System Modelling and Design

A Whirlwind Introduction to the B Method

Revision: 1.2, March 5, 2003

Ken Robinson

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

19th August 2005

©Ken Robinson 2005
mailto::k.robinson@unsw.edu.au
Outline I

1. A Homily: Formality is inexorable
2. Some Terminology
3. Notation
   - Abstract Machines
   - Machine Variables in B
   - Object based
   - Substitutions
   - Abstract Machine Notation
4. The B-Toolkit
   - The B-Toolkit interface
   - Introducing a new machine
5. A Simple Model
Outline II

- Machine Structure
- … Machine Structure
- Machine Parameters
- Operations
- Invariant and Preconditions
- Trivial preconditions
- Problem with the PiggyBank Machine
- Proof obligation generation and proof
- Viewing the proof obligations
- Adding a non-trivial precondition
- Towards understanding preconditions
III

- Total and Partial operations: preconditions

6  Modelling a Coffee Club
   - A CoffeeClub machine
   - Some notes on machine inclusion
   - Problems with CoffeeClub
   - Identifying and fixing the problems

7  Specifying a Robust machine
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples. Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics.

There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples. Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics. There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics. There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics.

There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics. There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics.

There is no reason to believe that this “progress” will not continue.
The increase of the use formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics.

There is no reason to believe that this “progress” will not continue.
The following terms will be used frequently:

**predicate**  
a predicate is a partial function from variables (state) to Boolean. The predicate is usually expressed as a closed expression, e.g. $\text{amount} < \text{balance}(\text{customer})$.

**satisfies**  
we talk of some variables satisfying a predicate. This means that substituting the values of the variables into the predicate will make the predicate true.

**stronger and weaker**  
if $P \Rightarrow Q$ we frequently say that, “$P$ is stronger than $Q$”, although strictly we should say, “$P$ is at least as strong as $Q$”. Similarly, we might say “$Q$ is weaker than $P$”.

In the same vein we will talk of strengthening or weakening a predicate. Strengthening a predicate subsets the set of values that satisfy the predicate. Weakening a predicate supersedes the set of values that satisfy the predicate.
Some Terminology

The following terms will be used frequently:

**predicate** a predicate is a partial function from variables (state) to Boolean. The predicate is usually expressed as a closed expression, e.g. $amount < balance(customer)$.

**satisfies** we talk of some variables satisfying a predicate. This means that substituting the values of the variables into the predicate will make the predicate *true*.

**stronger and weaker** if $P \Rightarrow Q$ we frequently say that, “$P$ is *stronger than* $Q$”, although strictly we should say, “$P$ is *at least as strong as* $Q$”. Similarly, we might say “$Q$ is *weaker than* $P$”.

In the same vein we will talk of *strengthening* or *weakening* a predicate. Strengthening a predicate subsets the set of values that satisfy the predicate. Weakening a predicate superset the set of values that satisfy the predicate.
Some Terminology

The following terms will be used frequently:

**predicate** a predicate is a partial function from variables (state) to Boolean. The predicate is usually expressed as a closed expression, e.g. \( amount < balance(customer) \).

**satisfies** we talk of some variables satisfying a predicate. This means that substituting the values of the variables into the predicate will make the predicate true.

**stronger and weaker** if \( P \Rightarrow Q \) we frequently say that, “\( P \) is stronger than \( Q \)”, although strictly we should say, “\( P \) is at least as strong as \( Q \)”. Similarly, we might say “\( Q \) is weaker than \( P \)”.

In the same vein we will talk of strengthening or weakening a predicate. Strengthening a predicate subsets the set of values that satisfy the predicate. Weakening a predicate supersetsthe set of values that satisfy the predicate.
Some Terminology

The following terms will be used frequently:

**Predicate** a predicate is a partial function from variables (state) to Boolean. The predicate is usually expressed as a closed expression, e.g. \( amount < balance(customer) \).

**Satisfies** we talk of some variables satisfying a predicate. This means that substituting the values of the variables into the predicate will make the predicate true.

**Stronger and weaker** if \( P \Rightarrow Q \) we frequently say that, “\( P \) is stronger than \( Q \)”, although strictly we should say, “\( P \) is at least as strong as \( Q \)”. Similarly, we might say “\( Q \) is weaker than \( P \)”.

In the same vein we will talk of **strengthening** or **weakening** a predicate. Strengthening a predicate subsets the set of values that satisfy the predicate. Weakening a predicate supersets the set of values that satisfy the predicate.
All components of a *B Method* (*B*) development will have a source form, used to specify machines and other input to the B-Toolkit, and a publication form used in documentation.

The notation for the source form will be **ASCII**. For example,

```
account : ACCOUNT
```

means the variable *account* is an element of the set *ACCOUNT*.

The notation for publication will is marked up high quality mathematics. For example,

```
account ∈ ACCOUNT,
```

which has the same meaning as the **ASCII** example.
All components of a B development will have a source form, used to specify machines and other input to the B-Toolkit, and a publication form used in documentation.

The notation for the source form will be ASCII. For example,

\[\text{account : ACCOUNT}\]

means the variable \text{account} is an element of the set \text{ACCOUNT}.

The notation for publication will is marked up high quality mathematics. For example,

\[\text{account} \in \text{ACCOUNT},\]

which has the same meaning as the ASCII example.
Notation

All components of a B development will have a source form, used to specify machines and other input to the B-Toolkit, and a publication form used in documentation.

The notation for the source form will be ASCII. For example,

\[ \text{account} : \text{ACCOUNT} \]

means the variable \text{account} is an element of the set \text{ACCOUNT}.

The notation for publication will is marked up high quality mathematics. For example,

\[ \text{account} \in \text{ACCOUNT}, \]

which has the same meaning as the ASCII example.
All components of a B development will have a source form, used to specify machines and other input to the B-Toolkit, and a publication form used in documentation.

The notation for the source form will be ASCII. For example,

\[
\text{account : ACCOUNT}
\]

means the variable \texttt{account} is an element of the set \texttt{ACCOUNT}.

The notation for publication will is marked up high quality mathematics. For example,

\[
\text{account} \in \text{ACCOUNT},
\]

which has the same meaning as the ASCII example.
B uses *Abstract Machines*, which are machines that encapsulate:

- A state consisting of a set of variables constrained by an invariant.
- Operations may change the state, while maintaining the invariant, and may return a sequence of results.
B uses Abstract Machines, which are machines that encapsulate:

- **state**: consisting of a set of variables constrained by an *invariant*
- **operations**: operations may change the state, *while maintaining the invariant*, and may return a sequence of results.
B uses *Abstract Machines*, which are machines that encapsulate:

- **state** consisting of a set of variables constrained by an *invariant*

- **operations** operations may change the state, *while maintaining the invariant*, and may return a sequence of results.
For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus $xx$ is a valid variable, while $x$ is not.

**Warning:** this is likely to cause many mysterious problems in your first attempts to write B machines. The error messages of the B-Toolkit will not clearly identify the problem!

Where single letters are used in describing the notation, those letters represent context dependent expressions, which include proper variables.
Machine Variables in B

For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus \( xx \) is a valid variable, while \( x \) is not.

Warning: this is likely to cause many mysterious problems in your first attempts to write B machines. The error messages of the B-Toolkit will not clearly identify the problem!

Where single letters are used in describing the notation, those letters represent context dependent expressions, which include proper variables.
Machine Variables in B

For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus \( xx \) is a valid variable, while \( x \) is not.

**Warning**: this is likely to cause many mysterious problems in your first attempts to write B machines. The error messages of the B-Toolkit will not clearly identify the problem!

Where single letters are used in describing the notation, those letters represent context dependent expressions, which include proper variables.
For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus \( xx \) is a valid variable, while \( x \) is not.

**Warning**: this is likely to cause many mysterious problems in your first attempts to write B machines. The error messages of the B-Toolkit will not clearly identify the problem!

Where single letters are used in describing the notation, those letters represent context dependent expressions, which include proper variables.
Abstract machines are sometimes described as object-based, rather than object-oriented.

You will notice that a machine can be compared with an object, that is, an instance of a class.

Importantly, a machine does not behave as a class, although it is possible to model a class.
Abstract machines are sometimes described as *object-based*, rather than *object-oriented*.

You will notice that a machine can be compared with an *object*, that is, an instance of a *class*.

Importantly, a machine does not behave as a class, although it is possible to model a class.
Abstract machines are sometimes described as *object-based*, rather than *object-oriented*.

You will notice that a machine can be compared with an *object*, that is, an instance of a *class*.

Importantly, a machine does not behave as a class, although it is possible to model a class.
Abstract machines are sometimes described as *object-based*, rather than *object-oriented*.

You will notice that a machine can be compared with an *object*, that is, an instance of a *class*.

Importantly, a machine does not behave as a class, although it is possible to model a class.
Substitutions

The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL. Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
Substitutions

The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL.

Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL.

Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
Substitutions

The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL. Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL.

Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

We won’t describe the GSL at this stage, but we will note that there are only 11 basic substitutions in the GSL. Substitutions are given a formal semantics that in turn is expressed in terms of substitution of values; thus the word “substitution” is a pun.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines.

AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.

We will use only a few AMN constructs here.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines.

AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.

We will use only a few AMN constructs here.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines.

AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.

We will use only a few AMN constructs here.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines.

AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.

We will use only a few AMN constructs here.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines.

AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.

We will use only a few AMN constructs here.
The B-Toolkit is a configuration management tool that provides the following facilities:

- Introduction of new machines
- Animation of specifications
- Automatic & interactive proof
- Markup of machines
- Generation of code
- Execution of generated code
- Automatic remakes
- Hypertext displays of machines
- Syntax and type analysis
- Generation of proof obligations
- Introduction of user theories
- Maintenance of documents
- Generation of interfaces
- Generation of base machines
- Browsing of designs & specifications
- Online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- Introduction of new machines
- Syntax and type analysis
- Animation of specifications
- Generation of proof obligations
- Automatic & interactive proof
- Introduction of user theories
- Markup of machines
- Maintenance of documents
- Generation of code
- Generation of interfaces
- Execution of generated code
- Generation of base machines
- Automatic remakes
- Browsing of designs & specifications
- Hypertext displays of machines
- Online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines

- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- Introduction of new machines
- Animation of specifications
- Automatic & interactive proof
- Markup of machines
- Generation of code
- Execution of generated code
- Automatic remakes
- Hypertext displays of machines
- Syntax and type analysis
- Generation of proof obligations
- Introduction of user theories
- Maintenance of documents
- Generation of interfaces
- Generation of base machines
- Browsing of designs & specifications
- Online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit

The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- Introduction of new machines
- Animation of specifications
- Automatic & interactive proof
- Markup of machines
- Generation of code
- Execution of generated code
- Automatic remakes
- HyperText displays of machines
- Syntax and type analysis
- Generation of proof obligations
- Introduction of user theories
- Maintenance of documents
- Generation of interfaces
- Generation of base machines
- Browsing of designs & specifications
- Online help
The B-Toolkit

The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit

The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines

- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The B-Toolkit is a configuration management tool that provides the following facilities:

- introduction of new machines
- animation of specifications
- automatic & interactive proof
- markup of machines
- generation of code
- execution of generated code
- automatic remakes
- hypertext displays of machines
- syntax and type analysis
- generation of proof obligations
- introduction of user theories
- maintenance of documents
- generation of interfaces
- generation of base machines
- browsing of designs & specifications
- online help
The interface of the B-Toolkit is very compact, but has a large number of configurations.

Menu bar: the top line contains menus that control the functions of the toolkit.

Environments: Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

Machine panel: below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

Log panel: at the bottom is another panel that contains a log of the interactions for the current session.
The B-Toolkit interface

The interface of the B-Toolkit is very compact, but has a large number of configurations.

**Menu bar** the top line contains menus that control the functions of the toolkit.

**Environments** Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

**Machine panel** below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

**Log panel** at the bottom is another panel that contains a log of the interactions for the current session.
The B-Toolkit interface

The interface of the B-Toolkit is very compact, but has a large number of configurations.

Menu bar  the top line contains menus that control the functions of the toolkit.

Environments  Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

Machine panel  below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

Log panel  at the bottom is another panel that contains a log of the interactions for the current session.
The interface of the B-Toolkit is very compact, but has a large number of configurations.

**Menu bar**  the top line contains menus that control the functions of the toolkit.

**Environments**  Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

**Machine panel**  below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

**Log panel**  at the bottom is another panel that contains a log of the interactions for the current session.
The B-Toolkit interface

The interface of the B-Toolkit is very compact, but has a large number of configurations.

**Menu bar** the top line contains menus that control the functions of the toolkit.

**Environments** Below the menu bar is a set of environments: **Main**, **Provers**, etc that present different views on the development process.

**Machine panel** below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

**Log panel** at the bottom is another panel that contains a log of the interactions for the current session.
The interface of the B-Toolkit is very compact, but has a large number of configurations.

Menu bar the top line contains menus that control the functions of the toolkit.

Environments Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

Machine panel below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

Log panel at the bottom is another panel that contains a log of the interactions for the current session.
The interface of the B-Toolkit is very compact, but has a large number of configurations.

**Menu bar** the top line contains menus that control the functions of the toolkit.

**Environments** Below the menu bar is a set of environments: **Main**, **Provers**, etc that present different views on the development process.

**Machine panel** below the **Environments** is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

**Log panel** at the bottom is another panel that contains a log of the interactions for the current session.
To introduce a new machine you would select \texttt{Introduce/New/Machine} in the \texttt{Main} environment of the B-Toolkit. Having introduced the machine, a template will appear in your editor. The machine should be “filled in” and saved. Then the machine should be committed and analyzed, by selecting the \texttt{cmt} (commit) and \texttt{anl} (analyze) buttons in the \texttt{Main} environment.
To introduce a new machine you would select **Introduce/New/Machine** in the **Main** environment of the B-Toolkit.

Having introduced the machine, a template will appear in your editor. The machine should be “filled in” and saved.

Then the machine should be committed and analyzed, by selecting the **cmt** (commit) and **anl** (analyze) buttons in the **Main** environment.
To introduce a new machine you would select `Introduce/New/Machine` in the `Main` environment of the B-Toolkit.

Having introduced the machine, a template will appear in your editor. The machine should be “filled in” and saved.

Then the machine should be committed and analyzed, by selecting the `cmt` (commit) and `anl` (analyze) buttons in the `Main` environment.
To introduce a new machine you would select **Introduce/New/Machine** in the **Main** environment of the B-Toolkit.

Having introduced the machine, a template will appear in your editor. The machine should be “filled in” and saved.

Then the machine should be committed and analyzed, by selecting the **cmt** (commit) and **anl** (analyze) buttons in the **Main** environment.
A Simple Model I

As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

- Feedbank\((amount)\)
  feed \(amount\) cents to the piggybank.

- RobBank\((amount)\)
  Rob the piggybank of \(amount\) cents.

- \(money \leftarrow \text{CashLeft}\)
  Query the piggybank to obtain the amount of \(money\) left in the piggybank.
A Simple Model I

As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

- **Feedbank**(amount)
  
  feed *amount* cents to the piggybank.

- **RobBank**(amount)
  
  Rob the piggybank of *amount* cents.

- \( money \leftarrow \text{CashLeft} \)
  
  Query the piggybank to obtain the amount of *money* left in the piggybank.
A Simple Model I

As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

\[ \text{Feedbank}(\text{amount}) \]

feed \textit{amount} cents to the piggybank.

\[ \text{RobBank}(\text{amount}) \]

Rob the piggybank of \textit{amount} cents.

\[ \text{money} \leftarrow \text{CashLeft} \]

Query the piggybank to obtain the amount of \textit{money} left in the piggybank.
A Simple Model I

As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

- **Feedbank**(\textit{amount})
  
  feed \textit{amount} cents to the piggybank.

- **RobBank**(\textit{amount})
  
  Rob the piggybank of \textit{amount} cents.

- \textit{money} ← **CashLeft**
  
  Query the piggybank to obtain the amount of \textit{money} left in the piggybank.
As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

- **Feedback\((amount)\)***
  feed \(amount\) cents to the piggybank.

- **RobBank\((amount)\)***
  Rob the piggybank of \(amount\) cents.

- **\(money \leftarrow \text{CashLeft}\)***
  Query the piggybank to obtain the amount of \(money\) left in the piggybank.
As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out using the following operations:

\[
\text{Feedbank}(amount) \quad \text{feed } amount \text{ cents to the piggybank.}
\]

\[
\text{RobBank}(amount) \quad \text{Rob the piggybank of } amount \text{ cents.}
\]

\[
\text{money ← CashLeft} \quad \text{Query the piggybank to obtain the amount of } money \text{ left in the piggybank.}
\]
In order to model the operations we will use a variable `piggybank` whose value is a natural number, representing the contents of the piggybank in cents.

Let's step through the specification of a machine that “owns” and manages the piggy bank.
In order to model the operations we will use a variable *piggybank* whose value is a natural number, representing the contents of the piggybank in cents.

Let’s step through the specification of a machine that “owns” and manages the piggy bank.
In order to model the operations we will use a variable *piggybank* whose value is a natural number, representing the contents of the piggybank in cents.

Let’s step through the specification of a machine that “owns” and manages the piggy bank.
MACHINE PiggyBank0
VARIABLES piggybank
INVARIANT piggybank ∈ ℕ
INITIALISATION piggybank := 0

OPERATIONS
FeedBank \( (\ amount\ ) \) \( \triangleq \)
\[
\text{PRE} \quad amount \in \mathbb{N} \text{ THEN} \\
\quad piggybank := piggybank + amount \\
\text{END} \; ;
\]
RobBank \( (\ amount\ ) \) \( \triangleq \)
\[
\text{PRE} \quad amount \in \mathbb{N} \text{ THEN} \\
\quad piggybank := piggybank - amount \\
\text{END} \; ;
\]
\[
money \leftarrow \text{CashLeft} \triangleq \\
\text{BEGIN} \\
\quad money := piggybank \\
\text{END} \\
\text{END}
\]
In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>name</th>
<th>set and numeric parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>INARIANT</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>name</th>
<th>set and numeric parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>INVARIANT</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>Clause Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>predicate</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
## Machine Structure

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name</td>
<td>set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>INARIANT</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
Machine Structure

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>name</th>
<th>set and numeric parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine</td>
<td>parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>INARIANT</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
<table>
<thead>
<tr>
<th>clauses</th>
<th>name</th>
<th>set and numeric parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MACHINE</strong></td>
<td>name</td>
<td>set and numeric parameters</td>
</tr>
<tr>
<td><strong>CONSTRAINTS</strong></td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td><strong>INCLUDES/SEES/USES</strong></td>
<td>machine</td>
<td>parameters</td>
</tr>
<tr>
<td><strong>SETS</strong></td>
<td>names</td>
<td></td>
</tr>
<tr>
<td><strong>CONSTANTS</strong></td>
<td>names</td>
<td></td>
</tr>
<tr>
<td><strong>PROPERTIES</strong></td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td><strong>VARIABLES</strong></td>
<td>names</td>
<td></td>
</tr>
<tr>
<td><strong>IN VaRIANT</strong></td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td><strong>INITIALISATION</strong></td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td><strong>OPERATIONS</strong></td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td><strong>END</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>predicate</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>predicate</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
Machine Structure

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
</tr>
<tr>
<td>INARIANT</td>
<td>predicate</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
### Machine Structure

<table>
<thead>
<tr>
<th>Clause</th>
<th>Parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>name, set and numeric parameters</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine, parameters</td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
</tr>
<tr>
<td>INARIANT</td>
<td>predicate</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
</tr>
</tbody>
</table>

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- constraints constrains the machine *parameters*
- properties constrains the sets and constants
- invariant constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Machine Structure

Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- constraints constrains the machine *parameters*
- properties constrains the *sets* and *constants*
- invariant constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- **constraints** constrains the machine **parameters**
- properties constrains the **sets** and **constants**
- invariant constrains the **variables**

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- **constraints** constrains the machine *parameters*
- **properties** constrains the *sets* and *constants*
- **invariant** constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- **constraints** constrains the machine *parameters*
- **properties** constrains the *sets* and *constants*
- **invariant** constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- **constraints** constrains the machine *parameters*
- **properties** constrains the *sets* and *constants*
- **invariant** constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

- **constraints** constrains the machine *parameters*
- **properties** constrains the *sets* and *constants*
- **invariant** constrains the *variables*

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Machine parameters enable the specification of generic machines. The parameters are either:

- sets: upper case identifiers; denote finite non-empty sets
- numeric: natural number constants
Machine parameters enable the specification of generic machines. The parameters are either:

- sets: upper case identifiers; denote finite non-empty sets
- numeric: natural number constants
Machine parameters enable the specification of generic machines. The parameters are either:

- **sets** upper case identifiers; denote finite non-empty sets
- **numeric** natural number constants
Machine parameters enable the specification of generic machines. The parameters are either:

- **sets** upper case identifiers; denote finite non-empty sets
- **numeric** natural number constants
Machine parameters enable the specification of generic machines. The parameters are either:

- **sets** upper case identifiers; denote finite non-empty sets
- **numeric** natural number constants
Machine parameters enable the specification of generic machines. The parameters are either:

- **sets**  upper case identifiers; denote finite non-empty sets
- **numeric**  natural number constants
The form of an operation is:

\[
\text{operation-signature} \triangleq \text{substitution}
\]

An \textit{operation-signature} has the form:

- \( \text{name}(\text{args}) \) for an operation that only makes a state substitution, or

- \( \text{results} \leftarrow \text{name}(\text{args}) \), where \text{results} is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
The form of an operation is:

\[
\text{operation-signature} \equiv \text{substitution}
\]

An \text{operation-signature} has the form:

- \text{name(args)} for an operation that only makes a state substitution, or
- \text{results } \leftarrow \text{name(args)}, where \text{results} is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
The form of an operation is:

\[
\text{operation-signature} \triangleq \text{substitution}
\]

An \textit{operation-signature} has the form:

- \textit{name(args)} for an operation that only makes a state substitution, or
- \textit{results \leftarrow name(args)} , where \textit{results} is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
The form of an operation is:

\[
\text{operation-signature} \equiv \text{substitution}
\]

An operation-signature has the form:

- \( \text{name(args)} \)
  - for an operation that only makes a state substitution, or
- \( \text{results} \leftarrow \text{name(args)} \)
  - where results is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
The form of an operation is:

\[
\text{operation-signature} \equiv \text{substitution}
\]

An operation-signature has the form:

- \( \text{name(args)} \) for an operation that only makes a state substitution, or
- \( \text{results} \leftarrow \text{name(args)} \), where \( \text{results} \) is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
The form of an operation is:

\[ \text{operation-signature} \equiv \text{substitution} \]

An \textit{operation-signature} has the form:

- \[ \text{name}(\text{args}) \] for an operation that only makes a state substitution, or
- \[ \text{results} \leftarrow \text{name}(\text{args}) \], where \text{results} is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called safety or integrity conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called safety or integrity conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called *safety* or *integrity* conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation *maintains* the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called safety or integrity conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called safety or integrity conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called safety or integrity conditions.

The initial state must satisfy the invariant, and it is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.
Although the specification of FeedBank and RobBank use a preconditioned substitution the precondition is used only to carry the type of the parameter to the operation.

This is a trivial precondition.
Although the specification of **FeedBank** and **RobBank** use a preconditioned substitution the precondition is used only to carry the type of the parameter to the operation.

This is a *trivial* precondition.
Although the specification of **FeedBank** and **RobBank** use a preconditioned substitution the precondition is used only to carry the type of the parameter to the operation.

This is a *trivial* precondition.
There is a problem with the PiggyBank machine.

See if you can spot it.

Alternatively, generate the proof obligations and try to discharge them.
Problem with the PiggyBank Machine

There is a problem with the PiggyBank machine.

See if you can spot it.

Alternatively, generate the proof obligations and try to discharge them.
There is a problem with the PiggyBank machine.

See if you can spot it.

Alternatively, generate the proof obligations and try to discharge them.
Problem with the PiggyBank Machine

There is a problem with the PiggyBank machine.

See if you can spot it.

Alternatively, generate the proof obligations and try to discharge them.
Proof obligation generation and proof

Having analyzed a machine, you should routinely generate the proof obligations by selecting the `pog` (proof obligation generator) button in the Main environment.

Then move to the Provers environment, select the `prv` (provers) button for the machine, and select AutoProver. If there are unproved obligations then you should either try to discharge the proof obligation using the BToolProver, or at least inspect the obligation to see if it is true.

This should be a routine validation step.
Proof obligation generation and proof

Having analyzed a machine, you should routinely generate the proof obligations by selecting the **pog** (proof obligation generator) button in the **Main** environment.

Then move to the **Provers** environment, select the **prv** (provers) button for the machine, and select **AutoProver**. If there are unproved obligations then you should either try to discharge the proof obligation using the **BToolProver**, or at least inspect the obligation to see if it is true.

This should be a routine validation step.
Proof obligation generation and proof

Having analyzed a machine, you should routinely generate the proof obligations by selecting the pog (proof obligation generator) button in the Main environment.

Then move to the Provers environment, select the prv (provers) button for the machine, and select AutoProver. If there are unproved obligations then you should either try to discharge the proof obligation using the BToolProver, or at least inspect the obligation to see if it is true.

This should be a routine validation step.
Proof obligation generation and proof

Having analyzed a machine, you should routinely generate the proof obligations by selecting the pog (proof obligation generator) button in the Main environment.

Then move to the Provers environment, select the prv (provers) button for the machine, and select AutoProver. If there are unproved obligations then you should either try to discharge the proof obligation using the BToolProver, or at least inspect the obligation to see if it is true.

This should be a routine validation step.
Viewing the proof obligations

Select the Provers environment and select the `ppf` (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the Documents environment, and notice that there is a green `.prf` construct for the chosen machine.

Mark-up the proof obligations by selecting the `dmu` (document markup) button; the view by selecting the `shw` (show) button.
Select the **Provers** environment and select the **ppf** (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the **Documents** environment, and notice that there is a green .prf construct for the chosen machine.

Mark-up the proof obligations by selecting the **dmu** (document markup) button; the view by selecting the **shw** (show) button.
Select the **Provers** environment and select the **ppf** (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the **Documents** environment, and notice that there is a green .prf construct for the chosen machine.

Mark-up the proof obligations by selecting the **dmu** (document markup) button; the view by selecting the **shw** (show) button.
Select the Provers environment and select the ppf (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the Documents environment, and notice that there is a green .prf construct for the chosen machine.

Mark-up the proof obligations by selecting the dmu (document markup) button; the view by selecting the shw (show) button.
Viewing the proof obligations

Select the **Provers** environment and select the `ppf` (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the **Documents** environment, and notice that there is a green `.prf` construct for the chosen machine.

Mark-up the proof obligations by selecting the `dmu` (document markup) button; the view by selecting the `shw` (show) button.
An attempt to discharge the outstanding proof obligation for the operation `RobBank` will leave `amount \leq piggybank` unprovable. This occurs because the machine invariant says that `piggybank \in \mathbb{N}`, that is `0 \leq piggybank` both before and after an operation.

Thus we need to add the conjunct `amount \leq piggybank` to the precondition of `RobBank`. 
An attempt to discharge the outstanding proof obligation for the operation RobBank will leave $\textit{amount} \leq \textit{piggybank}$ unprovable.

This occurs because the machine invariant says that $\textit{piggybank} \in \mathbb{N}$, that is $0 \leq \textit{piggybank}$ both before and after an operation.

Thus we need to add the conjunct $\textit{amount} \leq \textit{piggybank}$ to the precondition of RobBank.
An attempt to discharge the outstanding proof obligation for the operation RobBank will leave \( amount \leq piggybank \) unprovable. This occurs because the machine invariant says that \( piggybank \in \mathbb{N} \), that is \( 0 \leq piggybank \) both before and after an operation.

Thus we need to add the conjunct \( amount \leq piggybank \) to the precondition of RobBank.
An attempt to discharge the outstanding proof obligation for the operation RobBank will leave $amount \leq piggybank$ unprovable. This occurs because the machine invariant says that $piggybank \in \mathbb{N}$, that is $0 \leq piggybank$ both before and after an operation.

Thus we need to add the conjunct $amount \leq piggybank$ to the precondition of RobBank.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:
1. enable display invariant — the default is not display;
2. run:
   - FeedBank(5)
   - RobBank(10)
   - FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant—the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   - FeedBank(5)
   - RobBank(10)
   - FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   - FeedBank(5)
   - RobBank(10)
   - FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   - FeedBank(5)
   - RobBank(10)
   - FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   - FeedBank(5)
   - RobBank(10)
   - FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Towards understanding preconditions

Run the following experiment:

1. run the animator on PiggyBank with RobBank having a trivial precondition;
2. run the animator on PiggyBank with RobBank having the non-trivial precondition.

In each case:

1. enable display invariant — the default is not display;
2. run:
   1. FeedBank(5)
   2. RobBank(10)
   3. FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition does not stop the operation from going ahead.
Total and Partial operations: preconditions

Operations without non-trivial preconditions are *total* operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called *robust*.

Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition. Such operations are also called *fragile*.

A precondition is an *assumption*, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Total and Partial operations: preconditions

Operations without non-trivial preconditions are *total* operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called *robust*.

Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition. Such operations are also called *fragile*.

A precondition is an *assumption*, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Operations without non-trivial preconditions are *total* operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called *robust*.

Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition. Such operations are also called *fragile*.

A precondition is an *assumption*, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Total and Partial operations: preconditions

Operations without non-trivial preconditions are *total* operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called *robust*.

Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition. Such operations are also called *fragile*.

A precondition is an *assumption*, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Total and Partial operations: preconditions

Operations without non-trivial preconditions are *total* operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called *robust*.

Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition. Such operations are also called *fragile*.

A precondition is an *assumption*, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Modelling a Coffee Club

We will now model a coffee club with the following facilities for members:

Joining  a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set \( \text{NAME} \). Of course we want all members to be distinct.

Contributing members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

Buy coffee a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

Credit a member can obtain their current credit balance.

The above behaviour is modelled by the machine \( \text{CoffeeClub} \), initially named \( \text{CoffeeClub0} \).
Modelling a Coffee Club

We will now model a coffee club with the following facilities for members:

- **Joining** a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set $\text{name}$. Of course we want all members to be distinct.

- **Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

- **Buy coffee** a member can buy a cup of coffee. The price of a cup of coffee is deducted from the member's credit.

- **Credit** a member can obtain their current credit balance.

The above behaviour is modelled by the machine `CoffeeClub`, initially named `CoffeeClub0`.
Modelling a Coffee Club

We will now model a coffee club with the following facilities for members:

- **Joining** a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set \( \text{NAME} \). Of course we want all members to be distinct.

- **Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

- **Buy coffee** a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

- **Credit** a member can obtain their current credit balance.

The above behaviour is modelled by the machine \( \text{CoffeeClub} \), initially named \( \text{CoffeeClub0} \).
We will now model a coffee club with the following facilities for members:

**Joining** a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set \( \text{NAME} \). Of course we want all members to be distinct.

**Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

**Buy coffee** a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

**Credit** a member can obtain their current credit balance.

The above behaviour is modelled by the machine CoffeeClub, initially named CoffeeClub0.
We will now model a coffee club with the following facilities for members:

**Joining** a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set $\text{NAME}$. Of course we want all members to be distinct.

**Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

**Buy coffee** a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

**Credit** a member can obtain their current credit balance.

The above behaviour is modelled by the machine CoffeeClub, initially named CoffeeClub0.
We will now model a coffee club with the following facilities for members:

**Joining** a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set \( \text{NAME} \). Of course we want all members to be distinct.

**Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

**Buy coffee** a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

**Credit** a member can obtain their current credit balance.

The above behaviour is modelled by the machine CoffeeClub, initially named CoffeeClub0.
We will now model a coffee club with the following facilities for members:

**Joining**  a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set $\text{NAME}$. Of course we want all members to be distinct.

**Contributing** members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

**Buy coffee**  a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

**Credit**  a member can obtain their current credit balance.

The above behaviour is modelled by the machine $\text{CoffeeClub}$, initially named $\text{CoffeeClub0}$.
MACHINE CoffeeClub0 ( NAME )
INCLUDES PiggyBank
PROMOTES RobBank, CashLeft
CONSTANTS coffee
PROPERTIES coffee = 120
VARIABLES finances
INVARIANT finances ∈ NAME → N
INITIALISATION finances := {}

OPERATIONS
A CoffeeClub machine II

NewMember ( member ) \equiv
PRE member \in \text{NAME}
THEN

\text{finances} ( \text{member} ) := 0

END ;

Contribute ( member , amount ) \equiv
PRE member \in \text{NAME} \land amount \in \mathbb{N}
THEN

\text{finances} ( \text{member} ) := \text{finances} ( \text{member} ) + amount \parallel
\text{FeedBank} ( \text{amount} )

END ;
A CoffeeClub machine III

BuyCoffee ( member ) ≡
  PRE member ∈ NAME
  THEN
    finances ( member ) := finances ( member ) − coffee
  END ;
credit ← Credit ( member ) ≡
  PRE member ∈ NAME
  THEN credit := finances ( member )
  END
END
Aspects of CoffeeClub

Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to N.
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of CoffeeClub

Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of CoffeeClub

Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to N.
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of CoffeeClub

Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to N.
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The **PiggyBank** machine is *included* into this machine. This embeds the state of **PiggyBank** into this machine, and gives **CoffeeClub** access to the operations of **PiggyBank**.
- The operations RobBank and CashLeft are *promoted* to the interface of **CoffeeClub**.
- A constant *coffee* is used for the cost of a cup of coffee.
- The state of the machine consists of a variable *finances*, which is a partial function from **NAME** to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is *included* into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are *promoted* to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to \( \mathbb{N} \).
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
Some notes on machine inclusion

Included machine state: the included machine’s state is “added” to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, but only by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.
Some notes on machine inclusion

Included machine state: the included machine’s state is “added” to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, but only by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.
Some notes on machine inclusion

Included machine state: the included machine’s state is “added” to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, but only by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.
Some notes on machine inclusion

Included machine state: the included machine’s state is “added” to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, but only by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.
Some notes on machine inclusion

**Included machine state:** the included machine’s state is “added” to the state of the including machine.

**Referencing included state:** the variables in the state of the included machine may be referenced by the including machine.

**Modifying the variables of included state:** variables of the included machine may be modified by the included machine, but only by invoking operations of the included machine.

**Export of operations:** While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

**Included machine parameters:** if the included machine has parameters they must be instantiated by the including machine.
Some notes on machine inclusion

Included machine state: the included machine’s state is “added” to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, **but only** by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.
The specification given by this machine is not adequate. It is easy to show that the operations can break the invariant.

Generating the proof obligations and attempting to discharge them will illustrate some of the problems. Run the AutoProver on the proof obligations and examine any undischarged proof obligations.

Animation may help to illustrate where the problems lie.
Problems with CoffeeClub

The specification given by this machine is not adequate. It is easy to show that the operations can break the invariant.

Generating the proof obligations and attempting to discharge them will illustrate some of the problems. Run the AutoProver on the proof obligations and examine any undischarged proof obligations.

Animation may help to illustrate where the problems lie.
Problems with CoffeeClub

The specification given by this machine is not adequate. It is easy to show that the operations can break the invariant.

Generating the proof obligations and attempting to discharge them will illustrate some of the problems. Run the AutoProver on the proof obligations and examine any undischarged proof obligations.

Animation may help to illustrate where the problems lie.
Problems with CoffeeClub

The specification given by this machine is not adequate. It is easy to show that the operations can break the invariant.

Generating the proof obligations and attempting to discharge them will illustrate some of the problems. Run the AutoProver on the proof obligations and examine any undischarged proof obligations.

Animation may help to illustrate where the problems lie.
Identifying and fixing the problems I

The problems are enumerated below:

NewMember: this operation has an undesirable functional property: if an existing member—or a new member with the same name as an existing member—with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition $\text{member} \notin \text{dom}(\text{finances})$, that is, the prospective member is not an existing member.

Contribute: the function $\text{finances}$ is partial, so the expression used to update the member’s finances:

$$\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}$$

will be undefined when $\text{member} \notin \text{dom}(\text{finances})$. A precondition that $\text{member} \in \text{dom}(\text{finances})$ is required.
Identifying and fixing the problems I

The problems are enumerated below:

**NewMember** this operation has an undesirable functional property: if an existing member—or a new member with the same name as an existing member—with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition \( \text{member} \notin \text{dom(} \text{finances} \text{)} \), that is, the prospective member is not an existing member.

**Contribute** the function \( \text{finances} \) is partial, so the expression used to update the member’s finances:

\[
\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}
\]

will be undefined when \( \text{member} \notin \text{dom(} \text{finances} \text{)} \). A precondition that \( \text{member} \in \text{dom(} \text{finances} \text{)} \) is required.
The problems are enumerated below:

NewMember  this operation has an undesirable functional property: if an existing member—or a new member with the same name as an existing member—with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition \( \text{member} \not\in \text{dom}(\text{finances}) \), that is, the prospective member is not an existing member.

Contribute the function \( \text{finances} \) is partial, so the expression used to update the member’s finances:

\[
\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}
\]

will be undefined when \( \text{member} \not\in \text{dom}(\text{finances}) \). A precondition that \( \text{member} \in \text{dom}(\text{finances}) \) is required.
Identifying and fixing the problems I

The problems are enumerated below:

**NewMember** this operation has an undesirable functional property: if an existing member—or a new member with the same name as an existing member—with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition \( \text{member} \notin \text{dom}(\text{finances}) \), that is, the prospective member is not an existing member.

**Contribute** the function \( \text{finances} \) is partial, so the expression used to update the member’s finances:

\[
\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}
\]

will be undefined when \( \text{member} \notin \text{dom}(\text{finances}) \). A precondition that \( \text{member} \in \text{dom}(\text{finances}) \) is required.
Identifying and fixing the problems I

The problems are enumerated below:

**NewMember**  this operation has an undesirable functional property: if an existing member—or a new member with the same name as an existing member—with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition $\text{member } \not\in \text{dom}(\text{finances})$, that is, the prospective member is not an existing member.

**Contribute**  the function $\text{finances}$ is partial, so the expression used to update the member’s finances:

\[
\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}
\]

will be undefined when $\text{member } \not\in \text{dom}(\text{finances})$. A precondition that $\text{member } \in \text{dom}(\text{finances})$ is required.
BuyCoffee  In order to buy a coffee, two things are required

- the person must be a member, otherwise 
  \( \text{finances}(\text{member}) \) will be undefined;
- a member must have enough finance to cover the
  price of a cup of coffee. If this is not the case then
  \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural
  number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit  \( \text{finances}(\text{member}) \) assumes
\( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to
the precondition.

The following versions of **PiggyBank** and **CoffeeClub** have
appropriately strengthened preconditions.
Identifying and fixing the problems II

BuyCoffee  In order to buy a coffee, two things are required

1. the person must be a member, otherwise \( \text{finances}(\text{member}) \) will be undefined;
2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit  \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
BuyCoffee  In order to buy a coffee, two things are required

1. the person must be a member, otherwise \( \text{finances}(\text{member}) \) will be undefined;
2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \text{finances}(\text{member}) \geq \text{coffee}
\]

Credit \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
BuyCoffee In order to buy a coffee, two things are required

1. the person must be a member, otherwise finances(member) will be undefined;
2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then finances(member) − coffee will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit finances(member) assumes member \in \text{dom}(\text{finances}), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
Identifying and fixing the problems II

BuyCoffee  In order to buy a coffee, two things are required

1 the person must be a member, otherwise \( \text{finances}(\text{member}) \) will be undefined;

2 a member must have enough finance to cover the price of a cup of coffee. If this is not the case then \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit  \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
Identifying and fixing the problems II

BuyCoffee  In order to buy a coffee, two things are required

1. the person must be a member, otherwise \( \text{finances}(\text{member}) \) will be undefined;
2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then \( \text{finances}(\text{member}) - \text{ coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{ coffee}
\]

Credit  \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
Identifying and fixing the problems II

BuyCoffee  In order to buy a coffee, two things are required

1. the person must be a member, otherwise \( \text{finances}(\text{member}) \) will be undefined;

2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit  \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), so this needs to be added to the precondition.

The following versions of PiggyBank and CoffeeClub have appropriately strengthened preconditions.
MACHINE PiggyBank
VARIABLES piggybank
INVARIANT piggybank ∈ N
INITIALISATION piggybank := 0

OPERATIONS
FeedBank (amount) ≜
PRE amount ∈ N THEN
piggybank := piggybank + amount
END ;
RobBank (amount) ≜
PRE amount ∈ N ∧ amount ≤ piggybank THEN
piggybank := piggybank − amount
END ;
money ←− CashLeft ≜
BEGIN
money := piggybank
END
END
MACHINE CoffeeClub ( NAME )
INCLUDES PiggyBank
PROMOTES RobBank, CashLeft
CONSTANTS coffee
PROPERTIES coffee = 120
VARIABLES finances
INVARIANT finances ∈ NAME → N
INITIALISATION finances := {}
CoffeeClub.mch II

NewMember ( member ) ≡
PRE  member ∈ NAME ∧ member ∉ dom ( finances )
THEN
  finances ( member ) := 0
END ;
Contribute ( member , amount ) ≡
PRE  member ∈ NAME ∧
    member ∈ dom ( finances ) ∧ amount ∈ \mathbb{N}
THEN
  finances ( member ) := finances ( member ) + amount \parallel
  FeedBank ( amount )
END ;
CoffeeClub.mch III

\[\text{BuyCoffee} \ (\ member \ ) \triangleq \]
\[\begin{align*}
\text{PRE} & \quad member \in NAME \land member \in \text{dom}(\text{finances}) \land \\
& \quad \text{finances}(\ member \ ) \geq \ coffee \\
\text{THEN} & \quad \text{finances}(\ member \ ) := \text{finances}(\ member \ ) - \ coffee \\
\text{END} ; \quad \text{credit} \leftarrow \ \text{Credit} \ (\ member \ ) \triangleq \]
\[\begin{align*}
\text{PRE} & \quad member \in NAME \land member \in \text{dom}(\text{finances}) \\
\text{THEN} & \quad \text{credit} := \text{finances}(\ member \ ) \\
\text{END} \end{align*}\]
Specifying a Robust machine

Most of the operations of the **CoffeeClub** machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, **CoffeeClubAPI**, with robust versions of the operations of **CoffeeClub**. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.
Most of the operations of the **CoffeeClub** machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, **CoffeeClubAPI**, with robust versions of the operations of **CoffeeClub**. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.
Most of the operations of the CoffeeClub machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, CoffeeClubAPI, with robust versions of the operations of CoffeeClub. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.
Most of the operations of the **CoffeeClub** machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, **CoffeeClubAPI**, with robust versions of the operations of **CoffeeClub**. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.
Most of the operations of the **CoffeeClub** machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, **CoffeeClubAPI**, with robust versions of the operations of **CoffeeClub**. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.
MACHINE CoffeeClubAPI ( NAME )
INCLUDES CoffeeClub ( NAME )
SETS RESPONSE = { OK,

existing_member ,
not_a_member ,
not_enough_finance ,
not_enough_in_bank }
CoffeeClubAPI.mch II

\[ \text{response} \leftarrow \text{NewMemberAPI}(\text{member}) \triangleq \]
\[
\begin{align*}
\text{PRE} & \quad \text{member} \in \text{NAME} \quad \text{THEN} \\
\text{IF} & \quad \text{member} \in \text{dom}\left(\text{finances}\right) \\
\text{THEN} & \quad \text{response} := \text{existing\_member} \\
\text{ELSE} & \\
\text{response} & := \text{OK} \ |\ | \text{NewMember}(\text{member}) \\
\end{align*}
\]
\[
\text{END} \; ;
\]

\[ \text{response} \leftarrow \text{ContributeAPI}(\text{member}, \text{amount}) \triangleq \]
\[
\begin{align*}
\text{PRE} & \quad \text{member} \in \text{NAME} \land \text{amount} \in \mathbb{N} \quad \text{THEN} \\
\text{IF} & \quad \text{member} \notin \text{dom}\left(\text{finances}\right) \\
\text{THEN} & \quad \text{response} := \text{not\_a\_member} \\
\text{ELSE} & \quad \text{response} := \text{OK} \ |\ | \text{Contribute}(\text{member}, \text{amount}) \\
\text{END} \\
\text{END} \; ;
\]
CoffeeClubAPI.mch

\[\text{response} \leftarrow \text{BuyCoffeeAPI}(\text{member}) \triangleq \]
\[
\text{PRE } \text{member} \in \text{NAME} \text{ THEN }
\]
\[
\text{SELECT } \text{member} \notin \text{dom}(\text{finances}) \text{ THEN }
\]
\[
\text{response} := \text{not\_a\_member}
\]
\[
\text{WHEN } \text{finances}(\text{member}) < \text{coffee} \text{ THEN }
\]
\[
\text{response} := \text{not\_enough\_finance}
\]
\[
\text{ELSE } \text{response} := \text{OK} \parallel \text{BuyCoffee}(\text{member})
\]
\[
\text{END}
\]
\[
\text{END} ;
\]
\[
\text{response}, \text{credit} \leftarrow \text{CreditAPI}(\text{member}) \triangleq \]
\[
\text{PRE } \text{member} \in \text{NAME} \text{ THEN }
\]
\[
\text{IF } \text{member} \notin \text{dom}(\text{finances}) \text{ THEN }
\]
\[
\text{response} := \text{not\_a\_member} \parallel \text{credit} \in \mathbb{N}
\]
\[
\text{ELSE } \text{response} := \text{OK} \parallel \text{credit} \leftarrow \text{Credit}(\text{member})
\]
\[
\text{END}
\]
\[
\text{END} ;
\]
response ← RobBankAPI (amount) ≜

    PRE  amount ∈ \mathbb{N}  THEN
    IF  piggybank < amount THEN
        response := not_enough_in_bank
    ELSE  response := OK ∥ RobBank(amount)
    END

END ;

money ← CashLeftAPI ≜ money ← CashLeft

END