# Table of Contents

1. Overview .................................................................................................................. 1
   1.1. Introduction ........................................................................................................ 1
   1.2. Intended audience, prerequisites, learning outcome ........................................ 1
   1.3. Coverage ........................................................................................................... 1
   1.4. History and origin ............................................................................................ 2
   1.5. Software engineering ....................................................................................... 2
   1.6. Formal methods ............................................................................................... 3
   1.7. The whole picture ............................................................................................ 3
2. Motivation .................................................................................................................. 4
   1. Can you write a correct programme? .................................................................... 4
      1.1. Sum of two integer numbers ........................................................................ 4
      1.2. Calcul of the precondition ............................................................................ 6
   2. Safety critical systems ......................................................................................... 6
   3. Famous failures in software technology ............................................................. 7
      3.1. Ariane 5 ......................................................................................................... 7
      3.2. Floating point bug ....................................................................................... 8
3. Formal methods ...................................................................................................... 10
   1. Formal methods in computer science .................................................................. 10
   2. Limits of the formal methods ............................................................................. 10
   3. Abstract Machine Notation .............................................................................. 11
      3.1. Operator binding and priorities ................................................................... 11
      3.2. Predicates ...................................................................................................... 12
      3.3. Expressions ................................................................................................... 13
      3.4. Sets ................................................................................................................. 13
      3.5. Natural numbers ........................................................................................... 15
      3.6. Relations ........................................................................................................ 16
      3.7. Functions ....................................................................................................... 18
      3.8. Sequences ...................................................................................................... 19
      3.9. Variables ....................................................................................................... 20
      3.10. Generalised Substitutions .......................................................................... 20
4. The B method .......................................................................................................... 22
   1. Overview of the B method .................................................................................. 22
      1.1. Main components ......................................................................................... 22
      1.2. Characteristics ............................................................................................ 23
   2. Industrial use of the B method ........................................................................... 26
   3. The B language .................................................................................................. 28
      3.1. Substitution ................................................................................................... 28
      3.2. Abstract machine structure, proofs ............................................................. 30
      3.3. Abstract machine and implementation ....................................................... 32
5. Software tools of the B method ............................................................................. 34
   1. Atelier B ............................................................................................................. 34
      1.1. Starting a new project in Atelier B ............................................................... 36
      1.2. Creation of an abstract machine ............................................................... 39
      1.3. Type checking .............................................................................................. 41
      1.4. Generating the proof obligations ............................................................... 42
      1.5. Creation of an implementation .................................................................... 46
      1.6. Automatic proving ...................................................................................... 47
      1.7. Interactive proving ...................................................................................... 48
      1.8. B0 check ...................................................................................................... 54
   2. ProB ..................................................................................................................... 55
      2.1. Execution of the (possible) operations ....................................................... 58
      2.2. Animation .................................................................................................... 62
      2.3. Model checking ............................................................................................ 62
      2.4. View of the statespace ................................................................................. 64
      2.5. Violating the operation precondition .......................................................... 66
6. Case studies, examples ......................................................................................... 70

Created by XMLmind XSL-FO Converter.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Lift</td>
<td>70</td>
</tr>
<tr>
<td>2. SmallSet</td>
<td>70</td>
</tr>
<tr>
<td>3. Ticket machine</td>
<td>71</td>
</tr>
<tr>
<td>4. Wallet</td>
<td>71</td>
</tr>
<tr>
<td>5. Maximum search</td>
<td>74</td>
</tr>
<tr>
<td>6. Jukebox</td>
<td>76</td>
</tr>
<tr>
<td>7. Traffic lights</td>
<td>79</td>
</tr>
<tr>
<td>7.1. Choco vending machine</td>
<td>84</td>
</tr>
<tr>
<td>7. Annexes</td>
<td>91</td>
</tr>
<tr>
<td>1. Recommended readings, references</td>
<td>91</td>
</tr>
<tr>
<td>2. Usefull webpages</td>
<td>92</td>
</tr>
</tbody>
</table>
Chapter 1. Overview

1. Introduction

The Eötvös Loránd University has a long tradition in teaching informatics related topics.

The Faculty of Informatics has a very solid foundation in mathematics, thanks to the use of such formal approaches as logic, automates, formal languages, compilers, Petri nets, modelling tools and approaches etc.

The department of Software Technology and Methodology considers as its main task to both teach and research the theoretical bases, methodologies and technologies of programming.

The “Formal Methods in the Software Technology” course, using the B method and Atelier B as a guiding support, taught at our University, aligns this tradition.

2. Intended audience, prerequisites, learning outcome

The intended audience of this teaching material are the Computer Science students at the Master level.

Prerequisites:

• Programming in “classical”, imperative programming languages (such as C, C++, C#, Java, Ada, etc.)

• Logic, predicate calculus

Learning outcome:

• Students will understand the importance of the use of the formal methods such as the B method.

• Students will be able to specify, refine, implement and create executable code, using formal methods.

• Students will be able to prove formally the correctness of the program (specification or implementation) they created.

• Students will be able to use the B method and it’s software tool the Atelier B.

3. Coverage

This teaching material will cover:

• the B method with Classical B

• the Atelier B tool

• the basic creation of specifications, and implementations

• the ProB model checker tool

• ...
This teaching material will not cover:

- Event-B (event driven distributed programming)
- the Rodin tool
- construction of projects containing multiple machine’s and their relations
- the deep understanding of the interactive proving process, the proving methods, the different proving strategies
- the creation of source code and the creation of the executable (machine) code
- ...

4. History and origin

Concepts:
- Jean-Raymond Abrial : B method
- Tony Hoare, Edsgar Dijkstra : weakest precondition, guarded commands
- Cliff Jones : Pre and Post conditions
- Jean-Raymond Abrial : Z specification notation
- ...

Tools:
- Click’n’Prove : Dominique Cansell
- JBtools : Jannis Buttlar
- B-Toolkit : Ib Holm Sorensen
- Atelier-B : Clearsy

5. Software engineering

Software engineering.

- requirements
- specification
- design
- prototyping
- implementation
• verification
• maintenance

6. Formal methods

In computer science, formal methods refers to mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially important in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process.

Formal methods are particularly effective early in development at the requirements and specification levels, but can be used for a completely formal development of an implementation (e.g., a program).

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.\(^1\)

7. The whole picture...

1. software engineering - safety critical systems
2. formal methods - specification
3. abstract machine notation - pseudo-programming language to express abstract specifications and implementation
4. B method - using abstract machine notation, from specification through refinements to implementation and executable code,
5. Atelier B - toolkit to support development using B method, automatic and interactive proofs

\(^1\)http://en.wikipedia.org/wiki/Formal_methods
Chapter 2. Motivation

1. Can you write a correct programme?

<table>
<thead>
<tr>
<th>Simple program</th>
</tr>
</thead>
</table>

**Exercise.**

Write a simple program and write down what does it do!

Any program, any programming language!

Write down in English what it do!

Verify if it’s really does what ou have written!

Check it!

Double check it!

### 1.1. Sum of two integer numbers

<table>
<thead>
<tr>
<th>Sum of two integer numbers</th>
</tr>
</thead>
</table>

**Example description:**

Write a program that calculate the sum (add) of two integer numbers

**Example code:**

```java
import java.util.Scanner;
public class sum{
    public static void main(String [] args){
        int a;
        int b;
        Scanner read=new Scanner(System.in);
        System.out.println("1 Number:");
        a = read.nextInt();
        System.out.println("2 Number:");
        b = read.nextInt();
        System.out.println("Sum: "+ (a+b));
    }
}
```

**Adding two integer numbers**

Do you think that the previous program can add (calculate the sum of) two integer number?

Let’s try it!

Run and test:

```
$java sum
```
Motivation

Everything seems OK...

But are you sure?

Are really sure that the program can calculate two sum of two integer numbers?

Any integer numbers?

| Number: | 5 |
| Number: | 7 |
| Sum: | 12 |

What are the numbers used by computers?

More tests:

| Number: | 2147483646 |
| Number: | 1 |
| Sum: | 2147483647 |

Still OK!

But try:

| Number: | 2147483647 |
| Number: | 1 |
| Sum: | -2147483648 |

Numbers...

Computers, programming languages (mostly) uses fixed-precision numbers. Overflow and underflow (and other "problems") can occur.

Floating point numbers are even worse, see Intel Floating Point Unit (FPU) division problem.

Bigger numbers in Lisp

(defun fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))

fact
(fact 5) 120
(fact 10) 3628800
(fact 1000) 402387260007709377354370243392300398571.. [a 2625 digit number]
(fact 100000)
Number representation / calcul precision limits.
In some programming languages (lisp for example) number representation (calculus precision) is not limited by the processor (CPU) number representation but by the computer memory.
But the calculus precision is (still) limited!

1.2. Calcul of the precondition

<table>
<thead>
<tr>
<th>Sum of two ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

Calculate the sum of two integers where the result can be represented as a 32 bit integer number...

Can you calculate the domain (min and max values) of the two integers to sum?

Can you calculate the precondition of the "sum" operation?

How can you calculate the preconditions of the operations and the domain of the variables for a little more complex program?

Can you correct the program?

Can you tell now exactly what the program is doing?

How can you be sure, that you program works correctly?

Can you prove that?

2. Safety critical systems

Safety critical system.
A life-critical system or safety-critical system is a system whose failure or malfunction may result in:
- death or serious injury to people, or
Motivation

- loss or severe damage to equipment or
- environmental harm.

Risks of this sort are usually managed with the methods and tools of safety engineering.

Infrastructure

(Circuit breaker, Emergency services dispatch systems, Electricity generation, transmission and distribution, Fire alarm, Fire sprinkler, Fuse (electrical), Fuse (hydraulic), Telecommunications, Burner Control systems),

Medicine (Heart-lung machines, Mechanical ventilation systems, Infusion pumps and Insulin pumps, Radiation therapy machines, Robotic surgery machines, Defibrillator machines),

Nuclear engineering (Nuclear reactor control systems, Nuclear reactor cooling systems),

Recreation (Amusement rides, Climbing equipment, Parachutes, SCUBA Equipment),

Transport,

Railway (Railway signalling and control systems, Platform detection to control train doors, Automatic train stop),

Automotive (Airbag systems, Braking systems, Seat belts, Power Steering systems, Advanced driver assistance systems, Electronic throttle control, Battery management system for hybrids and electric vehicles, Electric Park Brake, Shift by wire systems, Drive by wire systems, Park by wire),

Aviation (Air traffic control systems, Avionics, particularly fly-by-wire systems, Radio navigation RAIM, Engine control systems, Aircrew life support systems, Flight planning to determine fuel requirements for a flight),

Spaceflight (Human spaceflight vehicles, Rocket range launch safety systems, Launch vehicle safety, Crew rescue systems, Crew transfer systems) http://en.wikipedia.org/wiki/Life-critical_system

3. Famous failures in software technology

20 Famous Software Disasters

“To err is human, but to really foul things up you need a computer.” -Paul Ehrlich

http://www.devtopics.com/20-famous-software-disasters


3.1. Ariane 5

ARIANE 5 - Flight 501 Failure

See video: http://www.youtube.com/watch?v=gp_D8r_-2hwk
At 36.7 seconds after H0 (approx. 30 seconds after lift-off) the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer."


3.2. Floating point bug

The Pentium FDIV bug was a bug in the Intel P5 Pentium floating point unit (FPU). Because of the bug, the processor would return incorrect results for many calculations used in math and science. Intel blamed the problem on a few missing entries in the lookup table used by the company. The error was rarely encountered by users (Byte magazine estimated that 1 in 9 billion floating point divides with random parameters would produce inaccurate results). However, both the flaw and Intel’s initial handling of the matter were heavily criticized. Intel ultimately recalled the defective processors.

The presence of the bug can be checked manually by performing the following calculation in any application that uses native floating point numbers, including the Windows Calculator or Microsoft Excel in Windows 95/98.

The correct value is:

\[
\frac{4195835}{3145727} = 1.333820449136241002
\]

However, the value returned by the flawed Pentium is incorrect at or beyond four digits:

\[
\frac{4195835}{3145727} = 1.333739068902037589
\]

Formal methods for safety critical systems

- In safety critical systems (and in many other systems) the "correctness" and "provability" of the system is crucial.

- The "correctness" and "provability" can be verified against, (relative) to a specification, description of the desired, required functionality of the system.

- Verify "by hand" the correctness of the system in not enough and/or not possible. A formal, mathematical approach is needed.

\(^1\)http://en.wikipedia.org/wiki/Pentium_FDIV_bug
To "mathematically" (formally) prove the correctness of the system,

- the specification must be a formal specification,
- the system (functionality) must be described (expressed) formally,
- a formal verification tool must be used to verify the correctness against the specification.

Formal methods provide the necessary methodology and tools...
Chapter 3. Formal methods

1. Formal methods in computer science

<table>
<thead>
<tr>
<th>Formal methods</th>
</tr>
</thead>
</table>

In computer science, *formal methods* refers to mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially important in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process. Formal methods are particularly effective early in development at the requirements and specification levels, but can be used for a completely formal development of an implementation (e.g., a program).

<table>
<thead>
<tr>
<th>Security</th>
</tr>
</thead>
</table>

Security.

A simple and clear definition of effective *security* could be: a secure system is a system which does exactly what we want it to do and nothing that we don’t want it to do even when someone else tries to make it behave differently.

2. Limits of the formal methods

<table>
<thead>
<tr>
<th>WIYSIWYG</th>
</tr>
</thead>
</table>

WIYSIWYG = What Is You See Is What You Get

WIYSIWYG = What Is You Specify Is What You Get

<table>
<thead>
<tr>
<th>Proved or Not proved</th>
</tr>
</thead>
</table>

Proved.

From specification trough refinements to implementation and source code.

Not proved.

• "Wish of the customer", informal descriptions, "oral specifications"
• the prover of the B method/Atelier B (itself, however... http://shemesh.larc.nasa.gov/people/cam/publications/bug99.pdf)
• compiler (from source code to machine, executable code)
• the hardware: the computer, the processor, etc...
• the other software elements: operating system, editor, etc...

<table>
<thead>
<tr>
<th>Use of the formal methods at different levels</th>
</tr>
</thead>
</table>
Formal methods can be used at a number of levels¹:

- **Level 0**: Formal specification may be undertaken and then a program developed from this informally. This has been dubbed formal methods lite. This may be the most cost-effective option in many cases.

- **Level 1**: Formal development and verification may be used to produce a program in a more formal manner. For example, proofs of properties or refinement from the specification to a program may be undertaken. This may be most appropriate in high-integrity systems involving safety or security.

- **Level 2**: Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (e.g., in critical parts of microprocessor design).

**Programming language semantics**

As with the sub-discipline of programming language semantics, styles of formal methods may be roughly classified as follows²:

- **Denotational semantics**, in which the meaning of a system is expressed in the mathematical theory of domains. Proponents of such methods rely on the well-understood nature of domains to give meaning to the system; critics point out that not every system may be intuitively or naturally viewed as a function.

- **Operational semantics**, in which the meaning of a system is expressed as a sequence of actions of a (presumably) simpler computational model. Proponents of such methods point to the simplicity of their models as a means to expressive clarity; critics counter that the problem of semantics has just been delayed (who defines the semantics of the simpler model?).

- **Axiomatic semantics**, in which the meaning of the system is expressed in terms of preconditions and postconditions which are true before and after the system performs a task, respectively. Proponents note the connection to classical logic; critics note that such semantics never really describe what a system does (merely what is true before and afterwards).

**Operational and axiomatic semantics**

- If the formal specification is in an operational semantics, the observed behavior of the concrete system can be compared with the behavior of the specification (which itself should be executable or simulateable). Additionally, the operational commands of the specification may be amenable to direct translation into executable code.

- If the formal specification is in an axiomatic semantics, the preconditions and postconditions of the specification may become assertions in the executable code.

### 3. Abstract Machine Notation

**Abstract Machine Notation**

The B-Method uses the notion of *Abstract Machines* to specify and design software systems. Abstract Machines are specified using the Abstract Machine Notation (AMN) which is in turn based on the mathematical theory of Generalised Substitutions.

### 3.1. Operator binding and priorities


Compound formulae (e.g. \( A \rightarrow B \land C \)) are given an unambiguous interpretation by the operator binding rules:

- All operators bind to the left (are left-associative) except “.” which binds to the right.
- Each symbol (e.g. \( \land \)) is given a priority, and the highest priorities bind strongest, e.g. \( A \rightarrow B \land C \) is equivalent to \( A \rightarrow (B \land C) \).
- In case of equal priority the leftmost operator binds the strongest, e.g. \( A \land B \land C \leftrightarrow (A \land B) \land C \).

The priorities of infix operators are listed on the next slide.

### Infix Operator Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>mod * /</td>
</tr>
<tr>
<td>2</td>
<td>- +</td>
</tr>
<tr>
<td>1</td>
<td>..</td>
</tr>
<tr>
<td>0</td>
<td>( \land \setminus \setminus \rightarrow )</td>
</tr>
<tr>
<td>0</td>
<td>&lt;</td>
</tr>
<tr>
<td>0</td>
<td>^ = &lt;= /\ /</td>
</tr>
<tr>
<td>-1</td>
<td>( \leftrightarrow \rightarrow \rightarrow )</td>
</tr>
<tr>
<td>-1</td>
<td>( \gg \gg \gg \gg \rightarrow \rightarrow \leftrightarrow \rightarrow )</td>
</tr>
<tr>
<td>-2</td>
<td>&lt; &lt;:: /&lt;&lt;:/&lt;&lt;:::&lt;-</td>
</tr>
<tr>
<td>-4</td>
<td>= == : ( \leftrightarrow ) ::</td>
</tr>
<tr>
<td>-5</td>
<td>&amp; &amp; &amp;</td>
</tr>
<tr>
<td>-6</td>
<td>( \rightarrow \rightarrow )</td>
</tr>
<tr>
<td>-7</td>
<td>( ;</td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.2. Predicates

<table>
<thead>
<tr>
<th>Predicates</th>
</tr>
</thead>
</table>
Let \( z \) be a Variable List, \( x \) Variable, \( E \) and \( F \) be Expression Lists, \( P \) and \( Q \) be Predicates, and \( S, T \) be Sets.

\( z \notin E \) means that there are no free occurrences in \( E \) of the variables in \( z \).

**General Predicates**

<table>
<thead>
<tr>
<th>( P \land Q )</th>
<th>Conjunction: “( P ) and ( Q )”</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \Rightarrow Q )</td>
<td>Implication: “( P ) implies ( Q )” or “if ( P ) then ( Q )”</td>
</tr>
<tr>
<td>( \neg(P) )</td>
<td>Negation: “Not ( P )”</td>
</tr>
<tr>
<td>( \forall z.(Q \Rightarrow P) )</td>
<td>Universal quantification: “For all ( z ) where ( Q ), ( P )” The predicate ( Q ) must, for each variable ( x ) in the list ( z ), contain a constraining predicate, i.e. ( x:S, x:&lt;:S, x:&lt;:&lt;:S ) or ( x=E ), where ( z \notin S, z \notin E )</td>
</tr>
<tr>
<td>( P \lor Q )</td>
<td>Disjunction: “( P ) or ( Q )”</td>
</tr>
<tr>
<td>( P \iff Q )</td>
<td>Equivalence: “( P ) is equivalent to ( Q )” An abbreviation for ( (P \Rightarrow Q) \land (Q \Rightarrow P) )</td>
</tr>
<tr>
<td>( \exists z.P )</td>
<td>Existential quantification: “For some ( z ), ( P ) holds”. The predicate ( Q ) must, for each variable ( x ) in the list ( z ), contain a constraining predicate, i.e. ( x:S, x:&lt;:S, x:&lt;:&lt;:S ) or ( x=E ), where ( z \notin S, z \notin E ).</td>
</tr>
</tbody>
</table>

**Predicates on Expressions**

<table>
<thead>
<tr>
<th>( E = F )</th>
<th>Equality: ( E ) equals ( F ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \neq F )</td>
<td>Inequality: ( E ) is not equal to ( F ).</td>
</tr>
</tbody>
</table>

**3.3. Expressions**

Let \( E \) and \( F \) be Expressions. \( E,F \) Expression list.

| \( E \mapsto F \) | Ordered pair (maplet). |

**3.4. Sets**

Let \( z \) be a Variable List, \( P \) be a Predicate, \( E \) and \( F \) be Expressions, and \( S \) and \( T \) be sets.

| \( E : S \) | Set membership: the predicate “\( E \) belongs to \( S \)” or “\( E \) is an element of \( S \)” |
### Set Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
| $E /: S$   | Set non-membership: the predicate “$E$ does not belong to $S$”, i.e. $\text{not}(E: S)$.
| $S <: T$   | Set inclusion: the predicate “$S$ is included in $T$”, i.e. “every element of $S$ is also an element of $T$”.
| $S /\:<: T$| Set non-inclusion: the negation of the predicate $S <: T$.
| $S <<: T$  | Strict set inclusion: the predicate “$S$ is included in $T$, but is not equal to $T$”.
| $S /\<<: T$| String set non-inclusion: the negation of the predicate $S <<: T$.

<table>
<thead>
<tr>
<th>Set Comprehension:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>${ z \mid P }$</td>
<td>Set comprehension: the subset such that $P$. The predicate $P$ must, for each variable $x$ in the list $z$, contain a constraining predicate, i.e. $x: S$, $x &lt;: S$, $x &lt;&lt;: S$ or $x = E$, where $z \backslash S$, $z \backslash E$.</td>
</tr>
<tr>
<td>${ z \mid z: S &amp; P }$</td>
<td>Set comprehension: the subset of $S$ such that $P$, e.g. $(x,y \mid x,y: S*T &amp; P)$.</td>
</tr>
<tr>
<td>$S \times T$</td>
<td>Cartesian product: the set of Ordered Pairs whose first component is from $S$ and second component is from $T$.</td>
</tr>
<tr>
<td>$\text{POW}(S)$</td>
<td>Power set: set of all subsets of $S$. $x: \text{POW}(S) \iff x &lt;: S$.</td>
</tr>
<tr>
<td>$S \backslash T$</td>
<td>Set union: the set of elements which are elements of $S$ or $T$.</td>
</tr>
<tr>
<td>$S \backslash \backslash T$</td>
<td>Set intersection: the set of elements which are elements of $S$ and $T$.</td>
</tr>
<tr>
<td>$S - T$</td>
<td>Set difference: the set of elements which are elements of $S$, but not of $T$.</td>
</tr>
<tr>
<td>$()$</td>
<td>Empty set: the set with no elements.</td>
</tr>
</tbody>
</table>

### Set Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{POW1}(S)$</td>
<td>Non-empty subset: Set of all non-empty subsets of $S$. $\text{POW1}(S) = \text{POW}(S) - {}$.</td>
</tr>
<tr>
<td>$\text{FIN}(S)$</td>
<td>Finite subsets: Set of all finite subsets of $S$.</td>
</tr>
<tr>
<td>$\text{FIN1}(S)$</td>
<td>Non-empty finite subsets: Set of all non-empty finite subsets of $S$. $\text{FIN1}(S) = \text{FIN}(S) - {}$.</td>
</tr>
<tr>
<td>${E}$</td>
<td>Singleton set: Provided that $E$ is not an Expression List, and $E: S$, $E$ is of $S$’s.</td>
</tr>
</tbody>
</table>
Formal methods

<table>
<thead>
<tr>
<th>{E,F}</th>
<th>a singleton set: { x \mid x: S \land x = E }.</th>
</tr>
</thead>
</table>

\{E,F\}  
Set enumeration: Provided that F is not an Expression List, this is the set with elements from E together with element F.  
\{E,F\} = \{E\} \cup \{F\}.

| union(U) | Generalised union: the generalised union of a set U of subsets of S  
\(U: \text{POW}(\text{POW}(S))\).  
union(U) =  
\{ x \mid x: S \land \#s.(s: U \land x: s) \}. |
|----------|--------------------------------------------------|

| inter(U) | Generalised intersection: the generalised intersection of a set U of subsets of S  
\(U: \text{POW}(\text{POW}(S))\).  
inter(U) =  
\{ x \mid x: S \land \neg s.(s: U \Rightarrow x: s) \}. |
|----------|--------------------------------------------------|

### 3.5. Natural numbers

**Natural Numbers**

A Natural Number (i.e. a non-negative integer) is an Expression, and the Natural Numbers form an infinite set. Let m and n be Natural Numbers, E and F be Expressions, and P be a Predicate.

**Predicates on Natural Numbers**

<table>
<thead>
<tr>
<th>m &gt; n</th>
<th>Strict inequality: m is greater than n.</th>
</tr>
</thead>
<tbody>
<tr>
<td>m &lt; n</td>
<td>Strict inequality: m is less than n.</td>
</tr>
<tr>
<td>m &gt;= n</td>
<td>Inequality: m is greater than or equal to n.</td>
</tr>
<tr>
<td>m &lt;= n</td>
<td>Inequality: m is less than or equal to n.</td>
</tr>
</tbody>
</table>

**Natural Number Expressions**

<table>
<thead>
<tr>
<th>NAT</th>
<th>The set of natural numbers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAT1</td>
<td>The set of non-zero natural numbers.</td>
</tr>
<tr>
<td>min(S)</td>
<td>Minimum of a non-empty subset, S, of NAT.</td>
</tr>
<tr>
<td>max(S)</td>
<td>Maximum of a non-empty finite subset, S, of NAT.</td>
</tr>
<tr>
<td>m+n</td>
<td>Addition: the sum of m and n.</td>
</tr>
<tr>
<td>m-n</td>
<td>Difference: the difference of m and n (defined for m &gt;= n).</td>
</tr>
<tr>
<td>m*n</td>
<td>Product: the product of m and n.</td>
</tr>
<tr>
<td>m/n</td>
<td>Division: the integer division of m by n.</td>
</tr>
</tbody>
</table>
### Natural Numbers

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m mod n</td>
<td>Remainder: the remainder of the integer division of m by n.</td>
</tr>
<tr>
<td>n .. m</td>
<td>Interval: the set of non-negative integers between n and m inclusive.</td>
</tr>
</tbody>
</table>

### Relations

A **Relation** is a set of Ordered Pairs. Therefore, any set operation may also be applied to Relations. Let S, T, U and V be sets, and r, r1, r2 be relations from S to T, and let E and F be Expressions. Also let $s <: S$ and $t <: T$.

#### Relational Expressions

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S &lt;&gt; T</td>
<td>Relation: Set of relations from S to T. Equivalent to $\text{POW}(S \times T)$.</td>
</tr>
<tr>
<td>dom(r)</td>
<td>Domain of r:</td>
</tr>
<tr>
<td></td>
<td>The set ${x \mid x: S \land #y.(x,y: r)}$.</td>
</tr>
<tr>
<td>ran(r)</td>
<td>Range of r:</td>
</tr>
<tr>
<td></td>
<td>The set ${y \mid y: T \land #x.(x,y: r)}$.</td>
</tr>
<tr>
<td>p;q</td>
<td>Relational composition: Composition of relations p and q, where p: $S &lt;&gt; T$ and q: $T &lt;&gt; U$.</td>
</tr>
<tr>
<td></td>
<td>The set ${x,z \mid x,z: S \times U \land #y.(y: T \land x,y: p \land y,z: q)}$. Also denoted by $q \circ p$.</td>
</tr>
<tr>
<td>q circ p</td>
<td>Composition of relations q and p. The same as $p;q$.</td>
</tr>
<tr>
<td>id(S)</td>
<td>Identity on S. The set ${x,y \mid x,y: S \times S \land x = y}$.</td>
</tr>
</tbody>
</table>
## Relations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s \mid r )</td>
<td>Restriction of ( r ) by ( s ). Also known as domain restriction. The relation formed from ( r ) by keeping only the pairs where the first element is in ( s ). The set ( {x,y \mid x,y \in r \land x \in s} ).</td>
<td></td>
</tr>
<tr>
<td>( r \mid t )</td>
<td>Co-restriction of ( r ) by ( t ). Also known as range restriction. The relation formed from ( r ) by keeping only those pairs where the last element is in ( t ). The set ( {x,y \mid x,y \in r \land y \in t} ).</td>
<td></td>
</tr>
<tr>
<td>( s \langle r )</td>
<td>Anti-restriction of ( r ) by ( s ). Also known as domain subtraction. The relation formed from ( r ) by keeping only those pairs where the first element is in the complement of ( s ). The set ( {x,y \mid x,y \in r \land x \in S - s} ).</td>
<td></td>
</tr>
<tr>
<td>( r \rangle t )</td>
<td>Anti-co-restriction of ( r ) by ( t ). Also known as range subtraction. The relation formed from ( r ) by keeping only those pairs where the last element is in the complement of ( t ). The set ( {x,y \mid x,y \in r \land y \in T - t} ).</td>
<td></td>
</tr>
<tr>
<td>( r^- )</td>
<td>Inverse of ( r ). The relation formed from ( r ) by interchanging the elements of each pair. The set ( {y,x \mid y,x \in T \ast S \land x,y \in r} ).</td>
<td></td>
</tr>
<tr>
<td>( r[s] )</td>
<td>Image of set ( s ) under relation ( r ). The set consisting of all those elements related to some element in the set ( s ) through relation ( r ). The set ( {y \mid y \in T \land #x. (x \in s \land x,y \in r)} ).</td>
<td></td>
</tr>
<tr>
<td>( r_1 &lt;+ r_2 )</td>
<td>Overriding of ( r_1 ) by ( r_2 ). The set ( \text{dom}(r_2) \triangledown r_1 \setminus r_2 ).</td>
<td></td>
</tr>
<tr>
<td>( r_1 \rightarrow r_2 )</td>
<td>Overriding of ( r_2 ) by ( r_1 ). The set ( r_2 &lt;+ r_1 ).</td>
<td></td>
</tr>
<tr>
<td>( p \times q )</td>
<td>Direct product of ( p ) and ( q ), where ( p: S \leftrightarrow U ) and ( q: S \leftrightarrow V ). The set ( {x, (y,z) \mid x, (y,z) \in S \times (U \times V) \land x,y \in p \land x,z \in q} ).</td>
<td></td>
</tr>
<tr>
<td>( p</td>
<td></td>
<td>q )</td>
</tr>
</tbody>
</table>
### Relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterate(r,n)</td>
<td>The nth iterate of r (where n: NAT), i.e. r composed with itself n times (defined only for r: S &lt;-&gt; S). iterate(r,0) = id(S) and iterate(r,n+1) = r;iterate(r,n).</td>
</tr>
<tr>
<td>closure(r)</td>
<td>The reflexive transitive closure of r (defined only for r: S &lt;-&gt; S). closure(r) = UNION(n).(n: NAT</td>
</tr>
<tr>
<td>prj1(S,T)</td>
<td>Projection: prj1(S,T) = { (x,y),z</td>
</tr>
<tr>
<td>prj2(S,T)</td>
<td>Projection: prj2(S,T) = { (x,y),z</td>
</tr>
</tbody>
</table>

### 3.7. Functions

A **Function** is a Relation with the additional property that each element of the domain is related to a unique element in the range. Any operation applicable to Relations may also be applied to Functions. Let S and T be sets, z a Variable List, E be an Expression, and P be a predicate.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S +--&gt; T</td>
<td>Set of partial functions from S to T (also known as ‘many-to-one relations’). The set { r</td>
</tr>
<tr>
<td>S --&gt; T</td>
<td>Set of total functions from S to T. The set { f</td>
</tr>
<tr>
<td>S ++&gt; T</td>
<td>Set of partial injections from S to T (also known as ‘one-to-one relations’). The set { f</td>
</tr>
<tr>
<td>S --&gt;&gt; T</td>
<td>Set of total injections from S to T. The set S --&gt;&gt; T \ / \ S --&gt; T.</td>
</tr>
<tr>
<td>S +--&gt;&gt; T</td>
<td>Set of partial surjections from S to T.</td>
</tr>
</tbody>
</table>
The set \( \{ f \mid f: S \rightarrow T & \text{ran}(f) = T \} \).

\( S \rightarrow T \)  
Set of total surjections from \( S \) to \( T \).

\( S \rightarrow T \)  
Set of bijections from \( S \) to \( T \).

\( S \rightarrow T / \setminus s\rightarrow T \)  
Function construction.

\( \%z.(z: S & P \mid E) \)  
The function \( \{x,y \mid z: S & y=E \mid P \} \) where \( y\not\in E \) and \( y\not\in P \), with domain \( \{ z \mid z: S & P \} \).

\( \%z.(P \mid E) \)  
Function construction.

\( f(x) \)  
For \( x: \text{dom}(f) \), \( f(x) \) denotes the value of the function \( f \) at \( x \), i.e. \( x \rightarrow f(x): f \).

### 3.8. Sequences

A sequence over a set \( S \) is a function from \( \text{NAT} \) to \( S \) whose domain is an interval \( 1..n \) for some natural number \( n \). Let \( s, t \) be sequences of elements from \( S \), \( e \) be an element of \( S \), and \( E \) and \( F \) be expressions.

\(<\>\)  
The empty sequence.

\( \text{seq}(S) \)  
The set of finite sequences of elements from \( S \).

\( \text{seq1}(S) \)  
The set of finite non-empty sequences of elements from \( S \).
\( \text{seq1}(s) = \text{seq}(s) - \{<\>\} \).

\( \text{iseq}(S) \)  
The set of injective sequences of elements from \( S \).
\( \text{iseq}(S) = \text{seq}(S) / \setminus (\text{NAT1} \rightarrow T) \).

\( \text{perm}(S) \)  
The set of bijective sequences of elements from a finite set \( S \). A sequence belonging to \( \text{perm}(S) \) is said to be a ‘permutation’ of \( S \). For finite \( S \), \( \text{perm}(S) = 1..\text{card}(S) \rightarrow T \).

\( s^t \)  
The concatenation of sequences \( s \) and \( t \).

\( e \rightarrow s \)  
The sequence formed by prepending \( e \) to \( s \).

\( s < e \)  
The sequence formed by appending \( e \) to \( s \).

\( \text{size}(s) \)  
The size of the finite sequence \( s \).
### Sequences

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>([E])</td>
<td>Provided that E is not an Expression List, ([E]) is the singleton sequence with element E, i.e ([E] = E \rightarrow \langle \rangle).</td>
</tr>
<tr>
<td>([E,F])</td>
<td>Provided F is not an Expression List, then this is ([E]) with F appended. Equivalent to ([E] \leftarrow F).</td>
</tr>
<tr>
<td>(\text{rev}(s))</td>
<td>The reverse of (s).</td>
</tr>
<tr>
<td>(s /|\langle n\rangle)</td>
<td>The sequence obtained from (s) by retaining only its first (n) elements, where (n \leftarrow \text{size}(s)).</td>
</tr>
<tr>
<td>(s |\langle n\rangle)</td>
<td>The sequence obtained by removing the first (n) elements of (s), where (n \leftarrow \text{size}(s)).</td>
</tr>
<tr>
<td>(\text{first}(s))</td>
<td>The first element of the non-empty sequence (s).</td>
</tr>
<tr>
<td>(\text{last}(s))</td>
<td>The last element of the non-empty sequence (s).</td>
</tr>
<tr>
<td>(\text{tail}(s))</td>
<td>The sequence (s) with its first element removed ((s) must be non-empty).</td>
</tr>
<tr>
<td>(\text{front}(s))</td>
<td>The sequence (s) with its last element removed ((s) must be non-empty).</td>
</tr>
<tr>
<td>(\text{conc}(s))</td>
<td>The generalised concatenation of a sequence of sequences, (s). For a sequence (t), (\text{conc}(&lt;&gt; \rightarrow &lt;&gt; \leftarrow )) and (\text{conc}(s \leftarrow t) = \text{conc}(s) \leftarrow ^{t}).</td>
</tr>
</tbody>
</table>

### 3.9. Variables

#### Variables, Variable Lists and Identifiers

A Variable is an Identifier.

An Identifier is a string of length 2 or more of alphanumeric characters (a to z, A to Z, 0 to 9 ASCII codes) or underscore ‘_’, with at least one letter.

An Upper Case Identifier is an Identifier made only from upper case letters and underscore.

An Infix operator is either a string of non-alphanumeric characters (excluding ‘_’ ‘ ’ ‘$’ and ‘?’) or an Identifier declared as an Infix Operator in the AMN Symbol Table, e.g. ‘mod’.

Let \(z\) be a Variable List and \(x\) be a Variable.

\(z,x ((z,x))\) is a Variable List.

### 3.10. Generalised Substitutions

#### Generalised Substitutions

Let \(x\) be a Variable, \(z\) be a variable List, \(P\) and \(R\) be predicates, \(E\) and \(F\) be Expressions, and \(S, T\) be Generalised Substitutions.

\([S]P\) is a predicate obtained by replacing the variables in \(P\) according to the rules below.
Formal methods

| (x:= E)F | An expression obtained by replacing all free occurrences of x in F by E. |
| z\A | Non-freeness: z is not free in E, i.e. there are no free occurrences of z in the Predicate or Expression A. |
| x:= E | Simple substitution. Substitute E for x in a Predicate or Expression formula. (Note that the applicability of a simple substitution on a formula is limited by non-freeness conditions when the formula is a quantified expression or a set comprehension). |
| x,y:= E,F | Simultaneous substitution. Substitute several Expressions for several Variables. |
| x:=E || y:=F | Simultaneous substitution. A form equivalent to the above simultaneous substitution. |

<table>
<thead>
<tr>
<th>Generalised Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>P --&gt; S</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>(#z.S)</td>
</tr>
<tr>
<td>S;T</td>
</tr>
<tr>
<td>skip</td>
</tr>
</tbody>
</table>
Chapter 4. The B method

1. Overview of the B method

1.1. Main components

Abstract machine.
In the first and the most abstract version, which is called Abstract Machine, designer should specify the goal of the design.

Refinement.
Then, during a refinement step, he may pad the specification in order to clarify the goal or to turn the abstract machine more concrete by adding more details about data structures and algorithms that explain how the goal may be achieved. The new version, which is called Refinement, should be proven to be coherent and including all the properties of the Abstract Machine.

Implementation.
The refinement in its turn may be refined one or many times to obtain a deterministic version which is called Implementation. During all of the development steps the same notation is used and the last version may be translated to Ada, C or C++ language.

From specification to source code

1 http://en.wikipedia.org/wiki/B_method
1.2. Characteristics

Characteristics of the B method

- Use same language in specification, design and programation.
- Mechanism include encapsulation and data locality.
- Clear and close introduction for refinement concept.
- Originated in the 1980s by Jean-Raymond Abrial.
- B method is a tool-supported formal methods based around AMN (Abstract Machine Notation), used in the development of correct software.
- B method has been used in some major safety-critical system applications in Europe.

Brief Description of the B Method: B method notation

The B method uses a notation based on the mathematical concept of theory sets. The notation is useable throughout the development cycle, meaning that you obtain a uniform formal setting which replaces conventional specifications from preliminary design to code generation.

²http://en.wikipedia.org/wiki/B_-_Method
The initial expression of need is generally produced using natural language, or combined descriptions (e.g., charts, automatons, logical tables, Petri networks and methods such as SADT or SA-RT).

The B development process begins with the construction of a model that incorporates all descriptions of the need and describes the system’s main state variables. Also described are the properties (or invariants) which these variables must meet and their transformation by services (or operations). The obtained B model constitutes the specifications of what the system must implement (the "what").

The B model is then refined until a complete implementation of the software system is obtained (the "how"). Several refinements may be needed to satisfy the specifications.

Using B in the development of a system is therefore about:

• removing all ambiguity from the interpretation of the need,
• building specifications that are coherent and compliant with the need (the “model”),
• developing the software system implements the specifications, in successive stages.

The coherence of the model, and the conformity of the final program in relation to this model are guaranteed by mathematical proofs. Automatic proof tools, such as those provided by Atelier B, must be used to demonstrate these proofs concretely.

The formal definition of the substitutions enables Proof Obligation to be demonstrated, to ensure that an operation call preserves the static properties of the abstract machine (the "invariant").

The refinement mechanism consists in reformulating the variables and operations of the abstract machine successively, to obtain a module that eventually constitutes a computer program. The intermediary stages of reformulation are called "refinements" and the final level of refinement is called “implementation.” Each B component (abstract machine, refinement or implementation) is structured using a unique language, the B Language.

During each refinement, it is necessary to prove that the behavior of an operation is compatible with the operation stated at the abstract level. The code of an implementation will thus effectively conform with the specifications of the corresponding abstract machine.
The B method is designed to provide a notation and a method for requirement modeling, software interface specification, software design, implementation and maintenance, thus supporting the major phases of a software process. Incremental construction of layered software as well as its incremental verification and validation are the guiding principles of the B method.

Tools supports the method over the entire software process and comprises a large suite of tools which can run automatically or interactively. All tools are integrated into a window-based development environment. The tool supports the incremental construction of the software. The validation processes are supported by static analysis which employ and checking, by dynamic analysis using simulation, as well as proof of correctness using an integrated theorem prover.

### B method - Abstract Machine Notation

The B Method is a collection of mathematically based techniques for the specification, design and implementation of software components. Systems are modeled as a collection of interdependent Abstract Machines, for which an object-based approach is employed at all stages of development. An Abstract Machine is described using the Abstract Machine Notation (AMN). A uniform notation is used at all levels of description, from specification, through design, to implementation.

AMN is a state-based formal specification language in the same school as VDM and Z. An Abstract Machine comprises a state together with operations on that state. In a specification and a design of an Abstract Machine the state is modeled using notions like sets, relations, functions, sequences etc. The operations are modeled using Pre- and Post-conditions using AMN.

In an implementation of an abstract machine the state is again modeled using a set-theoretical model, but this time we already have an implementation for the model. The operations are described using a pseudo-programming notation that is a subset of AMN.
B method: checking specifications for consistency and correctness

The B Method prescribes how to check the specification for consistency (preservation of invariant) and how to check designs and implementations for correctness (correctness of data refinement and correctness of algorithmic refinement).

The B-Method further prescribes how to structure large designs and large developments, and promotes the re-use of specification models and software modules, with object orientation central to specification construction and implementation design.

A great deal of attention has been paid to making the notational aspect of the method as simple as possible. To the engineer, the formal notation looks like a simple pseudo programming notation. And as mentioned above, there is no real distinction between the specification notation and the programming notation.

2. Industrial use of the B method

Industrial use of the B-method

- Paris metro line 14
  - 117.000 B loc
  - 29.000 proofs
  - 87.000 ADA loc
- Paris CDG airport metro
  - 180.000 B loc
  - 43.000 proofs
  - 140.000 ADA loc
  - 98% automatic proved
Ateleir B used in metros around the world

3. The B language

3.1. Substitution

Definition of the substitution

the \([x := E] F\) is: change all free occurrence of the \(x\) in \(F\) to \(E\)

Example for free and binded variables: \(\forall n. (n \in \mathbb{N} \Rightarrow n > x)\) \(x\) : free variable, \(n\) : binded variable

Substitution, initialisation - example

VARIABLES
z

INVARIANT

R

INITIALISATION

*Source: http://www.tools.clearsy.com/*
The B method

... 

\[ H[R \]

VARIABLES
\begin{itemize}
  \item \texttt{tanulo}
\end{itemize}

INVARIANT
\begin{itemize}
  \item \texttt{tanulo} \subseteq DIAKOK
\end{itemize}

INITIALISATION
\begin{itemize}
  \item \texttt{tanulo} := \emptyset
\end{itemize}

\[ \text{the initialisation keeps the invariant property} \]

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{Definition of the operations with substitution} & \\
\hline
\hline
\texttt{[G;H]P} & \texttt{[G][H]P} \\
\hline
\texttt{[PRE P THEN G END]Q} & P \land [G]Q \\
\hline
\texttt{[IF P THEN G ELSE H END]Q} & (P \Rightarrow [G]Q) \land (\neg P \Rightarrow [H]Q) \\
\hline
\texttt{[CHOICE G OR H END]Q} & [G]Q \land [H]Q \\
\hline
\texttt{[ANY x WHERE P THEN G END]Q} & \forall x. (P \Rightarrow [G]Q) \\
\hline
\end{tabular}
\end{table}
The B method

<table>
<thead>
<tr>
<th>[G; WHILE P DO H]</th>
<th>([G][I] ∧ (I ∧ ¬P ⇒ Q) ∧</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIANT E</td>
<td>(I ⇒ E ∈ N) ∧</td>
</tr>
<tr>
<td>INARIANT I</td>
<td>(I ∧ P ⇒ [y := E][H]E &lt; y) ∧</td>
</tr>
<tr>
<td>END] Q</td>
<td>(I ∧ P ⇒ [H]I)</td>
</tr>
</tbody>
</table>

**Operation, substitution verification - example**

INVARIANT rfree ⊆ RES

INITIALISATION rfree := ∅

OPERATIONS alloc(rr) = PRE rr ∈ rfree

THEN rfree := rfree − {rr} END;

rfree := ∅ rfree ⊆ RES

∅ ⊆ RES

][PRE P THEN G END] Q P ∧ [G]Q

rfree ⊆ RES ∧ rr ∈ rfree ⇒ [PRE rr ∈ rfree

THEN rfree := rfree − {rr} END]rfree ⊆ RES

rfree ⊆ RES ∧ rr ∈ rfree ⇒ rr ∈ rfree ∧ [rfree := rfree − {rr}]rfree ⊆ RES

rfree ⊆ RES ∧ rr ∈ rfree ⇒ [rfree := rfree − {rr}]rfree ⊆ RES

rfree ⊆ RES ∧ rr ∈ rfree ⇒ rfree − {rr} ⊆ RES

**3.2. Abstract machine structure, proofs**

**Abstract machine structure**

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>machine (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>P</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>y</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>Q</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>z</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>R</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>H</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td></td>
</tr>
</tbody>
</table>
Proof of the coherence of an abstract machine

\[
\begin{align*}
iope &= \text{PRE } \text{L THEN } \text{G END} \\
\text{END}
\end{align*}
\]

\[
\begin{array}{|c|l|}
\hline
\text{machine, opname} & : \text{identifiers} \\
\hline
\text{x, x, z} & : \text{identifier lists} \\
\hline
P, Q, R, L & : \text{predicates} \\
\hline
G, H & : \text{substitutions} \\
\hline
\end{array}
\]

. 

\[\exists x. P\]

constraints consistence

. 

\[P \Rightarrow \forall y. Q\]

properties consistence

. 

\[P \land Q \Rightarrow \exists z. R\]

"implementability"

. 

\[P \land Q \Rightarrow [G] R\]

correctness of the initialisation

. 

\[P \land Q \land R \land L \Rightarrow [G] R\]
correctness of the operations

3.3. Abstract machine and implementation

<table>
<thead>
<tr>
<th></th>
<th>absztrak</th>
<th>concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>$x_A$</td>
<td>$x_C$</td>
</tr>
<tr>
<td>invariant</td>
<td>$P_A$</td>
<td>$P_C$</td>
</tr>
<tr>
<td>initialisation</td>
<td>$G_A$</td>
<td>$G_C$</td>
</tr>
<tr>
<td>op. output</td>
<td>$y$</td>
<td>$y$</td>
</tr>
<tr>
<td>op. precondition</td>
<td>$Q_A$</td>
<td>$Q_C$</td>
</tr>
<tr>
<td>op. substitution</td>
<td>$H_A$</td>
<td>$H_C$</td>
</tr>
</tbody>
</table>

\[
[G_C] \vdash [G_A] \vdash P_C
\]
correctness of the initialisation

\[
P_A \land P_C \land Q_A \Rightarrow Q_C \land [H'_C] \vdash [H_A] \vdash (P_C \land y' = y)
\]
correctness of an operation

(the $H'_C$ is $H_C$ where $y$ is substituted by $y'$)

<table>
<thead>
<tr>
<th></th>
<th>absztrak</th>
<th>concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>$xx$</td>
<td>$yy$</td>
</tr>
<tr>
<td>invariant</td>
<td>$xx \in 0..100$</td>
<td>$yy \in 0..100 \land xx = yy$</td>
</tr>
<tr>
<td>initialisation</td>
<td>$xx := 0$</td>
<td>$yy := 0$</td>
</tr>
</tbody>
</table>

\[
[G_C] \vdash [G_A] \vdash P_C
\]
correctness of the initialisation
The B method

\[
\begin{align*}
[yy := 0] & \models [xx := 0] \models (yy \in 0..100 \land yy = xx) \\
[yy := 0] & \models (yy \in 0..100 \land yy = 0) \\
[yy := 0] & \models (yy \in 0..100 \land yy = 0) \\
        & \models 0 \in 0..100 \land 0 = 0
\end{align*}
\]

Structure of a B specification

- sets, constants, variables
- invariant
- initialisation
- operations
Chapter 5. Software tools of the B method

1. Atelier B

"Atelier B, the industrial tool to efficiently deploy the B method"

"Developed by ClearSy, Atelier B is an industrial tool that allows for the operational use of the B Method to develop defect-free proven software (formal software). Two versions are available: Community Edition available to anyone without any restriction, Maintenance Edition for maintenance contract holders only.

It is used to develop safety automatisms for the various subways installed throughout the world by Alstom and Siemens, and also for Common Criteria certification and the development of system models by ATMEI and STMicroelectronics. Additionally, it has been used in a number of other sectors, such as the automotive industry, to model operational principles for the onboard electronics of three car models. Atelier B is also used in the aeronautics and aerospace sectors."

Atelier B is a tool enabling the operational use of B method. In a coherent environment, it provides many functions for managing projects in B language.

These functions can be divided into three categories:

• proof aid, to demonstrate proof obligations using suitable proof tools
• development aid: automatic management of dependence between B components,
• user comfort tools: graphical representation of projects, display of project status and statistics, project archiving.

Atelier B is either used via a Man Machine Interface in QT format or using the commands directly (command mode). Atelier B is multi-user. Tasks that can be automated during project development are the following:

• syntax verification of components
• automatic proof obligation generation
• automatic translation of B installations to C or Ada language

From now on, Atelier B is available in Windows, Linux, Mac OS AND Solaris.

---

1 Atelier B presentation from Clearsy at http://www.atelierb.eu/en/
2 Presentation of the Atelier B tool is from http://www.atelierb.eu/en/atelier_b_tools/
• Automatic refinement

Integration of an automatic refinement tool (BART). BART enables refinement and implementation generation using a refinement rule base that can be expanded by the user. BART operates on a refinement rule basis. Additional refinement rules can be added for refinement personalisation of certain components.

Atelier B - The main functions of Atelier B

• Syntax analysers

These are used to carry out all syntax verifications of files in B language:

• a model editor has been integrated into Atelier B. This integrates a syntax analyser, automatic completion as well as navigation functions throughout the model.

• the Type Checker first carries out a grammatical verification of a B component, and then a certain number of contextual verifications including the type control and the control of identifier scopes. Components have to pass through the Type Checker before being proved and before any other of the verifications presented below

• the B0 Checker carries out verifications specific to the BO language used in the installations (a sub-division of B language) to ensure that they can be translated.

• the project checker checks all the components of a project to control its architecture (the links between the components). The project must have been checked before the final translation of the project.

• B models can be saved in pdf, rtf and LaTeX formats.

Atelier B - The main functions of Atelier B

• Proof tools

These have the following functions:

• the automatic generation of the proof obligations using the components in B language

• a B component is correct when its proof obligations are demonstrated

• proof in automatic mode: most of the proof obligations are demonstrated without user intervention

• the proof in interactive mode used when the automatic mode has failed: the user guides the prover in its proof obligation demonstration using interactive commands (lemma additions, proof by case etc.)

• the predicate prover: predicate demonstration: demonstration of rules added by the user

• viewing proof obligations, their origin and their status (trivial, proved, non-proved)

• the management of a validated rule base including more than 2 200 rules

Atelier B - The main functions of Atelier B

• Automatic translators

The installations make up the coding stage for a development in B language. They are written using a B language sub-assembly, similar to an imperative programming language. In order to facilitate code generation on any target system, the installations are translated automatically to standard programming language. The programmes obtained can then be compiled and assembled on the target machine to produce the executable software.
• The graphic representation of projects

It is used for the graphic representation of the components of a project and their links, by positioning them automatically on the graph. The user can choose different display options, for example the type of links to be viewed, the view of the whole dependence graph of a project or the dependence graph of a component.

---

**Atelier B - The main functions of Atelier B**

• Project management

Atelier B offers large size project management services (including for example 500 components). In particular:

• Atelier B used by several users in a network. These users can work on the same project at the same time

• to archive and restore a whole project

• to architecture a project or several sub-projects or libraries. As such, Atelier B enables large and modular developments by several developer teams

• to view the overall status of a project, by supplying for each component, its status (passed to Type Checker, translated to C or to Ada), the number of proof obligations and the percentage proved

• to generate automatically the dependencies between the project’s components. As such, to carry out an action (passage through the Type Checker, through the proof obligation generator etc.) on a selection of components, Atelier B reports the action(s) required for the components on which it depends.

---

**Download and installing Atelier B**


"The new licence associated with the Atelier B V4 is distributed free of charge to all those who wish to use Atelier B for research, teaching and development of their industrial projects. As soon as the tool is used for the first time it is allocated for an unlimited time.

The atelier B V4 comprises all the tools required for developing an industrial project and is available in WINDOWS, MAC, LINUX, SOLARIS

Users, companies and research or teaching organisations can subscribe to a maintenance contract for the intermediate versions (corrective or with new functional features).

The licence contract supplied with Atelier B can be downloaded and is available for consultation here in French and English."

**1.1. Starting a new project in Atelier B**

The first Atelier B project

1. Launch Atelier B

2.  

Creation of a new project (setting project parameters)

3.

Creation (edit) of an abstract machine (specification)

4.

Type check

5.

Generation of the proof obligations

6.

using the automatic prover

7.

B0 check

Launching Atelier B

Opening screen with recent projects, tutorial, websites...

Local projects...
Atelier B ...

Creating (naming) a new (software development) project

Atelier B ...

Setting project parameters: directories, configuration, etc..
1.2. Creation of an abstract machine

Creating (naming) a new specification (machine)...

In Atelier B vocabulary specification called machine...

Preview of the (empty) specification
Opening and editing a specification

Specification (with some syntax errors indicated)
1.3. Type checking

Specification type checked...
1.4. Generating the proof obligations

Atelier B ...

Generation of proof obligations (POs)

Proof obligations generated, # of proof obligations : 4, # proved: 0, # of unproved: 4

Atelier B ...

Using the automatic prover
Software tools of the B method

Atelier B ...

B0 check

B0 check is : OK

Specification, refinement and proof

1.

creation of an abstract machine
2. creation of an implementation of the abstract machine

3. interactive proof of the abstract machine and the implementation

4. interactive proof of the unproved proof obligations

Creation of a new project

Setting project parameters
Creation of a new abstract machine (a new specification)

Edit of the abstract machine
1.5. Creation of an implementation

**Loop specification**

```
MACHINE Looptest2
CONCRETE_VARIABLES
  xx
INVARIANT
  xx : 0..10 --> NAT
INITIALISATION
  xx := (0..10) * {0}
OPERATIONS
  nulla=
    BEGIN
    xx := (0..10) * {0}
    END
END
```

**Loop implementation**

```
IMPLEMENTATION Looptest2_i
REFINES Looptest2
INITIALISATION xx := (0..10) * {0}
OPERATIONS
  nulla=
    VAR ii IN
    ii := -1;
    WHILE ii /= 10 DO
      xx(ii+1):=0;
      ii := ii + 1
    INVARIANT
```
\[
\begin{align*}
ii & : \text{INTEGER} & \\
xx & : 0..10 \rightarrow \text{NAT} & \\
i1 & : -1..10 & \\
!jj. (jj:(0..ii) \rightarrow xx(jj)=0) & \\
\text{VARIANT} & \\
i1=i1 & \\
\text{END} & \\
\text{END} & \\
\text{END} & \\
\end{align*}
\]

Atelier B ...

Type checked, PO generated, Unproved...

1.6. Automatic proving

Atelier B ...

Abstract machine automatic proved 100% (4/4)
1.7. Interactive proving

Launching the interactive prover. 14 POs proved (PO1-14), 2 POs (PO15-16) unproved
Hypotheses and the predicate to prove...

first step is "Prove" (pr)
next step is "Predicate Prover with first level of hypothesis" (pp1)

success, the prof obligation (PO15) is proved...

interactive proving of the next (PO16) prof obligation, hypotheses and the predicate to prove...
first step is "Prove" (pr)... 

next step is "Predicate Prover with first level of hypothesis" (pp1)
success, the prof obligation (PO15) is proved...

Atelier B ... All the proof obligations are proved. Leaving (exit) the interactive prover, propose to save "User Pass" (the trace of the proving)

Atelier B ... Creation of an "User Pass"...
Atelier B ...

Preview of the "User Pass" : for the operation "nulla", "pr" then "pp1"

Atelier B ...

Now everything (abstract machine and the implementation) is proved, 0 Unproved...
1.8. B0 check

The B0 check...

Atelier B ...

The B0 check is OK...
Launching the ProB animation...

2. ProB

The ProB tool...

ProB.

ProB is an animator and model checker for the B-Method (see the B-Method site of Clearsy). It allows fully automatic animation of many B specifications, and can be used to systematically check a specification for range of errors. The constraint-solving capabilities of ProB can also be used for model finding, deadlock checking and test-case generation.

ProB...

Starting the tool

http://www.stups.uni-duesseldorf.de/ProB/index.php5/The_ProB_Animator_and_Model_Checker
ProB can be launched from AtelierB or standalone.

The Gearbox example

The Gearbox example: abstract machine to "simulate" the functionality of a car’s gearbox.

Desired functionality:

- 5 speed gearbox (0-5, no reverse)
- turn on-off the engine
- shift gear up and down
- use the clutch

```
MACHINE
gear
CONSTANTS
  GEARS
VARIABLES
  power, actual_gear, embreagem
PROPERTIES
  GEARS:NATURAL &
  GEARS = 5
INVARIANT
  actual_gear:NATURAL &
  power: BOOL &
  embreagem: BOOL &
  actual_gear >= 0 &
  actual_gear <= GEARS
INITIALISATION
  power := FALSE ||
  actual_gear := 0 ||
  embreagem := FALSE
```
Gearbox example

OPERATIONS
  turn_on =
  PRE actual_gear = 0 & embreagem = TRUE & power = FALSE
  THEN power := TRUE
  END;

  gear_up =
  PRE actual_gear < 5 & embreagem = TRUE
  THEN actual_gear := actual_gear+1
  END;

  gear_down =
  PRE actual_gear-1 >= 0 & embreagem = TRUE
  THEN actual_gear := actual_gear-1
  END;

change_embreagem =
  IF (embreagem = TRUE)
  THEN embreagem := FALSE
  ELSE embreagem := TRUE
  END;

status <- get_embreagem =
  status := embreagem
END

ProB ...

Loading a B specification
2.1. Execution of the (possible) operations

Executing (possible) operations

ProB ...

Executing (possible) operations

ProB ...
Executing (possible) operations
Executing (possible) operations

ProB ...

Executing (possible) operations
Executing (possible) operations
2.2. Animation

Animation (random or defined amount of steps)

2.3. Model checking

Model checking
Software tools of the B method

ProB ...

Model checking

ProB ...

Model checking
2.4. View of the statespace

Counting:

StateSpace of the specification...
Statespace of the specification...
2.5. Violating the operation precondition

ProB ...

Breaking an operation precondition...

ProB ...

Breaking an operation precondition... violated invariant
ProB ...

Breaking an operation precondition... violated invariant

ProB ...

Changing an operation precondition..
Changing an operation precondition...

Proved, but not good

Remember... slide 3.2... "What Is You Specify Is What You Get"

One can prove a "wrong", "bad" system...
You can prove system doing not what you wanted...

Model checking is a tool to verify the working of the system...
Chapter 6. Case studies, examples

1. Lift

The "elevator" (lift) example

Specify a lift!

• What are the components?
• How they are working?
• What is the invariant property?
• What are the variables, constants, invariant, operations?

2. SmallSet

The SmallSet:

• create a "SmallSet" abstract machine to create a set with limited size,
• operations:
  • enter,
  • remove.

SmallSet specification

```
MACHINE SmallSet
CONSTANTS maxsize
PROPERTIES maxsize : NAT1
VARIABLES numset
INVARIANT numset <: NAT1 & card(numset) <= maxsize
INITIALISATION numset := {}
OPERATIONS
  enter(nn) =
    PRE
      nn : NAT1 & card(numset) < maxsize
    THEN
      numset := numset \ {nn}
    END;
  remove(nn) =
    PRE
      nn : numset
    THEN
      numset := numset - {nn}
    END;
  nn <= minimum =
    PRE
```
The Ticket machine:

• create a "Ticket machine" abstract machine to create and serve tickets,

• operations:
  • serve_next,
  • take_ticket.

3. Ticket machine

```plaintext
MACHINE Ticket
VARIABLES serve, next
INVARIANT serve : NAT & next : NAT & serve <= next
INITIALISATION serve, next := 0, 0
OPERATIONS
ss <-> serve_next =
PRE serve < next
THEN ss, serve := serve + 1, serve + 1
END ;
tt <-> take_ticket =
PRE true
THEN tt, next := next, next + 1
END
END
```

4. Wallet

The Wallet:

• create a "Wallet" where one can store money,

• operations:
  • setBalance,
  • debit,
  • credit,
  • getBalance
• the balance is limited,

• the transactions are limited.

Wallet code:

• abstract machine of the "Wallet" + implementation

• abstract machine using the "Wallet" + implementation

### Wallet - specification (1)

```plaintext
MACHINE BWallet CONSTANTS MAX_BALANCE, MAX_TRANSACTION_AMOUNT, DEFAULT_BALANCE PROPERTIES MAX_BALANCE : NAT & MAX_BALANCE < 50000 & MAX_TRANSACTION_AMOUNT : NAT & DEFAULT_BALANCE : NAT & DEFAULT_BALANCE <= MAX_BALANCE CONCRETE_VARIABLES balance INITIALISATION balance := DEFAULT_BALANCE
```

### Wallet - specification (2)

```plaintext
OPERATIONS
setBalance (balanceInit) = PRE balanceInit : NAT & balanceInit : 0..MAX_BALANCE THEN balance := balanceInit END;

debit (debitAmount) = PRE debitAmount : NAT & (debitAmount >= 0) & (debitAmount <= MAX_TRANSACTION_AMOUNT) & (balance - debitAmount >= 0) THEN balance := balance - debitAmount END;
```

### Wallet - specification (3)
credit (creditAmount) =
PRE
  creditAmount : NAT &
  (creditAmount >= 0) &
  (creditAmount <= MAX_TRANSACTION_AMOUNT) &
  ((balance + creditAmount) <= MAX_BALANCE)
THEN
  balance := balance + creditAmount
END;
amount <= getBalance =
BEGIN
  amount := balance
END
END

Wallet - implementation (1)

IMPLEMENTATION  _BWallet_imp
REFINES    _BWallet
VALUES    MAX_BALANCE = 10000 ;
         MAX_TRANSACTION_AMOUNT = 100 ;
         DEFAULT_BALANCE = 0
INITIALISATION balance := 0
OPERATIONS
  setBalance (balanceInit) =
    BEGIN balance := balanceInit END ;
  debit (debitAmount) =
    BEGIN balance := balance - debitAmount END ;
  credit (creditAmount) =
    BEGIN balance := balance + creditAmount END ;
  amount <= getBalance =
    BEGIN amount := balance END
END

UsingWallet - specification (1)

MACHINE    UsingBWallet
OPERATIONS
  main = skip
END

UsingWallet - implementation (1)

IMPLEMENTATION  UsingBWallet_imp
REFINES    UsingBWallet
IMPORTS
UsingWallet - implementation (2)

5. Maximum search

MACHINE maxker
OPERATIONS
i,maxi <= op_maxker(m,n,f) -
PRE
m: 1..MAXINT - 1 & n: 1..MAXINT - 1 &
f: 0..100 --> 0..300 & m..n <= dom(f) &
m: dom(f) & n:dom(f) & m<=n
THEN
ANY k
WHERE
k:INTEGER & k:1..MAXINT - 1 &
!j.(j:m..n --> f(j)<=f(k) )
THEN
maxi:=f(k) || i:=k
END
END

Maxker implementation

IMPLEMENTATION maxker_imp
REFINES maxker
OPERATIONS
i,maxi <= op_maxker(m,n,f) -
BEGIN
VAR k IN i:=m; k:=m; maxi:=f(m);
WHILE k/=n DO
 IF f(k+1)>=maxi
   THEN i:=k+1; maxi:=f(k+1)
   ELSE skip
 END;
k:=k+1
INVARIANT
m<=n & k: m..n & i: m..k &
maxi=f(i) & !j.(j:m..k) => f(j)<=f(i) 
VARIANT
n-k
END /* WHILE */
END /* VAR */

Use Maxker specification

MACHINE
use_maxker
OPERATIONS
main = skip
END

Use Maxker implementation

IMPLEMENTATION use_maxker_imp
REFINES use_maxker
IMPORTS maxker , BT_IO
CONCRETE_VARIABLES g
INVARIANT
g: 0..100 --> 0..200
INITIALISATION
g := {x,y| x:0..100 & y: 0..200 & y=x}
OPERATIONS
main =
VAR j,res IN
 j,res <=-- op_maxker(2,5,g);
  writeInteger(j);
  writeInteger(res)
END
END

Maxker Java

import java.util.Arrays;
class maxker {
 static void INITIALISATION() {
 }
public static int op_maxker(int m, int n, int[] f, BInteger res_1) {
    int maxi = 0;
    int i = 0;
    int k;
    i = m;
    k = m;
    maxi = f[m - (0)];
    while (k != n) {
        if (f[k + 1 - (0)] >= maxi) {
            i = k + 1;
            maxi = f[k + 1 - (0)];
        } else {
            k = k + 1;
        }
    }
    res_1.setValue(maxi);
    return i;
}

import java.util.Arrays;
class use_maxker {
    static void INITIALISATION() {
        maxker.INITIALISATION();
        BT_IO.INITIALISATION();
    }
    public static void main(String[] Args[]) {
        INITIALISATION();
        int j;
        int res;
        BInteger res_0 = new BInteger();
        j = maxker.op_maxker(25, fv.u, res_0);
        res = res_0.getValue();
        BT_IO.writeInt(j);
        BT_IO.writeInt(res);
    }
}

6. Jukebox

The Jukebox:
- create a "Jukebox" music playing from a playlist for money machine,
- operations:
  - pay,
  - select,
  - play
  - pay to add music to playlist,
play music from the playlist.

Jukebox code:

- abstract machine of the "Jukebox",
- refinement,
- implementation.

### Jukebox 1

**MACHINE** Jukebox  
**SETS** TRACK  
**CONSTANTS** limit  
**PROPERTIES** limit : NAT1  
**VARIABLES** credit, playset  
**INVARIANT** credit : NAT &  
credit <= limit &  
playset <: TRACK  
**INITIALISATION** credit := 0 || playset := {}  

**OPERATIONS**  
pay(cc) =  
PRE cc : NAT1  
THEN credit := min({credit + cc, limit})  
END;

### Jukebox 2

**select(tt) =**  
PRE credit > 0 & tt : TRACK  
THEN  
CHOICE credit := credit - 1 ||  
playset := playset \ {tt}  
OR playset := playset \ {tt}  
END  
END;

### Jukebox 3

**tt <-- play =**  
PRE playset /= {}  
THEN  
ANY tr  
WHERE tr : playset  
THEN tt := tt || playset := playset - {tr}  
END  
END;
penalty =
  SELECT credit > 0 THEN credit := credit - 1
  WHEN playset /= {} THEN
    ANY pp
      WHERE pp / : playset
      THEN playset := playset - {pp}
    END
  ELSE skip
  END
END
END

Jukebox refinement 1

REFINEMENT JukeboxR
REFINES Jukebox
CONSTANTS freefreq
PROPERTIES freefreq : NAT
VARIABLES creditr, playlist, free
INVARIANT creditr : NAT &
  creditr = credit &
  playlist : iseq(TRACK) &
  ran(playlist) = playset &
  free : 0..freefreq

INITIALISATION creditr := 0 ;
  playlist := <> ;
  free := 0

Jukebox refinement 2

OPERATIONS
pay(cc) =
  PRE cc: NAT
  THEN creditr := min({creditr + cc, limit})
  END;
select(tt) =
  PRE tt: TRACK
  THEN
    IF tt / : ran(playlist)
      THEN playlist := playlist <= tt
      END;
    IF free = freefreq
      THEN CHOICE free := 0
          OR creditr := creditr - 1
      END
    ELSE free := free + 1 ; creditr := creditr - 1
    END
  END;
Jukebox refinement 3

\[
\text{tt} \leftarrow \text{play} = \\
\text{PRE} \; \text{playlist} /= <> \\
\quad \text{THEN} \\
\quad \quad \text{tt} := \text{first}(\text{playlist}); \\
\quad \quad \text{playlist} := \text{tail}(\text{playlist}) \\
\quad \text{END}; \\
\]

Jukebox refinement 4

\[
\text{penalty} = \\
\text{IF} \; \text{playlist} /= <> \\
\quad \text{THEN} \; \text{playlist} := \text{tail}(\text{playlist}) \\
\quad \quad \text{ELSEIF} \; \text{creditr} > 0 \\
\quad \quad \quad \text{THEN} \; \text{creditr} := \text{creditr} - 1 \\
\quad \text{END} \\
\quad \text{END} \\
\]

7. Traffic lights

Traffic regulation – Verified mobile components

- Controlling traffic lights – Dynamically download, link and execute code
- Road security – Ensure the correctness of mobile code
- B-method – Formal reasoning is preferred
- CPPCC – Minimal client-side / run-time overhead

Traffic lights at a cross
Conflict
Controller and sensors
Adapting to changes in traffic situation
Controlling traffic lights

- Controller contains mobile component
- Sensors provide input data
- Mobile component provides data to control lights
- Mobile component is proved to ensure road safety
- Controller downloads and verifies mobile component

Example
7.1. Choco vending machine

The Choco vending machine:

- create a "Choco vending machine"
- user can put coin 10 and/or coin 20 into the machine
- user can ask for small chocolate, the price is 10
- user can ask for big chocolate, the price is 20
- user can ask for return its money
Questions about the Choco vending machine:

- What are the “hardware” limitations of the machine?
- What are the constants and the variables?
- What should “express” the invariant?
• *What can be the invariant?*

• *What are the implications of the invariant?*

• Think about the "return money" operation...

### Choco vending machine

**MACHINE**

```plaintext
hivo
OPERATIONS
hivas= skip
END
```

### Choco vending machine

**IMPLEMENTATION**

```plaintext
hivo_i
REFINES
hivo
IMPORTS
csg,BASIC IO
OPERATIONS
hivas=
BEGIN
init; STRING WRITE("indul
");
szerviz; STRING WRITE("szerviz
");
bedob10; STRING WRITE("bedob10
");
kerkiscsoki; STRING WRITE("kerkiscsoki
");
bedob20; STRING WRITE("bedob20
");
kernagycsoki;STRING WRITE("kernagycsoki
");
bedob20; STRING WRITE("bedob20
");
kerkiscsoki; STRING WRITE("kerkiscsoki
");
visszaad; STRING WRITE("visszaad
");
END
```

### Choco vending machine

**MACHINE**

```plaintext
csg
CONSTANTS
maxkassza10,
maxkassza20,
maxkiscsoki,
maxnagycsoki
PROPERTIES
maxkassza10 : NAT & maxkassza10 = 10
& maxkassza20 : NAT & maxkassza20 = 5
& maxkiscsoki : NAT & maxkiscsoki = 5
& maxnagycsoki : NAT & maxnagycsoki = 5
VARIABLES
kiscsoki,
nagycsoki,
```
Choco vending machine

**INVARIANT**

- \( kassza10 : \text{NAT} \land kassza10 \leq \text{maxkassza10} \)
- \( kassza20 : \text{NAT} \land kassza20 \leq \text{maxkassza20} \)
- \( \text{kiscsoki : NAT} \land \text{kiscsoki} \leq \text{maxkiscsoki} \)
- \( \text{nagycsoki : NAT} \land \text{nagycsoki} \leq \text{maxnagycsoki} \)
- \( \text{bedobott : NAT} \)
- \( \text{bedobott} \leq \text{kassza10} \times 10 + \text{kassza20} \times 20 - \text{bedobott} \times 10 \)
- \( \text{(kassza10 \times 10 + (kassza20 \times 20) - bedobott)} \)

**INITIALISATION**

- \( \text{kiscsoki} := \text{maxkiscsoki} \)
- \( \text{nagycsoki} := \text{maxnagycsoki} \)
- \( \text{kassza10} := \text{maxkiscsoki} \)
- \( \text{kassza20} := 0 \)
- \( \text{bedobott} := 0 \)

**OPERATIONS**

**init=**

- **BEGIN**
  - \( \text{kiscsoki} := \text{maxkiscsoki} \land \text{nagycsoki} := \text{maxnagycsoki} \)
  - \( \text{kassza10} := \text{maxkiscsoki} \land \text{kassza20} := 0 \)
  - \( \text{bedobott} := 0 \)
  - **END**

**szerviz =**

- **PRE**
  - \( \text{bedobott} = 0 \)

**THEN**

- \( \text{kiscsoki} := \text{maxkiscsoki} \land \text{nagycsoki} := \text{maxnagycsoki} \)
- \( \text{kassza10} := \text{maxkiscsoki} \land \text{kassza20} := 0 \)
- **END**

**bedob10 =**

- **PRE**
  - \( \text{kassza10} < \text{maxkassza10} \)

**THEN**

- \( \text{bedobott} := \text{bedobott} + 10 \land \text{kassza10} := \text{kassza10} + 1 \)
- **END**

**bedob20 =**

- **PRE**
Case studies, examples

\begin{verbatim}
kassza20 < maxkassza20
THEN
  bedobott := bedobott + 20 || kassza20 := kassza20 + 1
END;
\end{verbatim}

**Choco vending machine**

\begin{verbatim}
kerkiscsoki =
PRE
  bedobott >= 10 & kiscsoki > 0
THEN
  bedobott := bedobott - 10 || kiscsoki := kiscsoki - 1
END;
\end{verbatim}

**Choco vending machine**

\begin{verbatim}
visszaad =
  ANY vissza10,vissza20
WHERE
  vissza10 : 0..kassza10
& vissza20 : 0..kassza20
& (vissza10 * 10) + (vissza20 * 20) = bedobott
THEN
  bedobott := 0
|| kassza10 := kassza10 - vissza10
|| kassza20 := kassza20 - vissza20
END
END
\end{verbatim}

**Choco vending machine implementation**

\begin{verbatim}
IMPLEMENTATION
csg_i
REFINES
csg
VALUES
  maxkassza10 = 10; maxkassza20 = 5;
  maxkiscsoki = 5; maxnagycsoki = 5
CONCRETE VARIABLES
  kiscsoki, nagycsoki, kassza10, kassza20, be10, be20
\end{verbatim}
Choco vending machine implementation

INVARIANT

\text{\texttt{kassza10 : NAT}} \& \text{\texttt{kassza10 <= maxkassza10}}
\& \text{\texttt{kassza20 : NAT}} \& \text{\texttt{kassza20 <= maxkassza20}}
\& \text{\texttt{kiscsoki : NAT}} \& \text{\texttt{kiscsoki <= maxkiscsoki}}
\& \text{\texttt{nagycsoki : NAT}} \& \text{\texttt{nagycsoki <= maxnagycsoki}}
\& \text{\texttt{be10 : NAT}} \& \text{\texttt{be20 : NAT}}
\& \text{\texttt{bedobott = 10 * be10 + 20 * be20}}
\& \text{\texttt{(maxkiscsoki - kiscsoki) * 10 + (maxnagycsoki - nagycsoki) * 20 + maxkiscsoki * 10}}
\& \text{\texttt{(kassza10 * 10)}}, \text{\texttt{(kassza20 * 20)}} - \text{\texttt{bedobott}}

Choco vending machine implementation

INITIALISATION

\text{\texttt{kiscsoki := maxkiscsoki; nagycsoki := maxnagycsoki;}}
\text{\texttt{kassza10 := maxkiscsoki; kassza20 := 0;}}
\text{\texttt{be10 := 0; be20 := 0}}

Choco vending machine implementation

OPERATIONS

\textit{init =}
\begin{verbatim}
BEGIN
  kiscsoki := maxkiscsoki;
  nagycsoki := maxnagycsoki;
  kassza10 := maxkiscsoki;
  kassza20 := 0;
  be10 := 0; be20 := 0
END

\textit{szerviz =}
\begin{verbatim}
BEGIN
  kiscsoki := maxkiscsoki;
  nagycsoki := maxnagycsoki;
  kassza10 := maxkiscsoki;
  kassza20 := 0
END
\end{verbatim}

\end{verbatim}
bedob10 =
BEGIN
  be10 := be10 + 1;
  kassza10 := kassza10 + 1
END;
bedob20 =
BEGIN
  be20 := be20 + 1;
  kassza20 := kassza20 + 1
END;

cherkiscsoki =
BEGIN
  IF be10 = 0 THEN
    be10 := 1; be20 := be20 - 1; kiscsoki := kiscsoki - 1
  ELSE
    be10 := be10 - 1; kiscsoki := kiscsoki - 1
  END
END;

kernagycsoki =
BEGIN
  IF be20 = 0 THEN
    be10 := be10 - 2
  ELSE
    be20 := be20 - 1
  END;
  nagycsoki := nagycsoki - 1
END;

visszaad =
BEGIN
  kassza10 := kassza10 - be10;
  kassza20 := kassza20 - be20;
  be10 := 0; be20 := 0
END
END

Choco vending machine implementation

Choco vending machine implementation
Chapter 7. Annexes

1. Recommended readings, references

<table>
<thead>
<tr>
<th>Recommended readings</th>
</tr>
</thead>
</table>
| • J-R Abrial, *The B-Book - Assigning Programs to Meanings*,
| • J. Wordsworth, *Software Engineering with B*,
| • S. Schneider, *The B Method: An Introduction*,
Annexes

Atelier B and B method documents

- Clearsy repository of Atelier B manuals, reports and papers on the B method
  http://www.tools.clearsy.com/resources/documents/
- Atelier B User Manual Version 4.0
- B Language User Manual Version 1.2 (in French)
  http://www.tools.clearsy.com/resources/B_manuel_utilisateur.pdf
- B Language Keywords and Operators Version 1.8.5

2. Usefull webpages

Usefull webpages

- http://vl.fmnet.info/b/
- http://www-_lsr.imag.fr/B/
- http://www.b-_core.com/ B-Core
- http://www.atelierb.societe.com/index_uk.htm AtelierB
- http://www.b4free.com/ B4free
- http://www.loria.fr/~cansell/cnp.html Click’n’Prove
• http://en.wikipedia.org

Other formal methods and B method related documents

• http://gergo.erd.hu/blog/2010--02--_16--_the_b_method_for_programmers_(part_1)/
• http://gergo.erd.hu/blog/2010--02--_22--_the_b_method_for_programmers_(part_2)/
• The B-Toolkit distributed by B-Core
• http://proglang.informatik.uni__freiburg.de/teaching/swt/2013/w04--_b--_method.pdf
• http://www.event--_b.org/
• http://wiki.event--_b.org/index.php/Main_Page
• Clearsy: ATELIER B, Interactive Prover User Manual, version 3.7,

Other recommended articles and books to read

• Event-B and the Rodin Platform
  http://www.event--_b.org/
• B tools and related documents
• Fóthi Ákos, Horváth Zoltán: Bevezetés a programozáshoz. ELTE Informatikai Kar, Budapest, 2005. digital coursebook, 510 pages,
Annexes


The end...

Please send your remarks, comments, corrections to:

Dr. Zoltán Istenes
Eötvös Loránd University
Faculty of Informatics
Department of Software Engineering and Methodology
istenes@inf.elte.hu

Thank you!