System Modelling and Design

Event B Semantics

Revision: 1.0, March 3, 2007

Ken Robinson

School of Computer Science & Engineering
The University of New South Wales, Sydney Australia

September 26, 2008

© Ken Robinson 2005

mailto:k.robinson@unsw.edu.au
Outline I

1. What is this lecture about?

2. Semantics in Event B
   - State Change
   - Substitution
   - Context Machines
   - Machines
   - Events
   - Machine Refinements
   - Refined Events
This lecture presents the semantics of Event-B.

The various Proof Obligations (POs) that result from those semantics.

An understanding of “what those POs mean”.

The roles of POs in verifying a refinement.

The classification of POs, which identify what a particular PO is “all about”.

What is this lecture about?
This lecture presents the semantics of Event-B.

1. The various *Proof Obligations* (PO) that result from those semantics.
2. An understanding of “what those POs mean”.
3. The roles of POs in verifying a refinement.
4. The classification of POs, which identify what a particular PO is “all about”.

Outline

1. What is this lecture about?
2. Semantics in Event B
This lecture presents the semantics of Event-B.

The various *Proof Obligations* (PO) that result from those semantics.

An understanding of “what those POs mean”.

The roles of POs in verifying a refinement.

The classification of POs, which identify what a particular PO is “all about”.

What is this lecture about?
What is this lecture about?

1. This lecture presents the semantics of Event-B.
2. The various *Proof Obligations* (PO) that result from those semantics.
3. An understanding of “what those POs mean”.
4. The roles of POs in verifying a refinement.
5. The classification of POs, which identify what a particular PO is “all about”.

Outline

- What is this lecture about?
- Semantics in Event B
This lecture presents the semantics of Event-B.

The various *Proof Obligations* (PO) that result from those semantics.

An understanding of “what those POs mean”.

The roles of POs in verifying a refinement.

The classification of POs, which identify what a particular PO is “all about”.

What is this lecture about?

1. This lecture presents the semantics of Event-B.
2. The various *Proof Obligations* (PO) that result from those semantics.
3. An understanding of “what those POs mean”.
4. The roles of POs in verifying a refinement.
5. The classification of POs, which identify what a particular PO is “all about”.
Each construct in B is given a formal semantics.

Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
Semantics in Event B

- Each construct in B is given a formal semantics.
- Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
Semantics in Event B

Each construct in B is given a formal semantics.

Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
Semantics in Event B

- Each construct in B is given a formal semantics.
- Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
Semantics in Event B

Each construct in B is given a formal semantics.
Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
Each construct in B is given a formal semantics. Additionally, machines must satisfy a set of constraints. These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is consistent with the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is correct, that is, conforms with the written requirements.
State Change

There are three principle constructions — that Event B calls *substitutions* — for changing the state of a machine:

- \( x := e \)  \( x \) becomes equal to the value of \( e \)
  This rule may be used recursively to assign to any number of variables.

- \( x :| P \)  \( x \) becomes such that it satisfies the *before-after* predicate \( P \)

- \( x :\in s \)  \( x \) becomes in the set \( s \)

All of the above, except apparently \(:\in\), can be extended to *multiple assignment*: \( x, y := e_1, e_2 \) and \( x, y :| P \), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \( x, y :| P \).
State Change

There are three principle constructions—that Event B calls *substitutions*—for changing the state of a machine:

\[ x := e \quad x \text{ becomes equal to the value of } e \]

This rule may be used recursively to assign to any number of variables.

\[ x :| P \quad x \text{ becomes such that it satisfies the before-after predicate } P \]

\[ x \in s \quad x \text{ becomes in the set } s \]

All of the above, except apparently \( \in \), can be extended to *multiple assignment*: \( x, y := e_1, e_2 \) and \( x, y :| P \), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \( x, y :| P \).
State Change

There are three principle constructions—that Event B calls *substitutions*—for changing the state of a machine:

\[
\begin{align*}
    x & := e \quad \text{*}x \text{ becomes equal to the value of } e* \\
    x & :| P \quad \text{*}x \text{ becomes such that it satisfies the before-after predicate } P* \\
    x & :\in s \quad \text{*}x \text{ becomes in the set } s*
\end{align*}
\]

All of the above, except apparently :\in, can be extended to *multiple assignment*: \(x, y := e_1, e_2\) and \(x, y :| P\), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \(x, y :| P\).
State Change

There are three principle constructions — that Event B calls *substitutions*— for changing the state of a machine:

- \( x := e \) *x becomes equal to the value of e*
  
  This rule may be used recursively to assign to any number of variables.

- \( x :| P \) *x becomes such that it satisfies the before-after predicate P*

- \( x :\in s \) *x becomes in the set s*

All of the above, except apparently \( :\in \), can be extended to *multiple assignment*: \( x, y := e_1, e_2 \) and \( x, y :| P \), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \( x, y :| P \).
There are three principle constructions—that Event B calls *substitutions*—for changing the state of a machine:

\[ x := e \]  
* \( x \) becomes *equal to* the value of \( e \)

This rule may be used recursively to assign to any number of variables.

\[ x :| P \]  
* \( x \) becomes *such that* it satisfies the *before-after* predicate \( P \)

\[ x :\in s \]  
* \( x \) becomes *in* the set \( s \)

All of the above, except apparently \( :\in \), can be extended to *multiple assignment*: \( x, y := e_1, e_2 \) and \( x, y :| P \), and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: \( x, y :| P \).
Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the *after* value of a variable and the unprimed variables the *before* value.

Thus,

\[ x :| x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y :| x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

$$x' = x + 1$$

where the primed variables represent the after value of a variable and the unprimed variables the before value.

Thus,

$$x :| x' = x + 1$$

and

$$x := x + 1$$

are equivalent.

Similarly we can write

$$x, y :| x' = x + 1 \land y' = y + 1$$

or

$$x, y := x + 1, y + 1.$$
Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the \textit{after} value of a variable and the unprimed variables the \textit{before} value.

Thus,

\[ x : | x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y : | x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the *after* value of a variable and the unprimed variables the *before* value.

Thus,

\[ x :| x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y :| x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the *after* value of a variable and the unprimed variables the *before* value.

Thus,

\[ x :| x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y :| x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the \textit{after} value of a variable and the unprimed variables the \textit{before} value.

Thus,

\[ x :| x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y :| x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

\[ x' = x + 1 \]

where the primed variables represent the \textit{after} value of a variable and the unprimed variables the \textit{before} value.

Thus,

\[ x :| \ x' = x + 1 \]

and

\[ x := x + 1 \]

are equivalent.

Similarly we can write

\[ x, y :| \ x' = x + 1 \land y' = y + 1 \]

or

\[ x, y := x + 1, y + 1. \]
We will frequently need to compute, for example in computing POs, the weakest predicate on the state *before* a state given a required predicate on the *after* state.

We can do this by *substituting* into the after state.

We will write

\[ [x, y := e_1, e_2] R \]

to denote the *concurrent* substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[
[x, y := y - 1, x + 1] x - y < x + y \\
= (y - 1) - (x + 1) < (y - 1) + (x + 1) \\
or y - x - 2 < y + x
\]

This gives the weakest constraint on the *before* state such that \( x, y := y - 1, x + 1 \) will give an *after* state satisfying \( x - y < x + y \).
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state \textit{before} a state given a required predicate on the \textit{after} state.

\textbf{We can do this by \textit{substituting} into the after state.}

We will write

\[ [x, y := e_1, e_2]R \]

\[ R \]

to denote the \textit{concurrent} substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[ [x, y := y - 1, x + 1]x - y < x + y \]
\[ = (y - 1) - (x + 1) < (y - 1) + (x + 1) \]
\[ \text{or } y - x - 2 < y + x \]

This gives the weakest constraint on the \textit{before} state such that \( x, y := y - 1, x + 1 \) will give an \textit{after} state satisfying \( x - y < x + y \).
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by substituting into the after state.

We will write

\[ [x, y := e_1, e_2] R \]

to denote the concurrent substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[ [x, y := y - 1, x + 1]x - y < x + y \]

\[ = (y - 1) - (x + 1) < (y - 1) + (x + 1) \]

or

\[ y - x - 2 < y + x \]

This gives the weakest constraint on the before state such that \( x, y := y - 1, x + 1 \) will give an after state satisfying \( x - y < x + y \).
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by substituting into the after state.

We will write

\[ [x, y := e_1, e_2]R \]

to denote the concurrent substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[
[x, y := y - 1, x + 1]x - y < x + y \\
= (y - 1) - (x + 1) < (y - 1) + (x + 1) \\
\text{or} \quad y - x - 2 < y + x
\]

This gives the weakest constraint on the before state such that \( x, y := y - 1, x + 1 \) will give an after state satisfying \( x - y < x + y \).
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by substituting into the after state.

We will write

\[
[x, y := e_1, e_2]R
\]

to denote the concurrent substitution of \(e_1\) and \(e_2\) for \(x\) and \(y\) in \(R\), respectively.

For example,

\[
[x, y := y - 1, x + 1]x - y < x + y
= (y - 1) - (x + 1) < (y - 1) + (x + 1)
= y - x - 2 < y + x
\]

This gives the weakest constraint on the before state such that \(x, y := y - 1, x + 1\) will give an after state satisfying \(x - y < x + y\).
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by substituting into the after state.

We will write

$$[x, y := e_1, e_2]R$$

to denote the concurrent substitution of $e_1$ and $e_2$ for $x$ and $y$ in $R$, respectively.

For example,

$$[x, y := y - 1, x + 1]x - y < x + y$$

$$= (y - 1) - (x + 1) < (y - 1) + (x + 1)$$

$$or \quad y - x - 2 < y + x$$

This gives the weakest constraint on the before state such that $x, y := y - 1, x + 1$ will give an after state satisfying $x - y < x + y$. 
Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state before a state given a required predicate on the after state.

We can do this by *substituting* into the after state.

We will write

\[ [x, y := e_1, e_2] R \]

to denote the *concurrent* substitution of \( e_1 \) and \( e_2 \) for \( x \) and \( y \) in \( R \), respectively.

For example,

\[
[x, y := y - 1, x + 1]x - y < x + y \\
= (y - 1) - (x + 1) < (y - 1) + (x + 1) \\
or \quad y - x - 2 < y + x
\]

This gives the weakest constraint on the before state such that \( x, y := y - 1, x + 1 \) will give an after state satisfying \( x - y < x + y \).
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \quad [\text{Substitution}]R \\
v := E & \quad [v := E]R \\
v :| P & \quad \forall v' \cdot P \implies [v := v']R \\
v :\in S & \quad \forall v' \cdot v' \in S \implies [v := v']R
\end{align*}
\]

where:

- \(v\) in general is a list of variables, and \(E\) a list of expressions;
- \(P\) is a predicate containing both \(v\) and \(v'\), where \(v'\) represents the value of \(v\) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \\
\nu := E & \rightarrow [\nu := E]R \\
\nu :\mid P & \rightarrow \forall \nu' \cdot P \Rightarrow [\nu := \nu']R \\
\nu :\in S & \rightarrow \forall \nu' \cdot \nu' \in S \Rightarrow [\nu := \nu']R
\end{align*}
\]

where:

- \(\nu\) in general is a list of variables, and \(E\) a list of expressions;
- \(P\) is a predicate containing both \(\nu\) and \(\nu'\), where \(\nu'\) represents the value of \(\nu\) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \quad [\text{Substitution}] R \\
\forall v \cdot P & \quad \Rightarrow [v := v'] R \\
\forall v' \cdot v' \in S & \quad \Rightarrow [v := v'] R
\end{align*}
\]

where:

- \( v \) in general is a list of variables, and \( E \) a list of expressions;
- \( P \) is a predicate containing both \( v \) and \( v' \), where \( v' \) represents the value of \( v \) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \quad [\text{Substitution}] \ R \\
\nu := E & \quad [\nu := E] \ R \\
\nu \mid P & \quad \forall \nu' \cdot P \implies [\nu := \nu'] \ R \\
\nu \in S & \quad \forall \nu' \cdot \nu' \in S \implies [\nu := \nu'] \ R
\end{align*}
\]

where:

- \( \nu \) in general is a list of variables, and \( E \) a list of expressions;
- \( P \) is a predicate containing both \( \nu \) and \( \nu' \), where \( \nu' \) represents the value of \( \nu \) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \quad \text{[Substitution]} \ R \\
\nu & := E \\
\nu : | \ P & \quad \forall \nu' \cdot P \implies [\nu := \nu'] R \\
\nu : \in \ S & \quad \forall \nu' \cdot \nu' \in S \implies [\nu := \nu'] R
\end{align*}
\]

where:

- \( \nu \) in general is a list of variables, and \( E \) a list of expressions;
- \( P \) is a predicate containing both \( \nu \) and \( \nu' \), where \( \nu' \) represents the value of \( \nu \) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\text{Substitution} \quad [\text{Substitution}]R \\
\begin{align*}
\nu &:= E \\
\nu :| P &\quad \forall v' \cdot P \implies [\nu := v']R \\
\nu :\in S &\quad \forall v' \cdot v' \in S \implies [\nu := v']R
\end{align*}
\]

where:

1. \(\nu\) in general is a list of variables, and \(E\) a list of expressions;
2. \(P\) is a predicate containing both \(\nu\) and \(v'\), where \(v'\) represents the value of \(\nu\) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

\[
\begin{align*}
\text{Substitution} & \quad [\text{Substitution}]R \\
\mathbf{v} := E & \quad [\mathbf{v} := E]R \\
\mathbf{v} : | P & \quad \forall \mathbf{v}' \cdot P \implies [\mathbf{v} := \mathbf{v}']R \\
\mathbf{v} : \in S & \quad \forall \mathbf{v}' \cdot \mathbf{v}' \in S \implies [\mathbf{v} := \mathbf{v}']R
\end{align*}
\]

where:

1. \( \mathbf{v} \) in general is a list of variables, and \( E \) a list of expressions;
2. \( P \) is a predicate containing both \( \mathbf{v} \) and \( \mathbf{v}' \), where \( \mathbf{v}' \); represents the value of \( \mathbf{v} \) after the action.
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

<table>
<thead>
<tr>
<th>Substitution</th>
<th>[Substitution]R</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v := E )</td>
<td>([v := E]R)</td>
</tr>
<tr>
<td>( v ::= P )</td>
<td>( \forall v' \cdot P \implies [v := v']R )</td>
</tr>
<tr>
<td>( v \in S )</td>
<td>( \forall v' \cdot v' \in S \implies [v := v']R )</td>
</tr>
</tbody>
</table>

where:

1. \( v \) in general is a list of variables, and \( E \) a list of expressions;
2. \( P \) is a predicate containing both \( v \) and \( v' \), where \( v' \); represents the value of \( v \) after the action.

**one point rule**
Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

<table>
<thead>
<tr>
<th>Substitution</th>
<th>[Substitution]R</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v := E$</td>
<td>$[v := E]R$</td>
</tr>
<tr>
<td>$v :</td>
<td>P$</td>
</tr>
<tr>
<td>$v :\in S$</td>
<td>$\forall v' \cdot v' \in S \Rightarrow [v := v']R$</td>
</tr>
</tbody>
</table>

where:

1. $v$ in general is a list of variables, and $E$ a list of expressions;
2. $P$ is a predicate containing both $v$ and $v'$, where $v'$; represents the value of $v$ after the action.

one point rule
Context Machines

Context machines are used to define abstract carrier sets ($S$) and constants ($C$).

The form of a context machine is:

Sets

Constants

Axioms

Theorems

$S$

$C$

$A$

$T_c$

Notice that $S$ and $C$ should be augmented with “builtin” sets and constants such as $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ etc and constants from those sets, but we will elide any explicit extension.
Context machines are used to define abstract carrier sets ($S$) and constants ($C$).

The form of a context machine is:

- **Sets**: $S$
- **Constants**: $C$
- **Axioms**: $A$
- **Theorems**: $T_c$

Notice that $S$ and $C$ should be augmented with “builtin” sets and constants such as $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ etc and constants from those sets, but we will elide any explicit extension.
Context machines are used to define abstract carrier sets \((S)\) and constants \((C)\).

The form of a context machine is:

\[
\begin{array}{c|c|c|c|c}
\text{Sets} & S \\
\hline
\text{Constants} & C \\
\hline
\text{Axioms} & A \\
\hline
\text{Theorems} & T_c
\end{array}
\]

Notice that \(S\) and \(C\) should be augmented with “builtin” sets and constants such as \(N, N_1, Z\) etc and constants from those sets, but we will elide any explicit extension.
The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of *feasibility*: show that sets and constants exist that will satisfy the axioms. That is:

\[(\exists S, C \cdot A)\]

The axioms are usually broken up into a sequence of sub-axioms \(a_1, a_2, \ldots a_n\), which are effectively conjuncted into a single \(A\). The POs can be recursively split into separate POs based on

\[(\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n) \equiv (\exists S, C \cdot a_1) \land ((\exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \land \ldots \land a_n))\]

This may require the sub-axioms to be ordered.

Of course, components of \(S, C\) that are not referenced in \(a_i\) can be eliminated from \(\exists S, C \cdot a_i\).
The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of \textit{feasibility}: show that sets and constants exist that will satisfy the axioms. That is:

\[(\exists S, C \cdot A)\]

The axioms are usually broken up into a sequence of sub-axioms \(a_1, a_2, \ldots a_n\), which are effectively conjuncted into a single \(A\). The POs can be recursively split into separate POs based on

\[(\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n) \equiv (\exists S, C \cdot a_1) \land ((\exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \land \ldots \land a_n))\]

This may require the sub-axioms to be ordered.

Of course, components of \(S, C\) that are not referenced in \(a_i\) can be eliminated from \(\exists S, C \cdot a_i\).
The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of feasibility: show that sets and constants exist that will satisfy the axioms. That is:

\[(\exists S, C \cdot A)\]

The axioms are usually broken up into a sequence of sub-axioms \(a_1, a_2, \ldots a_n\), which are effectively conjuncted into a single \(A\). The POs can be recursively split into separate POs based on

\[(\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n) \equiv (\exists S, C \cdot a_1) \land ((\exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \land \ldots \land a_n))\]

This may require the sub-axioms to be ordered.

Of course, components of \(S, C\) that are not referenced in \(a_i\) can be eliminated from \(\exists S, C \cdot a_i\).
The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of *feasibility*: show that sets and constants exist that will satisfy the axioms. That is:

\[(\exists S, C \cdot A)\]

The axioms are usually broken up into a sequence of sub-axioms \(a_1, a_2, \ldots a_n\), which are effectively conjuncted into a single \(A\). The POs can be recursively split into separate POs based on

\[(\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n) \equiv (\exists S, C \cdot a_1) \land ((\exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \land \ldots \land a_n))\]

This may require the sub-axioms to be ordered.

Of course, components of \(S, C\) that are not referenced in \(a_i\) can be eliminated from \(\exists S, C \cdot a_i\).
The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of feasibility: show that sets and constants exist that will satisfy the axioms. That is:

\[(\exists S, C \cdot A)\]

The axioms are usually broken up into a sequence of sub-axioms \(a_1, a_2, \ldots a_n\), which are effectively conjuncted into a single \(A\). The POs can be recursively split into separate POs based on

\[(\exists S, C \cdot a_1 \land a_2 \land \ldots \land a_n)\]
\[\equiv (\exists S, C \cdot a_1) \land ((\exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \land \ldots \land a_n))\]

This may require the sub-axioms to be ordered

Of course, components of \(S, C\) that are not referenced in \(a_i\) can be eliminated from \(\exists S, C \cdot a_i\).
Theorems describe properties that follow from the axioms, so the general PO for the theorems is

\[(\forall S, C.A \implies T_c)\]

The theorems will, in general, be broken in sub-theorems \(t_1, t_2, \ldots t_n\), and since universal quantification distributes through conjunction this breaks into multiple POs:

\[(\forall S, C.A \implies t_1), \ldots (\forall S, C.A \implies t_n)\]
Theorems describe properties that follow from the axioms, so the general PO for the theorems is

$$(\forall S, C.A \implies T_c)$$

The theorems will, in general, be broken in sub-theorems $t_1, t_2, \ldots t_n$, and since universal quantification distributes through conjunction this breaks into multiple POs:

$$(\forall S, C.A \implies t_1), \ldots (\forall S, C.A \implies t_n)$$
Theorems describe properties that follow from the axioms, so the general PO for the theorems is

\[
(\forall S, C.A \implies T_c)
\]

The theorems will, in general, be broken in sub-theorems \( t_1, t_2, \ldots t_n \), and since universal quantification distributes through conjunction this breaks into multiple POs:

\[
(\forall S, C.A \implies t_1), \ldots (\forall S, C.A \implies t_n)
\]
The form of a machine is:

<table>
<thead>
<tr>
<th>Context</th>
<th>$S,C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>$V$</td>
</tr>
<tr>
<td>Invariant</td>
<td>$I$</td>
</tr>
<tr>
<td>Theorems</td>
<td>$T_v$</td>
</tr>
<tr>
<td>Variant</td>
<td>$Var$</td>
</tr>
<tr>
<td>Events</td>
<td>$E$</td>
</tr>
</tbody>
</table>
The invariant  as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[(\exists S, C \cdot A) \implies (\exists V \cdot I)\]

The theorems must follow from the set/constant axioms and the invariant:

\[\forall S, C, V \cdot A \land I \implies T_v\]

Note: where we have \(A\) we could also have \(A \land T_c\), but since \(A \implies C\) this does not gain any extra strengthening.
The invariant as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[(\exists S, C \cdot A) \implies (\exists V \cdot I)\]

The theorems must follow from the set/constant axioms and the invariant:

\[\forall S, C, V \cdot A \land I \implies T_v\]

Note: where we have \(A\) we could also have \(A \land T_c\), but since \(A \implies C\) this does not gain any extra strengthening.
The invariant as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[(\exists S, C \cdot A) \implies (\exists V \cdot I)\]

The theorems must follow from the set/constant axioms and the invariant:

\[\forall S, C, V \cdot A \wedge I \implies T_v\]

Note: where we have \(A\) we could also have \(A \wedge T_c\), but since \(A \implies C\) this does not gain any extra strengthening.
The invariant as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[(\exists S, C \cdot A) \implies (\exists V \cdot I)\]

The theorems must follow from the set/constant axioms and the invariant:

\[\forall S, C, V \cdot A \land I \implies T_v\]

Note: where we have \(A\) we could also have \(A \land T_c\), but since \(A \implies C\) this does not gain any extra strengthening.
The invariant  as for the axioms for context machines, the invariant may raise feasibility proof obligations:

\[(\exists S, C \cdot A) \implies (\exists V \cdot I)\]

The theorems  must follow from the set/constant axioms and the invariant:

\[\forall S, C, V \cdot A \land I \implies T_v\]

Note: where we have \(A\) we could also have \(A \land T_c\), but since \(A \implies C\) this does not gain any extra strengthening.
Initialisation, which is a special part of the events, must establish a state in which the variables satisfy the invariant.

Let us represent the initialisation by a multiple substitution

\[ V := E(S, C) \]

where \( E(S, C) \) emphasises that the initialising expressions can only reference sets and constants: \( E \) must not reference any variables, since all variables at this point are undefined.

Then the proof obligation for initialisation is

\[ \forall S, C \cdot A \Rightarrow [V := E(S, C)] I \]
Initialisation, which is a special part of the events, must establish a state in which the variables satisfy the invariant.

Let us represent the initialisation by a multiple substitution

\[ V := E(S, C) \]

where \( E(S, C) \) emphasises that the initialising expressions can only reference sets and constants: \( E \) must not reference any variables, since all variables at this point are undefined.

Then the proof obligation for initialisation is

\[ \forall S, C \cdot A \rightarrow [V := E(S, C)] I \]
Initialisation, which is a special part of the events, must establish a state in which the variables satisfy the invariant.

Let us represent the initialisation by a multiple substitution

\[ V := E(S, C) \]

where \( E(S, C) \) emphasises that the initialising expressions can only reference sets and constants: \( E \) must not reference any variables, since all variables at this point are undefined.

Then the proof obligation for initialisation is

\[ \forall S, C \cdot A \implies [V := E(S, C)] I \]
Events have the following form

\[
\text{ANY } x \quad \text{WHEN } G \quad \text{THEN } Action
\]
There may be feasibility POs: that there exist parameters $P$ that will satisfy the guards $G$

$$\forall S, C \cdot A \land \exists V, x \cdot I \land G$$
The event must maintain the invariant of the machine: essentially the invariant will be true before the event is scheduled and must remain true when the event terminates.

\[ \forall S, C, V, x \cdot A \land I \land G \implies [Action] I \]
The form of a refinement machine is

<table>
<thead>
<tr>
<th>Context</th>
<th>$S_r, C_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>$V_r$</td>
</tr>
<tr>
<td>Invariant</td>
<td>$I_r$</td>
</tr>
<tr>
<td>Theorems</td>
<td>$T^+_v$</td>
</tr>
<tr>
<td>Events</td>
<td>$E_r$ refines $E$</td>
</tr>
<tr>
<td>Variant</td>
<td>$Var$</td>
</tr>
</tbody>
</table>

where $E_r$ represents a refined event and $E$ represents new normal events.
The variable \( V_r \) are in general a superset of the variables in the machine being refined.

The invariant is the invariant of the refined machine plus invariants for the new variables. In addition the invariant contains the refinement relation relating the state of the refined machine to the variables of the refining machine. This gives a *simulation* relation.

The proof obligations for the variables, invariant and theorems are similar to those for the machine given above. We will concentrate on the new proof obligations that arise from the refined events.
The variable $V_r$ are in general a superset of the variables in the machine being refined.

The invariant is the invariant of the refined machine plus invariants for the new variables. In addition the invariant contains the refinement relation relating the state of the refined machine to the variables of the refining machine. This gives a *simulation* relation.

The proof obligations for the variables, invariant and theorems are similar to those for the machine given above. We will concentrate on the new proof obligations that arise from the refined events.
The variable $V_r$ are in general a superset of the variables in the machine being refined.

The invariant is the invariant of the refined machine plus invariants for the new variables. In addition the invariant contains the refinement relation relating the state of the refined machine to the variables of the refining machine. This gives a \textit{simulation} relation.

The proof obligations for the variables, invariant and theorems are similar to those for the machine given above. We will concentrate on the new proof obligations that arise from the refined events.
\( \forall V_i, V \cdot I_r \implies I \)

the new invariant must not allow behaviour that was not part of the refined machine’s behaviour, excepting where the state of the refining machine is “orthogonal” to the refined machine.
Refined Events have the following form

\[
\text{ANY} \quad x_r \\
\text{WHEN} \quad G_r \\
\text{WITH} \quad w : W \\
\text{THEN} \quad \text{Action}_r
\]
Proof Obligations for Refined Events

**guard refinement**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \implies (G_r \implies G) \]

**witness**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \implies (\exists w \cdot W) \]

**Simulation**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \land W \land [Action_r]I_r \implies [Action]I \]

where \( A_r \) denotes the refinement axioms.
Proof Obligations for Refined Events

**guard refinement**
\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \implies (G_r \implies G)
\]

**witness**
\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \implies (\exists w \cdot W)
\]

**Simulation**
\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \land W \land [Action_r]I_r \implies [Action]I
\]

where \(A_r\) denotes the refinement axioms.
Proof Obligations for Refined Events

**guard refinement**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \Rightarrow (G_r \Rightarrow G) \]

**witness**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \Rightarrow (\exists w \cdot W) \]

**Simulation**
\[ \forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land A_r \land I \land I_r \land W \land [Action_r]I_r \Rightarrow [Action]I \]

where \( A_r \) denotes the refinement axioms.
The Variant and Convergent Events

The variant \((\text{Var})\) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{Var} \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies [\text{Action}_r] \text{Var} < \text{Var}
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(\text{Var})
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}([\text{Action}_r] \text{Var}) < \text{card}(\text{Var})
\]
The Variant and Convergent Events

The variant \((Var)\) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies Var \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{[Action}_r\text{]}Var < Var
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(Var)
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card([Action}_r\text{]}Var) < \text{card}(Var)
\]
The variant (Var) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies Var \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies [Action_r] Var < Var
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(Var)
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}([Action_r] Var) < \text{card}(Var)
\]
The variant \((\text{Var})\) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{Var} \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies [\text{Action}_r] \text{Var} < \text{Var}
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(\text{Var})
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}(\text{[Action}_r]\text{Var}) < \text{card}(\text{Var})
\]
The variant \((\text{Var})\) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{Var} \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \lbrack \text{Action}_r \rbrack \text{Var} < \text{Var}
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(\text{Var})
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}(\lbrack \text{Action}_r \rbrack \text{Var}) < \text{card}(\text{Var})
\]
The variant \((Var)\) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies Var \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies [Action_r]Var < Var
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(Var)
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}([Action_r]Var) < \text{card}(Var)
\]
The Variant and Convergent Events

The variant (\(Var\)) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

**Natural number variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies Var \in \mathbb{N}
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies [\text{Action}_r] Var < Var
\]

**Set variant**

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{finite}(Var)
\]

\[
\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \land I \land I_r \land W \implies \text{card}([\text{Action}_r] Var) < \text{card}(Var)
\]
Consider $\forall x \cdot x \in X \land x = e \implies P(x)$.

For any $x$ in $S$, $x = e$ is either true or false. If it is false then the universal quantification is trivially true; if it is true then the quantification reduces to $P(e)$. So

$$(\forall x \cdot x \in X \land x = e \implies P(x)) = P(e)$$

By a similar argument,

$$(\exists x \cdot x \in X \land x = e \land P(x)) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \land x = e$. 

By a similar argument,
Consider $\forall x \cdot x \in X \land x = e \implies P(x)$.

For any $x$ in $S$, $x = e$ is either *true* or *false*. If it is *false* then the universal quantification is trivially *true*; if it is *true* then the quantification reduces to $P(e)$. So

$$(\forall x \cdot x \in X \land x = e \implies P(x)) = P(e)$$

By a similar argument,

$$(\exists x \cdot x \in X \land x = e \land P(x)) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \land x = e$. 

Return
Consider $\forall x \cdot x \in X \land x = e \implies P(x)$.

For any $x$ in $S$, $x = e$ is either true or false. If it is false then the universal quantification is trivially true; if it is true then the quantification reduces to $P(e)$. So

\[(\forall x \cdot x \in X \land x = e \implies P(x)) = P(e)\]

By a similar argument,

\[(\exists x \cdot x \in X \land x = e \land P(x)) = P(e)\]

Strictly, each should be conjuncted with $\exists x \cdot x \in X \land x = e$. 
Consider $\forall x \cdot x \in X \land x = e \implies P(x)$.

For any $x$ in $S$, $x = e$ is either true or false. If it is false then the universal quantification is trivially true; if it is true then the quantification reduces to $P(e)$. So

$$\left( \forall x \cdot x \in X \land x = e \implies P(x) \right) = P(e)$$

By a similar argument,

$$\left( \exists x \cdot x \in X \land x = e \land P(x) \right) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \land x = e$. 
Consider $\forall x \cdot x \in X \land x = e \implies P(x)$.

For any $x$ in $S$, $x = e$ is either true or false. If it is false then the universal quantification is trivially true; if it is true then the quantification reduces to $P(e)$. So

$$(\forall x \cdot x \in X \land x = e \implies P(x)) = P(e)$$

By a similar argument,

$$(\exists x \cdot x \in X \land x = e \land P(x)) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \land x = e$. 