

An Introduction to the B Method

Machine Composition

Revision: 1.2, April 1, 2003

©Ken Robinson

[mailto::k.robinson@unsw.edu.au](mailto:k.robinson@unsw.edu.au)

Objectives of this lecture

- To introduce the constructs offered by B for composing machines.
- To discuss the rules and restrictions imposed by the different composition constructs.
- To give some examples of the use of each composition construct.

INCLUDES

The *INCLUDES* mechanism allows a machine to be embedded in another machine with read/write access to the variables of the included machine.

MACHINE A(formal-params-A)

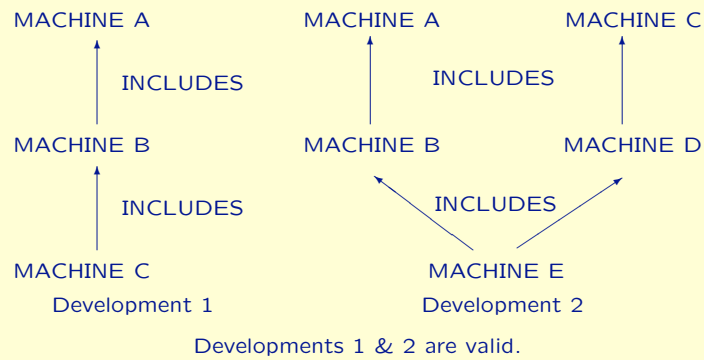
MACHINE B(formal-params-B) *MACHINE C*(formal-params-C)

INCLUDES A(actual-params-A) *INCLUDES B*(actual-params-B)

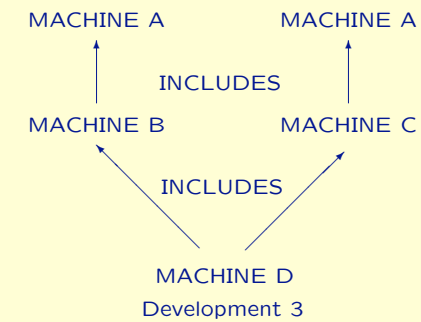
- The state of machine *A* is prepended to the state of machine *B*.
- Machine *B* has full access to the constants, sets, variables and operations of *A*.
- The state of machine *A* can be modified by machine *B*, but only by using operations of *A*.
- Within a single development, any single machine may be included only once.

- INCLUDES* is transitive: machine *A* is also included in machine *C*.
- The operations of an included machine are *not* automatically exported by the including machine. See *PROMOTES* and *EXTENDS*.
- The parameters of an included machine must be instantiated at the point of inclusion. Often the instantiation is to parameters of the including machine.

Include Once Only



Include Once Only



Machine Renaming

Different instantiations of a machine may be created by renaming using the following dot notation:

machine – name.modifier

All the variables and operations of the machine are similarly qualified, *but not the sets and constants*.

PROMOTES

PROMOTES is used in the including machine to promote an operation of the *included* machine to the interface of the *including* machine.

all operations of the included machine are available for the including machine to use, but they do not automatically become operations in the interface of that machine. If machine *B* *INCLUDES* machine *A*, then without promotion the only operations in the interface of machine *B* will be the operations defined in *B*.

EXTENDS

EXTENDS is used in place of *INCLUDES* to include a machine and promote *all* operations of the included machine.

If machine *B* *EXTENDS* machine *A* then the interface of the new machine will contain all the operations defined in *both* *A* and *B*.

SEES

SEES provides readonly access to the *SETS*, *CONSTANTS* and *VARIABLES* of the seen machine.

- A machine may be seen by any number of machines in a development.
- *SEES* is *not* transitive.
- The variables of a seen machine may be referenced in the seeing machine, *but not in the invariant*.
- Operations of the seen machine that do not change the seen machine's state may be used by the seeing machine. Such operations provide mathematical functions.
- The parameters of a seen machine are *not* instantiated by the seeing machine.
- A seen machine must be *implemented*, and *imported* into the development.

Comparison between INCLUDES and SEES

<i>INCLUDES</i>	<i>SEES</i>
read/write access	readonly access
parameters must be instantiated	parameters must not be instantiated
all operations accessible	only functional operations accessible
transitive	not transitive
may not be implemented	must be implemented

USES

USES provides read-only access by the using machine to the variables of the used machine.

- The using machine may reference the variables of the used machine in its invariant. This is sometimes convenient, but is insecure and gives rise to the following requirement.
- The used and using machines must be included in a larger machine.

Comparison between SEES and USES

	<i>SEES</i>	<i>USES</i>
access	readonly	readonly
parameters	must not be instantiated	must not be instantiated
variables	not referenced in invariant	may be referenced in invariant
operations	only functional accessible	only functional accessible
	not transitive	not transitive
	must be implemented	cannot be implemented

Inconsistent use of operations . . .

It is clear that the substitution $xx, xx := xx + 1, xx - 1$ is inconsistent. Thus B insists that the variables on the lhs of a multiple substitution are disjoint. Of course, $xx, xx := xx + 1, xx - 1$ is equivalent to $xx := xx + 1 \parallel xx := xx - 1$, and concurrently composed substitutions must modify different components of the state.

Any single operation of a machine that changes the state will generate a proof obligation that the operation restores the invariant. Proof of this obligation assumes that the only changes are those made by this operation. Composing any two operations in parallel, each of which independently maintains the invariant, can be expected, in general, to produce inconsistent state changes.

Inconsistent use of operations

For the above reasons the following restrictions apply to use of operations:

1. Although a machine B may have access to the operations of another machine A , for example via *INCLUDES*, machine B may not compose two or more operations of A concurrently.
2. Operations of a single machine may not be referenced within other operations of the same machine. If this is desired, then it can be achieved, when it is safe, by splitting the machine into two, including one machine in the other and using the operation of the included machine in the including machine.

Notice that splitting a machine into two machines demonstrates that the original single machine contains two independent sub-states, and operations on each of those sub-states cannot interfere.

Global Sets, Constants and Variables

In the simple library case study we needed to have access to a global set *BOOK*, the set of all books. We used a machine parameter, but there was a footnote saying that this was not the correct way of representing a global set.

The reason is that machine parameters are instantiated at the point where the machine is introduced into a development. Thus the headers

MACHINE A(BOOK) MACHINE B(BOOK)

do not guarantee that both A and B see the same set. Parameter names are dummy names and do not imply anything about the identity of the actual values used to instantiate the parameter.

The correct way to represent a universal set is to embed it in a machine, for example

```

MACHINE Book_TYPE ( maxbook )
CONSTRAINTS maxbook  $\in \mathbb{N}_1$ 
SETS BOOK
PROPERTIES card ( BOOK ) = maxbook
END

```

and to see that machine using *SEES*.

A Date Machine

```

MACHINE Date ( maxDate )
CONSTRAINTS maxDate  $\in \mathbb{N}_1$ 
VARIABLES today
INVARIANT today  $\in DATE$ 
INITIALISATION today : $\in DATE$ 

```

OPERATIONS

```

newday  $\hat{=}$ 
  PRE today  $\neq maxDate$ 
  THEN today := today + 1
  END

```

```

DEFINITIONS DATE  $\hat{=}$  0 .. maxDate
END

```

A Time Machine

```

MACHINE Time
SEES Bool_TYPE
SETS TIME
CONSTANTS
  FUTURE ,
  PAST ,
  BigCrunch ,
  BigBang

```

PROPERTIES

The future, modelled by *FUTURE*, is a relation that relates any time to all times in its future.

$$FUTURE \in TIME \leftrightarrow TIME \wedge$$

The past, modelled by *PAST*, is simply the inverse of the future.

$$PAST = FUTURE^{-1} \wedge$$

The relation *FUTURE* is total: any two unequal times t_1 and t_2 must be related in *FUTURE*.

$$\forall (t_1 , t_2) . (t_1 \in TIME \wedge t_2 \in TIME \wedge t_1 \neq t_2 \Rightarrow t_1 \mapsto t_2 \in FUTURE \vee t_2 \mapsto t_1 \in FUTURE) \wedge$$

FUTURE is irreflexive: no time is in its own future. We model this by saying that *FUTURE* and the identity relation on *TIME*, *id(TIME)*, are disjoint.

$$FUTURE \cap id(TIME) = \{\}$$

FUTURE is anti-symmetric: if t_2 is in the future of t_1 then t_1 is not in the future of t_2 .

$$FUTURE \cap PAST = \{\}$$

FUTURE is transitive, if t_2 is in the future of t_1 and t_3 is in the future of t_2 then t_3 is in the future of t_1

$$\forall (t_1, t_2, t_3). (t_1 \in TIME \wedge t_2 \in TIME \wedge t_3 \in TIME \wedge (t_2 \in FUTURE[\{t_1\}] \wedge t_3 \in FUTURE[\{t_2\}]) \Rightarrow t_3 \in FUTURE[\{t_1\}]) \wedge$$

The following cosmology is concerned with time in a finite universe.

There are two extreme times, *BigBang* and *BigCrunch*
 $BigBang \in TIME \wedge BigCrunch \in TIME \wedge$

they are not equal

$$BigBang \neq BigCrunch \wedge$$

no time is in the future of *BigCrunch*,

$$BigCrunch \notin \text{dom}(FUTURE) \wedge$$

and no time is in the past of *BigBang*.

$$BigBang \notin \text{dom}(PAST) \wedge$$

All times, except *BigBang*, are in the future of *BigBang*,

$$FUTURE[\{BigBang\}] = TIME - \{BigBang\} \wedge$$

and all times, except *BigCrunch*, are in the past of *BigCrunch*.

$$PAST[\{BigCrunch\}] = TIME - \{BigCrunch\}$$

The machine state has one variable, *currenttime*, that records the last time given by an operation of this machine.

VARIABLES *currenttime*

INVARIANT

$$currenttime \in TIME$$

INITIALISATION *currenttime* := *TIME*

OPERATIONS

Operation Clock gives a time that is in the future of the current value of *currenttime*.

time ← Clock ≐

PRE *currenttime* ≠ *BigCrunch*

THEN ANY *now*

WHERE *now* ∈ *FUTURE* [{ *currenttime* }]

THEN *time* := *now* || *currenttime* := *now*

END

END ;

Operation Tick returns the next time tick after the current time. This is modelled as the "least" time in the future of *currenttime*.

```
time ← Tick ≐  
PRE currenttime ≠ BigCrunch  
THEN ANY now  
  WHERE now ∈ FUTURE [ { currenttime } ] ∧  
        FUTURE [ { currenttime } ] ∩ PAST [ { now } ] = {}  
  THEN time := now || currenttime := now  
END  
END ;
```

Operation Later(*t1*,*t2*) returns TRUE if *t1* ∈ FUTURE[{*t2*}] and FALSE otherwise.

```
later ← Later ( t1 , t2 ) ≐  
PRE t1 ∈ TIME ∧ t2 ∈ TIME  
THEN later := bool ( t1 ∈ FUTURE [ { t2 } ] )  
END ;
```

Operation Earlier(*t1*,*t2*) returns TRUE if *t1* ∈ PAST[{*t2*}] and FALSE otherwise.

```
earlier ← Earlier ( t1 , t2 ) ≐  
PRE t1 ∈ TIME ∧ t2 ∈ TIME  
THEN earlier := bool ( t1 ∈ PAST [ { t2 } ] )  
END  
END
```