Outline I

1 INCLUDES
   - Include Once Only
   - PROMOTES
   - EXTENDS
   - SEES
   - Comparison between INCLUDES and SEES
   - USES
   - Comparison between SEES and USES
   - Inconsistent use of operations
   - Global Sets, Constants and Variables

2 Examples
   - A Date Machine
   - Example: A Time Machine
Objectives of this lecture

To introduce the constructs offered by B Method (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.
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.
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.
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.
The *INCLUDES* mechanism allows a machine to be embedded in another machine with read/write access to the variables of the included machine.

\[\text{MACHINE A(formal-params-A)}\]
\[\text{MACHINE B(formal-params-B)}\]
\[\text{INCLUDES A(actual-params-A)}\]
\[\text{MACHINE C(formal-params-C)}\]
\[\text{INCLUDES B(actual-params-B)}\]

- 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.
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.
The \textit{INCLUDES} mechanism allows a machine to be embedded in another machine with read/write access to the variables of the included machine.

\begin{verbatim}
MACHINE A(formal-params-A)
MACHINE B(formal-params-B)
INCLUDES A(actual-params-A)
\end{verbatim}

- 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.
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)
INCLUDES A(actual-params-A)

MACHINE C(formal-params-C)
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.
The *INCLUDES* mechanism allows a machine to be embedded in another machine with read/write access to the variables of the included machine.

\[
\text{MACHINE } A(\text{formal-params-A}) \\
\text{MACHINE } B(\text{formal-params-B}) \quad \text{MACHINE } C(\text{formal-params-C}) \\
\text{INCLUDES } A(\text{actual-params-A}) \quad \text{INCLUDES } B(\text{actual-params-B})
\]

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

\[
\text{MACHINE } A(\text{formal-params-A}) \\
\text{MACHINE } B(\text{formal-params-B}) \quad \text{MACHINE } C(\text{formal-params-C}) \\
\text{INCLUDES } A(\text{actual-params-A}) \quad \text{INCLUDES } B(\text{actual-params-B})
\]

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

\[
\begin{align*}
\textit{MACHINE A} & (\text{formal-params-A}) \\
\textit{MACHINE B} & (\text{formal-params-B}) \\
\textit{INCLUDES A} & (\text{actual-params-A}) \\
\textit{MACHINE C} & (\text{formal-params-C}) \\
\textit{INCLUDES B} & (\text{actual-params-B})
\end{align*}
\]

- 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.
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)**  
**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.
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.
- 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.
Development 1

Development 2

Developments 1 & 2 are valid.
Development 1

MACHINE A 
  INCLUDES 
  MACHINE B 
  INCLUDES 
  MACHINE C

Development 2

MACHINE A 
  INCLUDES 
  MACHINE B 
  INCLUDES 
  MACHINE C

MACHINE B 
  INCLUDES 
  MACHINE C

MACHINE D 
  INCLUDES 
  MACHINE E

MACHINE C 
  INCLUDES 
  MACHINE B 
  INCLUDES 
  MACHINE A

MACHINE E

Developments 1 & 2 are valid.
Development 1

Development 2

Developments 1 & 2 are valid.
Developments 1 & 2 are valid.
Development 3 is invalid, as machine A is included twice.

Within a single development, any single machine may be included only once.
Development 3 is invalid, as machine A is included twice.

Within a single development, any single machine may be included only once.
Development 3 is invalid, as machine A is included twice.

Within a single development, any single machine may be included only once.
Development 3 is invalid, as machine A is included twice.

Within a single development, any single machine may be included only once.
Machine Renaming

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

```
machinename.modifier
```

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

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

\[ \text{machinename}.\text{modifier} \]

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

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

\[ \text{machinename} \cdot \text{modifier} \]

All the variables and operations of the machine are similarly qualified, \textit{but not the sets and constants}. 
PROMOTES is used in the including machine to promote an operation of the included machine to the interface of the including machine.

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

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

Clarification: 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 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*. 
**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*. 
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 provides readonly access to the \textit{SETS}, \textit{CONSTANTS} and \textit{VARIABLES} of the seen machine.

- A machine may be seen by any number of machines in a development.
- \textsc{SEES} is not transitive: if $B \textsc{SEES} A$, and $C \textsc{INCLUDES} or \textsc{SEES} B$, then $C$ does not automatically see $A$.
- 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 \textit{not} instantiated by the seeing machine.
- A seen machine must be \textit{implemented}, and \textit{imported} into the development.
SEESE 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.
- SEESE is not transitive: if B SEESE A, and C INCLUDESE or SEESE B, then C does not automatically see A.
- 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.
SEES provides readonly access to the \textit{SETS}, \textit{CONSTANTS} and \textit{VARIABLES} of the seen machine.

- A machine may be seen by any number of machines in a development.
- \textit{SEES} is \textit{not} transitive: if B \textit{SEES} A, and C \textit{INCLUDES} or \textit{SEES} B, then C does not automatically see A.
- The variables of a seen machine may be referenced in the seeing machine, \textit{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 \textit{not} instantiated by the seeing machine.
- A seen machine must be \textit{implemented}, and \textit{imported} into the development.
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: if B SEES A, and C INCLUDES or SEES B, then C does not automatically see A.
- 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.
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: if B SEES A, and C INCLUDES or SEES B, then C does not automatically see A.
- 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.
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: if B SEES A, and C INCLUDES or SEES B, then C does not automatically see A.
- 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.
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: if B SEES A, and C INCLUDES or SEES B, then C does not automatically see A.
- 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.
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: if B SEES A, and C INCLUDES or SEES B, then C does not automatically see A.
- 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**

<table>
<thead>
<tr>
<th><strong>INCLUDES</strong></th>
<th><strong>SEES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>may not be implemented</td>
<td>must be implemented</td>
</tr>
</tbody>
</table>
Comparison between *INCLUDES* and *SEES*

<table>
<thead>
<tr>
<th></th>
<th>INCLUDES</th>
<th>SEES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>Parameters</td>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>Operations</td>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>Transitivity</td>
<td>transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>Implementation</td>
<td>may not be implemented</td>
<td>must be implemented</td>
</tr>
</tbody>
</table>
## Comparison between `INCLUDES` and `SEES`

<table>
<thead>
<tr>
<th></th>
<th><code>INCLUDES</code></th>
<th><code>SEES</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>Parameters</td>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>Operations</td>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>Transitivity</td>
<td>transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>Implementation</td>
<td>may not be implemented</td>
<td>must be implemented</td>
</tr>
<tr>
<td><strong>INCLUDES</strong></td>
<td><strong>SEES</strong></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>read/write access</td>
<td>readonly access</td>
<td></td>
</tr>
<tr>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
<td></td>
</tr>
<tr>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
<td></td>
</tr>
<tr>
<td>transitive</td>
<td>not transitive</td>
<td></td>
</tr>
<tr>
<td>may not be implemented</td>
<td>must be implemented</td>
<td></td>
</tr>
</tbody>
</table>
## Comparison between **INCLUDES** and **SEES**

<table>
<thead>
<tr>
<th><strong>INCLUDES</strong></th>
<th><strong>SEES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>may not be implemented</td>
<td>must be implemented</td>
</tr>
</tbody>
</table>

**Global Sets, Constants and Variables**

- Comparison between INCLUDES and SEES
- Inconsistent use of operations
## Comparison between **INCLUDES** and **SEES**

<table>
<thead>
<tr>
<th><strong>INCLUDES</strong></th>
<th><strong>SEES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>transitive</td>
<td>not transitive</td>
</tr>
</tbody>
</table>

- may not be implemented
- must be implemented
# Comparison between INCLUDES and SEES

<table>
<thead>
<tr>
<th>INCLUDES</th>
<th>SEES</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write access</td>
<td>readonly access</td>
</tr>
<tr>
<td>parameters must be instantiated</td>
<td>parameters must not be instantiated</td>
</tr>
<tr>
<td>all operations accessible</td>
<td>only functional operations accessible</td>
</tr>
<tr>
<td>transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>may not be implemented</td>
<td>must be implemented</td>
</tr>
</tbody>
</table>
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.
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.
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.
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**

<table>
<thead>
<tr>
<th>Access</th>
<th><strong>SEES</strong></th>
<th><strong>USES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>Variables</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>Operations</td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td></td>
<td>only functional accessible</td>
<td>only functional accessible</td>
</tr>
<tr>
<td></td>
<td>not transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>
## Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th>Access</th>
<th>SEES</th>
<th>USES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>readonly must not be instantiated</td>
<td>readonly must not be instantiated</td>
</tr>
<tr>
<td>Variables</td>
<td>not referenced in invariant only functional accessible</td>
<td>may be referenced in invariant only functional accessible</td>
</tr>
<tr>
<td>Operations</td>
<td>not transitive must be implemented</td>
<td>not transitive cannot be implemented</td>
</tr>
</tbody>
</table>
### Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th></th>
<th>SEES</th>
<th>USES</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>parameters</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td></td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td></td>
<td>only functional accessible</td>
<td>only functional accessible</td>
</tr>
<tr>
<td></td>
<td>not transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td>variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>

- **SEES**:
  - Parameters and variables must not be instantiated and not referenced in the invariant.
  - Only functional accessible and not transitive.
  - Variables must be implemented.

- **USES**:
  - Parameters and variables must not be instantiated.
  - May be referenced in the invariant.
  - Only functional accessible and not transitive.
  - Cannot be implemented.
Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th>access parameters</th>
<th><strong>SEES</strong></th>
<th><strong>USES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>access parameters</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>parameters</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>variables</td>
<td>not referenced in invariant only functional accessible not transitive</td>
<td>may be referenced in invariant only functional accessible not transitive</td>
</tr>
<tr>
<td>operations</td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>
## Comparison between \textit{SEES} and \textit{USES}

<table>
<thead>
<tr>
<th>Access</th>
<th>\textit{SEES}</th>
<th>\textit{USES}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td></td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>Variables</td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td>Operations</td>
<td>only functional accessible</td>
<td>only functional accessible</td>
</tr>
<tr>
<td></td>
<td>not transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>
### Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th></th>
<th>SEES</th>
<th>USES</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>parameters</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>variables</td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td>operations</td>
<td>only functional accessible</td>
<td>only functional accessible</td>
</tr>
<tr>
<td></td>
<td>not transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>

**SEES** and **USES** are compared in terms of access, parameters, variables, and operations. SEES is more restrictive in terms of access and parameters, while USES allows more flexibility in terms of variables and operations.
## Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th></th>
<th><strong>SEES</strong></th>
<th><strong>USES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>parameters</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>variables</td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td>operations</td>
<td>only functional accessible, not transitive</td>
<td>only functional accessible, not transitive</td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>
Comparison between **SEES** and **USES**

<table>
<thead>
<tr>
<th></th>
<th><strong>SEES</strong></th>
<th><strong>USES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>parameters</td>
<td>must not be instantiated</td>
<td>must not be instantiated</td>
</tr>
<tr>
<td>variables</td>
<td>not referenced in invariant</td>
<td>may be referenced in invariant</td>
</tr>
<tr>
<td>operations</td>
<td>only functional accessible</td>
<td>only functional accessible</td>
</tr>
<tr>
<td></td>
<td>not transitive</td>
<td>not transitive</td>
</tr>
<tr>
<td></td>
<td>must be implemented</td>
<td>cannot be implemented</td>
</tr>
</tbody>
</table>
Inconsistent use of operations I

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 \( xx := xx + 1 \parallel xx := xx - 1 \), so 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, and is therefore forbidden.
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 $xx := xx + 1 || xx := xx - 1$, so 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, and is therefore forbidden.
Inconsistent use of operations I

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 \( xx := xx + 1 \| xx := xx - 1 \), so 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, and is therefore forbidden.
Inconsistent use of operations I

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 $xx := xx + 1 \parallel xx := xx - 1$, so 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, and is therefore forbidden.
Inconsistent use of operations I

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 $xx := xx + 1 \parallel xx := xx - 1$, so 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, and is therefore forbidden.
Inconsistent use of operations I

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 $xx := xx + 1 || xx := xx - 1$, so 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, and is therefore forbidden.
Inconsistent use of operations I

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 \( xx := xx + 1 \parallel xx := xx - 1 \), so 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, and is therefore forbidden.
Inconsistent use of operations II

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

1. 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.

2. 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.
Inconsistent use of operations II

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

1. 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.

2. 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.
Inconsistent use of operations II

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

1. 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.

2. 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.
Inconsistent use of operations II

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

1. 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.

2. 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.
For the above reasons the following restrictions apply to use of operations:

1. 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.

2. 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.
Inconsistent use of operations II

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

1. 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.

2. 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.
For the above reasons the following restrictions apply to use of operations:

1. 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.

2. 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.
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.
Global Sets, Constants and Variables

In the simple library case study we needed to have access to a global set \textit{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

\begin{verbatim}
MACHINE A(BOOK)  MACHINE B(BOOK)
\end{verbatim}

do not guarantee that both \textit{A} and \textit{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.
Global Sets, Constants and Variables

In the simple library case study we needed to have access to a global set \textit{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

\texttt{MACHINE A(BOOK) \quad MACHINE B(BOOK)}

do not guarantee that both \texttt{A} and \texttt{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.
Global Sets, Constants and Variables

In the simple library case study we needed to have access to a global set \textit{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

\begin{verbatim}
MACHINE A(BOOK) \hspace{1cm} MACHINE B(BOOK)
\end{verbatim}

do not guarantee that both \textit{A} and \textit{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.
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 \textit{SEES}. 
The correct way to represent a universal set is to embed it in a machine, for example

\[
\text{MACHINE } \text{Book\_TYPE ( maxbook )}
\]

\[
\text{CONSTRAINTS } \text{maxbook} \in \mathbb{N}_1
\]

\[
\text{SETS } \text{BOOK}
\]

\[
\text{PROPERTIES } \text{card ( BOOK )} = \text{maxbook}
\]

and to see that machine using \textit{SEES}. 
Example: A Date Machine

MACHINE Date (maxDate)
CONSTRAINTS maxDate ∈ N
VARIABLES today
INVARIANT today ∈ DATE
INITIALISATION today ∈ DATE

OPERATIONS
    newday ≜
        PRE today ≠ maxDate
        THEN today := today + 1
        END

DEFINITIONS DATE ≜ 0 .. maxDate
END
MACHINE Time
SEES Bool_TYPE
SETS TIME
CONSTANTS
    FUTURE,
    PAST,
    BigCrunch,
    BigBang
A Time Machine II

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 $FUTURE$.

$$\forall (t_1, t_2). (t_1 \in TIME \wedge t_2 \in TIME \wedge t_1 \neq t_2 \Rightarrow t_1 \leftrightarrow t_2 \in FUTURE \lor t_2 \leftrightarrow t_1 \in FUTURE) \wedge$$
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.

$$FUTURE \cap id(TIME) = \emptyset \land$$

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

$$FUTURE \cap PAST = \emptyset \land$$

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 $t_1$.

$$\forall (t_1, t_2, t_3) . (t_1 \in TIME \land t_2 \in TIME \land t_3 \in TIME \land (t_2 \in FUTURE [\{t_1\}] \land t_3 \in FUTURE [\{t_2\}]) \Rightarrow t_3 \in FUTURE [\{t_1\}]) \land$$
The following cosmology is concerned with time in a finite universe.

There are two extreme times, \textit{BigBang} and \textit{BigCrunch}.

\[ \text{BigBang} \in \text{TIME} \land \text{BigCrunch} \in \text{TIME} \land \]

they are not equal

\[ \text{BigBang} \neq \text{BigCrunch} \land \]

no time is in the future of \textit{BigCrunch},

\[ \text{BigCrunch} \notin \text{dom ( FUTURE )} \land \]

and no time is in the past of \textit{BigBang}.

\[ \text{BigBang} \notin \text{dom ( PAST )} \land \]
All times, except $\text{BigBang}$, are in the future of $\text{BigBang}$,

\[ FUTURE \left[ \{ \text{BigBang} \} \right] = TIME - \{ \text{BigBang} \} \wedge \]

and all times, except $\text{BigCrunch}$, are in the past of $\text{BigCrunch}$.

\[ PAST \left[ \{ \text{BigCrunch} \} \right] = TIME - \{ \text{BigCrunch} \} \]
A Time Machine VI

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

\textbf{VARIABLES} \textit{currenttime}

\textbf{INVARIANT} \hfill \textit{currenttime} \in \textit{TIME}

\textbf{INITIALISATION} \textit{currenttime} \in \textit{TIME}

\textbf{OPERATIONS}
A Time Machine VII

Operation Clock gives a time that is in the future of the current value of \( \text{currenttime} \).

\[
\text{time} \leftarrow \text{Clock} \equiv \\
\text{PRE} \quad \text{currenttime} \neq \text{BigCrunch} \\
\text{THEN ANY} \quad \text{now} \\
\text{WHERE} \quad \text{now} \in \text{FUTURE} [ \{ \text{currenttime} \} ] \\
\text{THEN} \quad \text{time} := \text{now} \parallel \text{currenttime} := \text{now} \\
\text{END} \\
\text{END} ;
\]
Operation Tick returns the next time tick after the current time. This is modelled as the “least” time in the future of $currenttime$.

\[
\begin{align*}
time & \leftarrow \text{Tick} \\
& \equiv \\
& \text{PRE } currenttime \neq \text{BigCrunch} \\
& \text{THEN ANY } now \\
& \quad \text{WHERE } now \in FUTURE [ \{ currenttime \} ] \land \\
& \quad FUTURE [ \{ currenttime \} ] \cap \text{PAST} [ \{ now \} ] = \{ \} \\
& \quad \text{THEN } time \coloneqq now \ || \ currenttime \coloneqq now \\
& \end{align*}
\]
A Time Machine IX

Operation Later(t1, t2) returns TRUE if \( t1 \in FUTURE[t2] \) and FALSE otherwise.

\[
\text{later} \leftarrow \text{Later} (t1, t2) \equiv \\
\begin{align*}
\text{PRE} & \quad t1 \in TIME \land t2 \in TIME \\
\text{THEN} & \quad \text{later} := \text{bool} (t1 \in FUTURE[t2]) \\
\text{END} & ;
\end{align*}
\]
Operation Earlier($t_1, t_2$) returns TRUE if $t_1 \in PAST[\{t_2\}]$ and FALSE otherwise.

\[
earlier \leftarrow \textbf{Earlier} ( t_1 , t_2 ) \triangleq
\begin{align*}
\text{PRE} & \quad t_1 \in TIME \land t_2 \in TIME \\
\text{THEN} & \quad \text{earlier} := \text{bool} ( t_1 \in PAST [ \{ t_2 \} ] )
\end{align*}
\]

END

END