components properties'" &
maxsize: INTEGER & 0<=maxsize & maxsize<=2147483647 &
ELEMENTS: FIN(INTEGER) & not(ELEMENTS = {}) &
"`Previous components invariants'" &
stack: seq(ELEMENTS) & size(stack)<=maxsize &
"`Component invariant'" &
array$1: 1..maxsize +-> ELEMENTS & dom(array$1) = 1..maxsize &
currentsize$1: 0..maxsize & currentsize$1 = size(stack) &
!ii.(ii: 1..currentsize$1 => stack(ii) = array$1(ii)) &
"`push preconditions in previous components'" &
xx: ELEMENTS & size(stack)+1<=maxsize &
"`push preconditions in this component'" &
"`Check that the invariant (currentsize=size(stack)) is preserved
by the operation - ref 4.4, 5.5'"
=>
currentsize$1+1 = size(stack<-xx)
```

The text items tell where the parts of the proof obligation come from. The `$1` suffix means that the operation has updated the variable.

A p.o. which is not proved automatically

```
"`Valuations'" &
maxsize = 100 & ELEMENTS = INT &
"`Previous components properties'" &
maxsize: INTEGER & 0<=maxsize & maxsize<=2147483647 &
ELEMENTS: FIN(INTEGER) & not(ELEMENTS = {}) &
"`Previous components invariants'" &
stack: seq(ELEMENTS) & size(stack)<=maxsize &
"`Component invariant'" &
array$1: 1..maxsize +-> ELEMENTS & dom(array$1) = 1..maxsize &
currentsize$1: 0..maxsize & currentsize$1 = size(stack) &
!ii.(ii: 1..currentsize$1 => stack(ii) = array$1(ii)) &
"`get preconditions in previous components'" &
not(stack = {}) &
"`get preconditions in this component'" &
"`Check that the invariant (xx$1 = xx) is preserved by the
operation - ref 4.4, 5.5'" &
"`Check operation refinement - ref 4.4, 5.5'"
=>
array$1(currentsize$1) = last(stack)
```

This is the proof obligation that the `get` operation returns the same value in both the specification and the refinement (implementation).

How to prove this p.o. by hand

Understand what the parts of the p.o. are!!

Identify parts relevant to the *goal* (the part after the assumptions).

```
"`Valuations'" &
maxsize = 100 & ELEMENTS = INT &
"`Previous components properties'" &
maxsize: INTEGER & 0<=maxsize & maxsize<=2147483647 &
ELEMENTS: FIN(INTEGER) & not(ELEMENTS = {}) &
"`Previous components invariants'" &
stack: seq(ELEMENTS) & size(stack)<=maxsize &
"`Component invariant'" &
array$1: 1..maxsize +-> ELEMENTS & dom(array$1) = 1..maxsize &
currentsize$1: 0..maxsize & currentsize$1 = size(stack) &
!ii.(ii: 1..currentsize$1 => stack(ii) = array$1(ii)) &
"`get preconditions in previous components'" &
not(stack = {}) &
"`get preconditions in this component'" &
"`Check that the invariant (xx$1 = xx) is preserved by the
operation - ref 4.4, 5.5'" &
"`Check operation refinement - ref 4.4, 5.5'"
=>
array$1(currentsize$1) = last(stack)
```


How to prove this p.o. (2)

The goal is: `array$1(currentsize$1) = last(stack)`

Take the assumption
`!ii.(ii:1..currentsize$1=>stack(ii)=array$1(ii))`
instantiate `ii` with `currentsize$1` to get
`currentsize$1:1..currentsize$1 =>`
`stack(currentsize$1)=array$1(currentsize$1))`
simplify (but see next slide!!!) to
`stack(currentsize$1)=array$1(currentsize$1)`

Use to rewrite the goal: `stack(currentsize$1)=last(stack)`

Use `currentsize$1 = size(stack)`
to rewrite the goal: `stack(size(stack))=last(stack)`

This is true by the definition of `last`.

Things are not always as easy as they seem

Is `currentsize$1:1..currentsize$1` always true?
What if `currentsize$1≤0`?? The the set will be empty!
We must prove that `currentsize$1≥1`… This is a new *subgoal*.
Identify new relevant assumptions:

```
maxsize = 100
currentsize$1: 0..maxsize
not(stack = {})
```

From `maxsize = 100` and `currentsize$1: 0..maxsize` we get `currentsize$1≥0` and `currentsize$1≤100`.
From `not(stack = {})` we get `size(stack)≠0`.
Since `currentsize$1=size(stack)` we get `currentsize$1≠0`.
Together with `currentsize$1≥0` we get `currentsize$1≥1`.

What to do next week (April 28 – May 2)

- Read chapters 15-17 of the textbook. Do the self-tests!!
- Work on the specification for your project. Try out the specification in Atelier B (and ProB) early (as soon as you have something written).

At the seminar on Monday you should give a status report on your project (10 minutes per group or so). Also be prepared to discuss chapters 12-14 of the book!

Implementations

Implementation machines are a special kind of refinement machines intended to represent a computer program.

Implementation machines are written in a restricted language (the B0 language).

The B0 language is a subset of B which can be translated in a straightforward way into a computer program in a standard language such as C or ADA.

In addition, B0 includes a few things not allowed in a specification or general refinement machine, notably *loops*.

The B0 language *differs in some important respects* between Atelier B and the B-Toolkit (the B implementation used in the textbook). Refer to these slides and to the course web site!

Data in implementations

Variables, constants, operation arguments and results must all be *concrete data*.

Concrete data are:

- Implementable integers – elements of `INT`.
- Elements of enumerated sets (including the predefined enumerated set `BOOL={TRUE,FALSE}`).
- Arrays of implementable integers or elements of enumerated sets. Array indices can be taken from an interval of implementable integers or be elements of enumerated sets.
  Formally, arrays are *total functions* (see next slide).
- Strings (can only be used as arguments of operations).

Arrays

Arrays are represented as total functions with a fixed, finite domain.

This domain can be an interval of implementable integers or an enumerated set or the cartesian product of any number of these sets.

Examples:

An integer array `aa` with 10 elements having indices 0 to 9 is represented as a function `aa:(0..9)-->INT`.

An 3x3 array `bb` of values in the range 0 to 20 with indices 1 to 3 is represented as a function `bb:(1..3)*(1..3)-->(0..20)`.

An 3 element natural number array `cc` with indices taken from the enumerated set `ANSWER={yes,maybe,no}` is represented as a function `cc:ANSWER-->NAT`.

Surjective, injective and bijective total functions can also be used.

Array expressions

Arrays are formally sets of ordered pairs.

In an implementation, only a few ways of computing such sets (*array expressions*) are allowed:

- The name of an array variable or array operation argument.
- An explicit set: `{I1|->V1, I2|->V2,…}` where the `In` are array indices and the `Vn` are the values stored in the corresponding position of the array. The `In` must be variables or constants, while the `Vn` can be arbitrary B0 expression. E.g. `{1|->4,2|->xx*2,3|->zz}` is an array of the numbers `4`, `xx*2` and `zz` indexed by 1 to 3.
- A cartesian product `D*{V}`, where `D` is a domain like the ones described in the previous slide and `V` is an arbitrary B0 expression. E.g. `(0..9)*{0}` is an array of all zeroes, indexed by 0 to 9.

Variables in implementations

State variables in implementations must be declared using the `CONCRETE_VARIABLES` clause. Such variables can only hold concrete data. The standard `VARIABLES` clause is not allowed.

```
IMPLEMENTATION Test
CONCRETE_VARIABLES vv
INVARIANT vv:NAT
```

This is a difference from the B-Toolkit which does not allow state variables at all in implementations. Instead, library machines are used to hold the implementation state. (This is certainly possible with Atelier B as well.)

Structuring implementations

An implementation can ”import” and ”see” another specification machine. `INCLUDES` is not allowed in an implementation.

`IMPORTS` is similar to `INCLUDES`. One important difference between them is that an imported machine must be implemented while an included machine need to be. There are other minor differences.

Note that a *specification* machine is imported – *not* its implementation. Only specification machines can be imported/included!

Each B project should have exactly one specification machine which is not included or imported (at the top of the includes/imports hierarchy). It should be implemented and have exactly one operation without arguments or return values. This operation will the one called to start the C program generated from the implementation machines.

Libraries

Atelier B includes library machines which can be imported into an implementation. They implement basic I/O and more complicated data structures: dynamic arrays, sorted arrays, sequences, functions etc. Using them can simplify your implementation and proof obligations.

Note that the Atelier B library machines are almost completely different from those of the B-Toolkit which are described in the textbook. Thus chapter 18 of the textbook can be used for the *ideas* presented there, but not as a concrete refence to library machines.

Refer to the ”Resusable Components Reference Manual”, available from the course web page using the Atelier B documentation link.

Sample library machine

`L_ARRAY5` implements a table with ordered values. Operations:

`VAL_ARRAY` value of an element.
`STR_ARRAY` set an element.
`SET_ARRAY` write the same value in part of the table.
`SWAP_ARRAY` exchange two elements.
`RIGHT_SHIFT_ARRAY` shift a portion to the large index.
`LEFT_SHIFT_ARRAY` shift a portion to the small index.
`SEARCH_MAX_EQL_ARRAY` search for a value in part of the table.
`SEARCH_MIN_EQL_ARRAY` search for a value in part of the table.
`REVERSE_ARRAY` invert the order of elements in part of the table.
`SEARCH_MIN_GEQ_ARRAY`
search for the first element greater than a value.
`ASCENDING_SORT_ARRAY` sort part of the table.

Instantiation of deferred sets and constants

Implementations must provide concrete values to any deferred sets and constant declared in a refined machine. The `VALUES` clause is used for this.

```
MACHINE Test
SETS FOO
CONSTANTS bar
PROPERTIES card(FOO)>2 & bar:FOO

IMPLEMENTATION TestI
REFINES Test
VALUES FOO=0..20; bar=2
```

Again, only concrete data can be used.
Note that in Atelier B the `PROPERTIES` clause is *not* used to provide values to deferred sets and constants.

Statements

Only the following statements are allowed in implementations:

- Block (`BEGIN…END`)
- Local variable statement (`VAR`)
- Do nothing (`skip`)
- Assignment (`:=`)
- Operation call
- `IF`
- `CASE`
- Sequence (`;`)
- While loop (see next slide).

An oddity of B0 is that the test of an `IF` statement can only compare variables and constants – tests such as `2*xx>0` are not allowed! Similarly, a `CASE` can only branch on a variable.

Overflow checks

In an implementation, weakest preconditions will be strengthened to guarantee that arithmetic operations do not overflow.

E.g. the weakest precondition of the statement

`xx:=yy*zz`

will include the condition `yy*zz:INT`.

(A likely reason for the reason for the ”oddity” described in the previous slide is that only statements have weakest preconditions – not expressions or predicates – so there is no obvious place to put the condition generated by a test such as `2*xx>0`.)

Loops

B0 includes while loops:

`WHILE E DO S INVARIANT I VARIANT V END`

The `WHILE` statement behaves just like the `WHILE` loop of any programming language. `E` and `S` are the loop test and body.

`I` is the *loop invariant*, a predicate which must be true at the beginning of the loop and after every execution of the loop body. The loop invariant is crucial in defining the weakest precondition of `WHILE`.

`V` is the *loop variant*, a numeric expression which is strictly decreasing at every iteration of the loop body and which can not be less than 0. The loop variant guarantees that the loop will terminate.

Just like the test in an `IF` statement, the loop test can only compare variables and constants.

Loop invariants

A problem with determining the weakest precondition `[WHILE E do S…]P` is that the number of iterations is not known in advance.

If the number of iterations was known to be zero, the weakest precondition would simply be `P`. If one, it would be `[S]P`, if two it would be `[S][S]P` etc.

Suppose that there is some predicate `I` such that `I=>P` and the loop body preserves `I`, i.e. `I=>[S]I`. In that case `[WHILE…]P` could be `I`!

If I holds before the loop, it will continue to hold with every iteration of the loop. When the loop stops, `I` will still hold and `P` will now hold because `I=>P`.

Such a predicate `I` is called a *loop invariant*.

Weakest precondition of a loop

`[WHILE E DO S INVARIANT I VARIANT V END]P`
consists of the conjunction (&) of five parts.

We use the fact that `E` is true in the loop body and false after the loop. `l` is the variable(s) which are assigned new values in the loop body.

- The loop body preserves the invariant: `!l.(I&E=>[S]I)`
- The loop establishes `P` on exit: `!l.(I&¬E=>P)`
- The variant is a natural number (i.e. ≥0): `!l.(I&E=>v:NATURAL)`
- The loop body must decrease the variant:
```
!(l,g).(I&E&v=g=>
          [S](v<g)
```
- The invariant must hold before the loop starts: `I`

Loop example

```
MACHINE Exp
OPERATIONS rr<--exp(bb,ee) =
  PRE bb:NAT&ee:NAT&bb**ee:NAT
  THEN rr:=bb**ee END
END
```

```
IMPLEMENTATION ExpI
REFINES Exp
OPERATIONS pp<--exp(bb,ee) =
  VAR kk IN
    pp:=1; kk:=ee;
    WHILE kk>0 DO
      pp:=pp*bb; kk:=kk-1
      INVARIANT pp=bb**(ee-kk)&ee>=kk&kk:NAT
      VARIANT kk
    END
  END
END
```


The refinement proof obligation

```
bb:NAT&ee:NAT&bb**ee:NAT=>
   [VAR…]¬[rr:=bb**ee]¬pp=rr
bb:NAT&ee:NAT&bb**ee:NAT=>
   [VAR…]¬¬pp=bb**ee
bb:NAT&ee:NAT&bb**ee:NAT=>
   [VAR…]pp=bb**ee
bb:NAT&ee:NAT&bb**ee:NAT=>
   !kk.([pp:=1;kk:=ee;WHILE…]pp=bb**ee
bb:NAT&ee:NAT&bb**ee:NAT=>
   !kk.([pp:=1][kk:=ee][WHILE…]pp=bb**ee
```

What is `[WHILE…]pp=bb**ee`?

Weakest precondition of the example loop

What is the weakest precondition of this loop to establish `pp=bb**ee`?

Let `I` be `pp=bb**(ee-kk)&ee>=kk&kk:NAT`

- The loop body preserves the invariant:
  `!(pp,kk).(I&kk>0=>[pp:=pp*bb;kk:=kk-1]I)`
- The loop establishes `pp=bb**ee` on exit:
  `!(pp,kk).(I&¬kk>0=>pp=bb**ee)`
- The variant is a natural number (i.e. ≥0):
  `!(pp,kk).(I&kk>0=>kk:NATURAL)`
- The loop body must decrease the variant:
  `!(pp,kk,g).(I&kk>0&kk=g=>[pp:=pp*bb;kk:=kk-1](kk<g)`
- The invariant must hold before the loop starts:
  `pp=bb**(ee-kk)&ee>=kk&kk:NAT`

Weakest precondition of example (part 1)

Calculate the first part (the loop body preserves the invariant):

```
!(pp,kk).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&kk>0
          =>[pp:=pp*bb;kk:=kk-1]
            pp=bb**(ee-kk)&ee>=kk&kk:NAT)
!(pp,kk).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&kk>0
          =>pp*bb=bb**(ee-(kk-1))&ee>=kk-1&
            kk-1:NAT&pp*bb:INT)
```

Note that `pp*bb:INT` is in the weakest precondition of `pp:=pp*bb` to show that the multiplication does not overflow. (Also `kk-1:INT` is in the weakest precondition of `kk:=kk-1` for similar reason but this particular condition is redundant as the statement needs to establish `kk:NAT` anyway.) Continued....

Weakest precondition of example (part 1 cont'd)

```
!(pp,kk).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&kk>0
          =>pp*bb=bb**(ee-(kk-1))&ee>=kk-1&
            kk-1:NAT&pp*bb:INT)
```

Now,

`pp*bb=bb**(ee-(kk-1))` because `pp=bb**(ee-kk)` and `ee>=kk`

`ee>=kk-1` because `ee>=kk`

`kk-1:NAT` because `kk:NAT` and `kk>0`

`pp*bb:INT` because `bb**ee:NAT` (precondition!),
`bb**(ee-(kk-1))<=bb**ee` (as `kk-1>=0`) and
`pp*bb=bb**(ee-(kk-1))`

So this part is always true.

Weakest precondition of example (part 2)

Calculate the second part (the loop establishes `pp=bb**ee` on exit):

```
!(pp,kk).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&¬kk>0
          =>pp=bb**ee)
```

Now,

`kk=0` because `kk:NAT` and `¬kk>0`

so `pp=bb**ee` because `pp=bb**(ee-kk)` and `kk=0`

So this part is always true.

Weakest precondition of example (parts 3-4)

Calculate the third part (the variant is a natural number (i.e. ≥0)):

```
!(pp,kk).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&kk>0
          =>kk:NATURAL)
```

Now, `kk:NATURAL` because `kk:NAT`

So this part is always true.

Calculate the fourth part (the loop body must decrease the variant):

```
!(pp,kk,g).(pp=bb**(ee-kk)&ee>=kk&kk:NAT&kk>0&kk=g
            =>[pp:=pp*bb;kk:=kk-1](kk<g)
!(pp,kk,g).(pp=bb**(ee-kk)&ee>=kk&
            kk:NAT&bb**ee:NAT&kk>0&kk=g
            =>kk-1<g
```

Now, `kk-1<g` because `kk=g`

So this part is always true.

Weakest precondition of example (part 5)

Calculate the fifth part (the invariant must hold before the loop starts):

```
pp=bb**(ee-kk)&ee>=kk&kk:NAT
```

This can not be simplified.

The weakest precondition of the loop is exactly this predicate as all other parts were always true.

The refinement proof obligation again

```
bb:NAT&ee:NAT&bb**ee:NAT=>
   !kk.([pp:=1][kk:=ee][WHILE…]pp=bb**ee
bb:NAT&ee:NAT&bb**ee:NAT=>
 !kk.([pp:=1][kk:=ee]pp=bb**(ee-kk)&ee>=kk&kk:NAT)
bb:NAT&ee:NAT&bb**ee:NAT=>
 !kk.([pp:=1]pp=bb**(ee-ee)&ee>=ee&ee:NAT)
bb:NAT&ee:NAT&bb**ee:NAT=>
 !kk.(1=bb**(ee-ee)&ee>=ee&ee:NAT)
bb:NAT&ee:NAT&bb**ee:NAT=>
 !kk.(ee:NAT)
bb:NAT&ee:NAT&bb**ee:NAT=>
 ee:NAT
```

So the implementation is correct!

A sample p.o. in Atelier B

```
    "`exp preconditions in previous components'" &
    bb: INTEGER & 0<=bb & bb<=2147483647 &
    ee: INTEGER & 0<=ee & ee<=2147483647 &
    bb**ee: INTEGER & 0<=bb**ee & bb**ee<=2147483647 &
    "`exp preconditions in this component'" &
    "`Local hypotheses'" &
    kk<=ee &
    kk: INTEGER & 0<=kk & kk<=2147483647 &
    1<=kk &
    "`Check preconditions of called operation, or While loop
construction, or Assert predicates'"
    =>
    kk-1+1<=kk
```

This is the check that the variant decreases. Atelier B rewites inequalities to use <=, so the goal `kk-1+1<=kk` really is `kk-1<kk`.

Another sample p.o. in Atelier B

```
    "`exp preconditions in previous components'" &
    bb: INTEGER & 0<=bb & bb<=2147483647 &
    ee: INTEGER & 0<=ee & ee<=2147483647 &
    bb**ee: INTEGER & 0<=bb**ee & bb**ee<=2147483647 &
    "`exp preconditions in this component'" &
    "`Local hypotheses'" &
    not(1<=kk$7777) &
    kk$7777<=ee &
    kk$7777: INTEGER & 0<=kk$7777 & kk$7777<=2147483647 &
    "`Check that the invariant (pp$1 = pp) is preserved by the
operation - ref 4.4, 5.5'" &
    "`Check operation refinement - ref 4.4, 5.5'"
    =>
    bb**(ee-kk$7777) = bb**ee
```

This is the check that after the loop has terminated, the value of `pp` – which is `bb**(ee-kk$7777)` – is equal to the value required by the specification – which is `bb**ee`.
The `$7777` suffix is used to indicate that it is the value of the variable after the loop exit.

How to find the loop invariant?

- Try to express what the loop achieves during its execution in terms similar to that of the desired result. E.g. our example program should establish that `pp=bb**ee`. The loop repeatedly multiplies `bb` into `pp` – i.e. computes a power. That power is `bb**(ee-kk)`, so try `pp=bb**(ee-kk)` as (part of) the invariant.
- If a part of the loop p.o. can not be proved, look for what assumptions would be needed to prove them. Maybe they should be part of the loop invariant. E.g. to prove `pp*bb=bb**(ee-(kk-1))` from `pp=bb**(ee-kk)` in *integer* arithmetic, we must know that `ee>=kk`. This is because if `ee=kk-1` and `bb>1` then `bb**(ee-kk)= bb**(-1) = 0 ≠ 1 = bb**0 = bb**(ee-(kk-1))`
- Many other good suggestions are given in the textbook!

What to do next…

- Complete your specifications!
- Start working on your implementation (and possibly intermediate refinements).

On Tuesday (April 6) next week, we will discuss chapters 15-17 of the book!

On Wednesday (April 7), each group should give a presentation (15 minutes each) of their specification.

After that, there will be no more meetings of the whole class until the final seminar (June 2) where all projects should be presented (20-25 minutes each).