School of Computer Science and Engineering

June 2001
Fourth year thesis

Title: An exploration of implementation and code generation by the B-Toolkit

Student: Thai Son Hoang
Student ID: 2201905

Supervisor: Ken Robinson

Assessor: Albert Nymeyer
Contents

1 Introduction 2

2 Overview 3
  2.1 C code generation ............................................. 4
  2.2 C interface generation and linking .......................... 6
  2.3 B-Toolkit interface ............................................. 7
    2.3.1 Menus ...................................................... 7
    2.3.2 Environments ............................................. 8

3 JAVA code generation 10
  3.1 Implementing the JAVA system library ........................ 18
    3.1.1 Renamed library machines ................................. 19
    3.1.2 Type definition library machines ........................ 21
    3.1.3 Operation library machines ............................... 23
  3.2 Translating into JAVA code .................................. 24
    3.2.1 Creating the header ...................................... 30
    3.2.2 Creating constructor method .............................. 31
    3.2.3 Translating operations ................................... 32
    3.2.4 Translating substitutions ................................ 32
  3.3 Linking the JAVA code ........................................ 40
  3.4 Changing in B-Toolkit interface ............................... 43
    3.4.1 Changing in Translators/Compilers Option ............... 43
    3.4.2 Changing the translate button in Translator environment .... 45
    3.4.3 Changing the interface button in Generator environment ........ 47
    3.4.4 Changing the execute button in Translator environment ........ 47

4 Conclusion 48
  4.1 Limitation ..................................................... 48
  4.2 Suggestion ..................................................... 48

A B-Toolkit interface 50

B Student array example 63
  B.1 B implementation ............................................. 63
  B.2 C code ........................................................ 65
  B.3 JAVA code ..................................................... 67

C Manual on JAVA translation 69
1 Introduction

In the past few years, there has been significant improvement in the application of mathematics to software engineering. Comparing computing hardware and software engineering, mathematics was regarded as more closely connected to the former than the latter. With the introduction of software development notations like Z and VDM, software engineering step into a new era of an engineering approach to programming.

Invented by Jean-Raymond Abrial, the B-Method uses the Abstract Machine Notation (AMN). This is a formal method for developing software including a systematic way to progress from small to large software system. With the help of mathematics, the B-Method provides great power to software developers. The power includes producing safety-critical system where the correctness of the software is first priority.

B-Toolkit is built to illustrate all the aspects of B-Method. The toolkit covers the circle of software development from specification to documentation. During the development, B-Toolkit provides all the activities of software engineering: Animation, Proof obligation generation, correctness verification, code generation, and test application generation. By using AMN notation, B-Toolkit provides step by step development, with proof of consistency at each step. A large software development is divided into considerably smaller parts which are carefully checked using mathematical tools.

Implementation is one of the phases in developing software in B-Toolkit. The implementation consists of source code extracted from the library of pre-implemented reusable components, combined with the generated code from the B-Toolkit. Originally, the source code produced is C code, but theoretically, different programming languages can be generated. The object of the project is to change the B-Toolkit so that JAVA code is produced in place of C code. Other programming languages including PASCAL, ADA and VHDL have been generated. The available source code also included the PASCAL and VHDL versions of code generation.

Some useful notations:

- AMN: Abstract Machine Notation.
- SLIB: B-Toolkit System Library.
2 Overview

The structure of a software development using B-Toolkit is summarized in the diagram below:

![Diagram showing the development process]

Figure 1: B development process

From the user requirement, a formal specification is written using Abstract Machine Notation. In the specification, the behaviour of the software is described. It models what the software should do rather than how this should be achieved. At any time, the specification can be animated using the Animator tool of the Toolkit. The purpose of animation is to check the behaviour of the specification against the user requirement. On the other hand, formal proof obligations for the specification can be generated using Proof Obligation tools. In specification of B-Method, software is described as a state machine. With this model, the state of software is built from abstract, mathematical terms, such as values, sets and relations (including functions). The specification has a global invariant with operations acting as transitions in the state machine, to modify its state. The proof obligations are generated based on maintaining the correctness of the invariant after state transitions. The invariant must be correct at every states of the machine. With help from the Animator and Proof Obligation tool, a consistent specification is written. Since all the later steps of development depend on the specification, the consistency of specification is very important.
Because of its abstraction, the specification needs to go through several steps of refinement before the implementation. At each step of the refinement, the state machine model is used to represent a refinement machine. The new state in the refinement is written in a more concrete way compared with specification. The invariant of a refinement contains the transition relations between the state of the refinement machine and the state of the machine being refined. The Proof Obligation tools are based on this invariant. The correctness of the software preserves from specification through several steps of refinement by the relation between state machines. Steps by steps refinement reduce the complexity of software development. Using the Proof Obligation tool base on mathematics, the refinement can be proved to be correct (consistent with the previous level). Therefore, the input of the implementation phase is a concrete, proof to be correct refinement machine.

At implementation, the state of the machine is formed by other machines. The invariant of implementation machine is predicates on the state of the imported machines related to refinement machine. By using Proof Obligation tools, the implementation can be proved to be correct. Since the imported machines are proved correct, the software produced from the implementation will also be correct. This means the implementation will be consistent with the initial specification.

The code and document generations is done automatically by the B-Toolkit after having the fully developed implementation. Standard programming language for code generation is C. The work of changing the target programming language including investigating how the code is generated in the current version of the Toolkit. From the understanding about the current system, a clearly defined process for JAVA code generation can be specified.

2.1 C code generation

The diagram for C translating process as follow:
In pre-translating phase, the **Toolkit** checks all the conditions for translating the code. The checking list includes if all seen and imported machines are implemented, checking if there are duplicated imported machine. The translated programming language is checked at this stage in order to have correct translation process.

In the translating phase, the **Toolkit** invoke a program and base on the theory about the C code to translate from AMN into C. It is an internal translating process that depends on the knowledge about C language and using special tools in **B-Toolkit**. The C code is generated using the rules that working backward. All the rules for translating C code is defined in the TransC and TranslatorsC scripts in SRC directory of the **B-Toolkit** source code. A supporting script **Utils** and **Toolkit** are also used in the process. As an example, a rule is defined as below:

```
SetWriteToXTerm &
LoadToolkit(?)
=>
run_Bplf1(Prog_Language_Flag,C);
```

By working backward, the **Toolkit** sees which rules match the current situation of the translating process. The goal of the rule is matched with the status of the process. The
new status of the process comprise all the components of the hypotheses. In the example above, if the program language is C, the Toolkit is set to write to X-term and load toolkit number 7. The translating process is really a mathematical based process. It is matched with the idea of consistency in the B-Method.

After generating C code using the internal theory based translator, the code produced needs to be wrapped up. The reason for that is the limitation in the internal translator. It is done by calling an external script named TranCFile. The code produced after this step is compile-able C code.

The final step in C translating is producing a document for the generated code. The format of the document is chosen to be HTML. An external script named CreHtmlC is called to make the HTML document from the code generated in the earlier phase. The resulting documents are places in the HTX folder of the development.

The C code generation is based on the combination of the internal theory based translator and external scripts to produce a complete compile-able code and document for maintainability of the produced code.

2.2 C interface generation and linking

After translating, the produced C code need to be linked together into a top level interface. The diagram for C linking process as follow:
In the pre-linking phase, the **Toolkit** checks all the conditions for linking the C code. The checking list includes if all seen and imported machines are implemented, if there are any duplicate imported machine. The programming language flag is checked at this stage.

In generating the interface, again theories are used to create a main interface for the development. The syntax of the theory is the same as in translating.

Finally, the external script `LinkCFile` is called to link the interface with the code generated by the translator.

### 2.3 B-Toolkit interface

To accommodate new JAVA code generator, the interface of the **B-Toolkit** also need to be modified. The interface of the **B-Toolkit** included some menus and the different environments:

#### 2.3.1 Menus

- **Utils**: In the utils menu, **B-Toolkit** has different functions for changing the directory, saving the development in remarkable form, and exiting the **toolkit**.
• *Introduce:* From the options chosen from this menu, a new specification, refinement or implementation can be introduced to the development. Also B-Toolkit can introduce the machine from SRC directory or from standard library (SLIB).

• *Construct:* The options from the Construct menu are used to manipulate the constructs (machine, refinement, implementation, etc.) including remove construct from development, rename the construct, etc.

• *Remake:* The Toolkit has tools to remake all the constructs or some chosen constructs via menu Remake.

• *Browse:* The developer can browse through the hypertext of the development, or review the design, specification or even the selected construct. The syntax for AMN is found in the Palette option from this menu.

• *Options:* In the Options menu, the Toolkit provide tools for software developer customize the system. The available options available for customizing are: Remake, Construct display, Editor/Viewer/Shell, Documents option, Translators/Compilers, Interfaces, etc.

• *Help:* The help menu provides all helpful information about B-Method, B-Toolkit and system library (SLIB) in B-Toolkit.

Pictures of the Toolkit’s menus are in Appendix A

### 2.3.2 Environments

Other aspects of the B-Toolkit interface are environments. B-Toolkit has five different environments.

• *Main:* The Main environment contains almost every construct within the development. A construct can be committed after editing, analysed from this environment. The proof obligations for a construct can be generated by \textit{peg} button. The behaviour of the specification can be animated by \textit{ann} button.

• *Provers:* The provers environment controls all the proving of the proof obligations. After generating, the proof obligations can be mathematically proved and pretty printed in the Provers environment.

• *Generators:* In the generators environment, the interface of a development can be generated. Also, the database is generated from this environment.

• *Translators:* After being fully implemented, an implementation in B-Toolkit is translated into executable code in the Translators environment. By using the \textit{exe} button, the executable code may be run for actual testing.
- **Document**: An important environment in **B-Toolkit** is document. All the documents about specification, refinement, implementation, proof are kept in here. The documents can be marked up using Latex, and can be viewed by external viewer or printed.

Pictures of the environments of **B-Toolkit** interface are in **Appendix A**.
3 JAVA code generation

Consider writing a new JAVA code generation for B-Toolkit, two possible solutions are available. The first solution is to write an internal translator based on the theories similar to the C code generator. The second solution is to write an external translator and plug it into the Toolkit. The advantage of the former is consistency. Since internally, the theories covers all the grammar of AMN so writing JAVA generator based on these theories is ideal. The theories are placed in the source files in the folders SRC of the B source code. During the compilation of the source code, the .src files are compiled into binary file name Toolkit\n where n is a number. Each toolkit deals with different process of the B-Toolkit. For example, Toolkit 1 is for translation in general, Toolkit 7 is for C code translation. Toolkit 35 is for PASCAL code translation. Other processes using the theories format included Animator, Proof Printer, Base Generator. The format of the theories is the same as the format of the theories when running proof obligations. And similar to proof obligation, the B-Toolkit searches through the list of the theories and finds the rules that match the current state of the process. At the lowest level of the rule, the Toolkit uses bprintf to write to the output file

\[\text{bprintf(}}^\text{"\ void INL\%()\; }^\text{",m)}\]
\[\Rightarrow\]
\[ti-h(m)(h);\]

For example, the simple rule above is used to write the header of the function INL\_name where name is the name of the development. One rule can call other rules, for example:

\[\text{ReadTrAbs(m,b)} \& \text{WriteDot} \& \text{StoreAllPropertiesConstants(k,imp)} \& \text{StoreVisSetConstNames(m,b)} \& \text{TranslateImp(m(p),beatl(m,"\#"),beatl(m,"\#"),beatl(m,"\#"))}\]
\[\Rightarrow\]
\[\text{TransC\_parse(k(p))};\]

In the above rule, if the current status of the translation process matches with the state TransC\_parse(k(p));, the actions in the antecedent become the new goals for the translation process.

By working backward through the set of rules like this, the Toolkit is capable of translating from B implementation into C code.

There is a problem adapting the approach due to a lack of documentation of the theories, how the rules are constructed, etc. The format of the theories included the internal B-Toolkit rules such as brule, becall, and how to construct the other rules from these basic rules. Without documentation, the problem of understanding the theories become difficult. The whole translation process might be corrupted if one of the rules is not correct. This
proved to be a great obstacle, and could not be solved in the very limited time of the project. An external JAVA translator is used to solve the problem. The advantage of the external JAVA translator is in documentation, JAVA already has the tool to do that using javadocs command. Similarly for the linker, external JAVA linker is a possible solution for that.

Consider the changing in the interface. Changing the translated programming language only effects the implementation phase, so there are few changes in the interface related to that.

- In Options menu, the Translators/Compilers option needs to be changed to accommodate new programming language JAVA.

- The external JAVA translator needs to plug in the Translator environment. Associated with trl button, the translator is called to translate the B implementation into JAVA.

- The external JAVA linker needs to be plug into the Generator environment. When the JAVA language flag is selected in Translators/Compilers option, the JAVA linker is associated with itf button. Pressing the button generates a JAVA frame interface.

- In the Translator environment, the exe button needs to be changed to execute the JAVA codes instead of C code when the language option is JAVA. The JAVA frame interface is externally run when the button is pressed.

The conversion of the process of implementing and generating code by the B-Toolkit, from the implementation to the external JAVA code generator can be divided into three phases:

- Since the implementation of B machine is based entirely on the pre-implemented libraries, the system library (SLIB) needs to be rewritten in JAVA.

- Translating the implementation machine into JAVA code. The generated code needs to be documented using javadoc command. The generated code imports the pre-written JAVA library.

- Linking the JAVA code together by creating a top-level interface for the machine. The interface is written using JAVA frame class and compiled using the javac command externally. The generated classes can be executed using the java command.

The process for external JAVA code generation can be seen in the diagram below:
Consider simple example of the **B-Toolkit** implementation of an array of Student

```plaintext
IMPLEMENTATION Student_array_imp
SEES
  Bool_TYPE
REFINES
  Student_array
IMPORTS
  Student.Varr(STUDENT, max_size)
INVARIANT
  array = Student.Varr
INITIALISATION
  skip
OPERATIONS
  resp, student < -- Student.Read(index) =
  BEGIN
    IF (index <= max_size) & (index >= 1)
      THEN
        resp:=ok;
```

Figure 4: Writing external JAVA code generation
\begin{verbatim}
student < Student_VAL.ARR(index)
ELSE
  resp:=out_of_bound;
  student < Student_VAL.ARR(max_size)
END
END;

resp < Student_Store(index, student) =
BEGIN
  IF (index <= max_size) & (index >= 1)
  THEN
    resp := ok;
    Student_STO.ARR(index, student)
  ELSE
    resp := out_of_bound
  END
END
END;
....
\end{verbatim}

Above is part of the Student_array development (See Appendix B). In the example, the B implementation imports a library machine to store an array of student. The Student_Var library has two input parameters, the set of STUDENT and the maximum size of the array. The Student_Var library actually is an instance of Rename_Var library. By replacing the Rename prefix with different strings, different instances of this library are created. The INARIANT is the relation between a variable in the previous level of development (refinement) and a variable in the library machine. That relation is established and used for proving the consistency of the implementation. In the OPERATIONS part of the implementation, the operations in the library are used to change the variable inside the library. For example, to store an student at position index, the operation Student_STO.ARR is used. The name of the operation also follows the Rename convention. In the Rename_Var machine, the name of the operations are Rename_STO.ARR, Rename_VAL.ARR, etc, therefore in the machine Student_Var, the name of the operations are Student_STO.ARR, Student_VAL.ARR accordingly.

For C code translation, the importing of a library machine is done by generating the actual C code for that library by replacing all the occurrences of Rename with Student. Student_Var.c and Student_Var.h are the code represented the Student_Var instance. The header file of the C library is included in the code generated for the implementation. Below is the C code generated by the B-Toolkit for above implementation.

\texttt{#include "Student_array.h"}  
\texttt{#include "Student_Var.h"}
# include "Bool_TYPE.h"

void
INLStudent_array()
{
    INLStudent_Varr();
    ;
}

void
Student_Read(_, resp , _student, _index)
int *resp, *student, _index;
{
    if ( _index <= Student_arrayP1 & & _index >= 1 ) {
        *resp = ok;
        Student_VAL_ARR(_,student, _index);
    }
    else {
        *resp = out_of_bound;
        Student_VAL_ARR(_, Student_arrayP1);
    }
}

void
Student_Store(_, resp, _index , _student)
int *resp, _index, _student;
{
    if ( _index <= Student_arrayP1 & & _index >= 1 ) {
        *resp = ok;
        Student_STO_ARR(_,index, _student);
    }
    else {
        *resp = out_of_bound;
    }
}

The Bool_TYPE machine is seen in the B implementation. In the above C code, the
header of the library Bool_TYPE.h is included. Notice here that, the operations in the B
implementation have equivalent functions in the C code. The operation Student_STO_ARR
has the function equivalent to it named Student_STO_ARR in the library instance Student_Varr. By renaming, the C code generator produces actual source code for the library.
For example, if another Varr library is imported named Teacher_Varr, the code generator
produces two copies of the Rename_Varr by changing Rename to Student and Teacher
accordingly. These two copies are really the same but the operation names are different according to the prefixes _Student_ and _Teacher_. The variables inside the library also follow the rule. In the C code, the variables is also renamed but in the JAVA code, the principle of object oriented omits the renaming of the variables.

The advantage of JAVA over C when going into the translation process is that the AMN notation included the object-oriented model into it. The idea is that each machine can be represented as a JAVA class. An instance of that machine (imported in other implementation) is one object of that class. The operations in the machine are regarded as methods and all the variables are fields of the JAVA class. The constructor in JAVA is the _INITIALISATION_ in B implementation. The syntax of AMN is already similar to an object-oriented language translation. Below is possible JAVA code for the simple implementation above.

```java
import _Bool_TYPE;
import _RESPONSE;
import Varr;

public class _Student_array {
    private Varr _Student_Var;
    private int _max_size;

    public _Student_array (int _max_size) {
        this._max_size = _max_size;
        _Student_Var = new Varr("Student", "Student_array", _max_size);
    }

    public void _Student_Read(int [] _resp, int [] _student, int _index) {
        if (_index <= _max_size && _index >= 1) {
            _resp[0] = _RESPONSE_ok;
            _Student_Var.VAL_ARR(_student, _index);
        } else {
            _resp[0] = _RESPONSE_out_of_bound;
            _Student_Var.VAL_ARR(_student, _max_size);
        }
    }

    public void _Student_Store(int [] _resp, int _index, int _student) {
```
if ( _index <= _max_size && _index >= 1 ) {
    _resp[0] = _RESPONSE_ok;
    _Student_Varr.STO.ARR( _index , _student );
} else {
    _resp[0] = _RESPONSE_out_of_bound;
}
}

The full B implementation, C code and JAVA code for Student_array example are in Appendix B.

Similar to the C code, the library __BoolTYPE is imported as for SEES clause in implementation. But the code equivalent to the IMPORTS clause is different. The library Varr is implemented as a class. And the instance Student_Varr is created as an object of Varr class. The operation Student_STO_ARR is created as a method called by the object _Student_Varr. Therefore if more than one instance of the library is required, they can be created as objects in JAVA. The overhead of creating a new copy of the library as in C code translation is eliminated. The system library (SLIB) therefore can be created as a set of classes in JAVA with the operations corresponding to the specification of the B library.

One of the differences between the B implementation and a programming language is the variable declaration. A variable in JAVA is declared as belonging to a type such as integer or float. In B, a variable is declare as belonging to a set. The JAVA translator needs to handle the set in B as a type definition. Fortunately, all sets in B are assumed to be finite. There is a mapping one to one from the set in B to a natural number. Before doing the translation, the type checking is already done carefully by the analyser. To reduce the complexity of the translation process, in library and translation phase, all variables can be regarded as type integer. The type checking is validated at the interface level. To support that idea, all the sets in B are modelled as a class in JAVA with two basic static methods. The first method valid is used to convert from String to integer and the second method toString is used to convert from integer to String for displaying. Virtually, all the set classes have methods to establish a mapping one to one to integer. Since all the type checking is assumed to be correct, integer can be used in place of every set in the library and translation phases.

The same as for C, each function or method in JAVA has at most one return value. But in the B implementation, one operation can have more than one return value. The C code generator already has a neat way to handle that. All the functions in C are void (return nothing). The output as in B is passed as pointer in the input of the function. Similarly for JAVA, all the methods are written as returning void. The output as in B is returned to array variable parameters. Since all the variables are integer at this stage, the
format of the translation method corresponding to an B operation is as follows:

\[ \text{B-Operation} \]
\[ \text{output1, output2, ... outputN <--- Operation(input1, input2, ... inputN)} \]

\[ \text{JAVA} \]
\[ \text{void Operation(int [] output1, int [] output2, ..., int [] outputN, int input1, int input2, ... inputN)} \]

As in C code, the names of the variables should have some conventions to avoid collision with the keywords. By putting underscore before the names of the variables, the names of the variables are never the same as the keywords. Similarly for JAVA, every set, variable, constant and operation name has an prefix underscore. Further more, the name of the development also has an underscore prefixed to the name to distinguish from a JAVA library. For example, the name of the development is StudentArray, the name of the class associated with it is _StudentArray.

Following the above standard, an external JAVA code generation can be written and the B-Toolkit interface can be modified to include the new code generation.
3.1 Implementing the JAVA system library

The SLIB is the core part of the code generation. All the implementation are entirely based on these libraries. The implementation of these libraries in JAVA are different depending on the behaviour of the libraries themselves. For the system library in B, there are three different types of library machines.

The first is the renamed library machine. B implementation uses renamed library machines as one way to get an instance of a library machine. For example, the Rename_Varr library machine. In a B implementation, one can import First_Varr and Second_Varr to have two instances of Rename_Varr representing two arrays. Clearly, it suggests the relationship between class and objects. In JAVA, Rename_Varr library can be modelled as a class named Varr, with two imported machines First and Second being two objects of that class.

The second type of library machine is the Type definition machine such as Bool_TYPE, String_TYPE, etc. These machines provide access to a type (set) like BOOL, STRING accordingly.

The last type of library machine is the Operation machine. These machines provide only operation to manipulate the inputs. These libraries include Bool_TYPE_Ops, String_TYPE_Ops, file_dump, etc.

There are some machines in SLIB that provide I/O facilities. They are imported to implement the interface for the C code generation. In JAVA the interface is done by using the Frame class and the I/O is implemented by the standard JAVA I/O. Therefore in the JAVA version of the library, the following machines are not implemented: basic_io, Rename_Client, Rename_Server, Rename_token_io, Rename_File_io.

Since the implementation in B can use both the system library or implemented user development as the library, for consistency, the names of the library machines also follow the naming convention. The names of the type libraries (Bool_TYPE, String_TYPE) are prefixed with one underscore.

A similar naming convention is adopted for object names:

_Name_Type

For example, if the imported machine is Student_Varr; the name of the object associated with it is _Student_Varr.

For the operation machines, the operations are called according to the name of the machine. The operation CNJ_BOOL of Bool_TYPE_Ops is called as:

_Bool_TYPE_Ops.CNJ_BOOL

The reason is that it is treated in the same way as imported machine, with the name of the machine and an empty instance name. Hence the name of the machine starts with two underscores. For example, the operation library is Bool_TYPE_Ops, the name of the JAVA class should be _Bool_TYPE_Ops.
3.1.1 Renamed library machines

The renamed libraries in B are implemented as classes in JAVA with the INITIALISATION clause modeling the constructor method. Since the name of the renamed instance of the library is needed in some libraries, the name of the object and the name of the machine owner of this object are passed as arguments to the constructor of the class. Two private fields are created to keep the value of these name:

private String name;
private String parent;

With the input of the machine, there are two types. If the input of the machine is a set, it is ignored. This is because every set is regarded as integer in the library. If the input is not a set, it is passed as integer argument to the constructor method. For example, the Rename_Varr machine has two inputs: the set VALUE stored in the array and maxidx, the maximum index of the array:

MACHINE Rename_Varr(VALUE,maxidx)

The set VALUE is ignored. Therefore, the format of the constructor for Rename_Varr as follow:

public Varr(String name, String parent, int maxidx)

The non-set input for the machine is needed to store in the private fields of the class for further uses. In the Varr class, define private field

private int maxidx;

The values of these variables are assigned in the constructor method from the parameters including the name of the object and parent of the object:

this.maxidx = maxidx;
this.name = name;
this.parent = parent;

In the constructor method, all private variables need to be initialised to match the INITIALISATION clause in B implementation. In Varr class, the private array of integer Varr[] is used to modelled the state of the library instance. In the INITIALISATION clause:

INITIALISATION
Rename_Varr:: 1..maxidx --> VALUE
The array is assigned to any array of VALUE. Therefore in the constructor of JAVA class, a new array of integer is created

\[ Varr = \text{new int[maxidx];} \]

In summary, the Varr library machine can be implemented as follows:

```java
public class Varr {
    private int // Varr;
    private int maxidx;
    private String name;
    private String parent;

    public Varr(String name, String parent, int maxidx) {
        Varr = \text{new int[maxidx];}
        this.maxidx = maxidx;
        this.name = name;
        this.parent = parent;
    }

    \textbf{Other operations are here}
}
```

For library operations, the JAVA library needs to modify its fields to match with the implementation in B. Consider the operation \texttt{Rename\_STO\_ARR(\textit{ii}, \textit{vv})} in \texttt{Rename\_Varr} library. In B the behaviour of the operation is described as below:

\texttt{Rename\_STO\_ARR(\textit{ii}, \textit{vv}) =}
\texttt{PRE}
\texttt{\textit{vv}: VALUE &}
\texttt{\textit{ii}: 1..maxidx}
\texttt{THEN}
\texttt{Rename\_Varr(\textit{ii}) := \textit{vv}}
\texttt{END;}

The operation store an value in a specified location in the array. In JAVA, we can implement the operation as below, according to the private fields within the class.

```java
public void STO\_ARR(int \textit{ii}, int \textit{vv}) {
    Varr[\textit{ii}-1] = \textit{vv};
}
```
return;
}

Again the ignorance of type checking is applied in the library. Even the parameter \( vv \), an element of \( VALUE \), is regarded as belonging to type \( int \). If the operations has output like:

\[
bb <--- \text{Rename}_EQL\_ARR(\overline{ii},vv) = \\
\text{PRE} \quad vv\_VALUE & \quad ii:1\_maxidx \\
\text{THEN} \quad bb:=\text{bool}(\text{Rename}_Varr(ii)=vv) \\
\text{END};
\]

In JAVA the equivalent methods is:

```java
public void EQL_ARR(int[] bb, int ii, int vv) {
    if (Varr[ii-1] == vv) bb[0] = _BOOL_TRUE;
    else bb[0] = _BOOL_FALSE;
    return;
}
```

The value of the output is either \( _{\text{TRUE}} \) or \( _{\text{FALSE}} \) value of the \( _{\text{BOOL}} \) class.

Similarly, all other \( \text{Rename} \) library machines can be implemented in a similar way.

### 3.1.2 Type definition library machines

Type definition library machines only grant access to a set. There are five type machines in \( \text{SLIB} \): \( _{\text{Bool}}\_\text{TYPE}, \text{String} \_\text{TYPE}, \text{Scalar} \_\text{TYPE}, \text{Int} \_\text{TYPE}, \text{Bit} \_\text{TYPE} \) which provided access to sets \( _{\text{BOOL}}, \text{STRING}, \text{SCALAR}, \text{INT} \) and \( _{\text{BIT}} \) accordingly. Since set already is a class in \( \text{JAVA} \), the type library can import that set class. Consider \( _{\text{Bool}}\_\text{TYPE} \) machine, it owns the set \( _{\text{BOOL}} \). By convention, the name of the class containing \( _{\text{BOOL}} \) set is \( _{\text{BOOL}} \). So \( _{\text{Bool}}\_\text{TYPE} \) can be implemented as follow:

```java
import _{BOOL};
public class _{Bool}_TYPE {
}
```

Notice that, also by convention, the name of the type library start with two underscores.

As discussed above, the set class needs to have two static methods to convert from string to integer and vice versa to establish the mapping between them and the set of integer number. Take \( _{\text{BOOL}} \) as an example. Set \( _{\text{BOOL}} \) has two elements \( _{\text{TRUE}} \) and
**FALSE.** It is define as final static constant in class _BOOL:

```java
public static final int _TRUE = 1;
public static final int _FALSE = 2;
```

The method to convert string to integer is named **valid.** It throws an `IOException` in case of invalid input. In the case of the set _BOOL_, the method is implemented as below:

```
public static int valid(String string) throws IOException {
    if (string.compareTo("TRUE") == 0) return 1;
    if (string.compareTo("FALSE") == 0) return 2;
    throw (new IOException());
}
```

Vice versa, the method to convert from integer to string is named **toString.** Given an integer, it returns the string represent the element of the set associated with that integer. For _BOOL_ class, the method is implemented as follow.

```
public static String toString(int val) {
    switch (val) {
        case 1: return "TRUE";
        case 2: return "FALSE";
        default: return ";";
    }
}
```

From the design of _BOOL_ class, we can implement all enumerated sets in B by knowing the mapping from the set to integer. Similar to _BOOL_ class are _SCALAR, _INT_ and _BIT_ classes.

The only different class is **STRING.** Since the mapping from strings to integers is not obvious. Given a string, the translator needs to find a mapping from this string to an integer which is unique. The solution for that is keeping a global file of all used strings. When a new constant string appears, first, it is checked with the global file, if the string exists, the number associated with it is returned. If not, a new number is generated and the string is added to the global file. Vice versa, when given a number, a string associated with it is found by searching through the global string file. This global string also applies to all strings appearing during the translation process for consistancy. Using the global string file is an advance compares to the C code where String is represent as an array of integer. The global string file supports any string of any size.
3.1.3 Operation library machines

Operation machines by convention have names starting with two underscores. Since the operations in this type of library only manipulate the inputs, the JAVA class do not have a constructor method. All the methods and fields in the class are static. For example, the Bool-TYPE-Ops machine is used to manipulate BOOL inputs. Consider the operation CNJ_BOOL to return the conjunction of two BOOL variables as follow:

\[ bb <--- \text{CNJ_BOOL}(cc, dd) = \]

PRE
\[
\begin{align*}
cc & : \text{BOOL} & & \text{&} \\
dd & : \text{BOOL} \\
\end{align*}
\]

THEN
\[
bb := \text{bool}(cc=\text{TRUE} \text{&} dd=\text{TRUE})
\]

END;

In JAVA class, we have:

```java
public static void CNJ_BOOL(int [] bb, int cc, int dd) {
    if ((cc == _BOOL_TRUE) && (dd == _BOOL_TRUE))
        bb[0] = _BOOL_TRUE;
    else bb[0] = _BOOL_FALSE;
    return;
}
```

The _Bool-TYPE-Ops class imports _BOOL. The operation is static since it is only used for manipulating the inputs. Similarly the case for other operations machine.
3.2 Translating into JAVA code

The JAVA translator needs to understand the grammar of B implementation. Fortunately, after the analysing process, B-Toolkit produces a fixed format analysed file of the implementation in the /ANL folder of the development. The translator can get the information needed from the well-formatted analysed file. The reason for looking at the analysed file is that the DEFINITIONS clause in B implementation is already parsed. Therefore the translator does not need to parse the implementation again. Also the format of the analysed file is restricted, much more easy to parse rather than the free format of the implementation.

During the translation of the implementation, some global variables are used to store the information needed.

- **Imported list**: List of imported classes. These classes are added to the list during the translation.

- **Operation list**: This is list of imported instances and the operations belonging to those instances. The instance in the list has name, type, and two lists. The first is a list of operations and the second a is list of markup operation. They are used to translate the operation in B implementation into JAVA by applying the following naming rule.
  
  `_Name_Type.Operation_`

  For example if one in B implementation is `Student.VAL.ARR` in JAVA, the operation become `_Student.Varr.VAL.ARR` since the operation belongs to the instance named `Student` and has type `Varr`. By searching through the list, the translator can get the markup for every operation appearing in the implementation.

- **Constant list**: List of all the machine with the constants associated with it.

- **Set list**: List of all the machines with the enumerated sets associated with it.

- **Imported machine list**: List of the library and user machines that are imported by the implementation, along with their parameters.

The analysed file has different parts separated by ";;;":

*Part 1; Part 2; ....*

Only parts related to the translation process are discussed in this report.

- The second part of the analysed file is associated with the SEES clause. All the seen machines are listed in here. The format in the analysed file is:
  
  `((seen_Machine1), (seen_Machine2); ....)`

  The `seen_Machine` contains the name of the machine according to the format:

  `Name(?,?,(?,?,?,?)`  

  The JAVA translator extracts the name of the seen machine. According to the name of the machine, the translator can know the type of the library. The possible
seen machines are *Type* library machines, *Operation* library machines, user created machines and context machines. The context machines are machines that provide access to a set of tokens. These machines are used to support the multiple object machines. These machine included *Rename_set_obj*, *Rename_str_obj*, *Rename_seq_obj*, 
*Rename_func_obj* and *Rename_func_obj*. For example, if the implementation imports the *Rename_set_obj* library, it needs to see the *Rename_set_ctx* to get access to the tokens to indentify different sets. Since all sets at the library level are integer, the set of tokens is also integer. Therefore in the multiple object library machine, integer can be used as tokens for accessing the objects. Implementation of these context machines becomes redundant. The translator can ignore the seeing of the context machine. For other types of the seen machine, the translator can do the following steps.

- If the seen machine is a *Type* library machine, the machine is added to the list of imported classes if it is not in there. For *Bool_TYPE* machine, the set *BOOL* containing *TRUE* and *FALSE* is added to the list of sets.

- If the seen machine is a *Operation* library machine, The machine is added to the list of imported classes if it is not in there. The operations contained in that operation library machine are added to the list of operation along with the name of the instance that contain it as empty and the type of the class is the name of the library. The markup for each operation is also added to the list. The markup for an operation is the same as the name of the operation. Consider operation from *Bool_TYPE_Ops*, *CNL_BOOL*, in B implementation, it is written as:

\[
  \text{bb} \leftarrow \text{CNL_BOOL}(cc, dd);
\]

In **JAVA**, it becomes:

\[
  \text{bb} = \_\text{Bool_TYPE_Ops}_\text{CNL_BOOL}(cc, dd);
\]

The list of operations in operation library is stored in file in the /BLIB/LIB JAVA/ folder of the B-Toolkit directory. By storing the format of the library in the file, in the future, the library can be extended, and the new **JAVA** library format can be added without changing the translation code.

- If the seen machine is a user machine, the enumerated sets and constants in that machine are found and added to the list of sets and constants. This information is found by parsing through the analysed file associated with the user machine.

- The fifth part of the analysed file is associated with the *SETS* clause. All the sets belonging to the development are listed in this part. The format in an analysed file is:
There are two types of sets: enumerated set and non-enumerated set. For non-enumerated set, the name of the set is listed. For enumerated sets, the name and the members of the set are listed according to the following format:

\[ \text{Name} \text{ele}1, \text{ele}2, \ldots \]

where \text{ele}1, \text{ele}2, \ldots are the elements of the set. The sets are added to the list of sets along with their members. The members are empty if the set is not enumerated. The enumerated sets are added to the list of imported classes if they are not in the list. The class associated with this enumerated set is created. For example, if in the development, the set RESPONSE is defined as follows:

\[ \text{RESPONSE} = \{ \text{ok, out\_of\_bound, already\_full} \} \]

A class \_RESPONSE is added to the list of imported and is created (the underscore follow the naming convention):

```java
import java.io.*;

public class _RESPONSE {
    public static final int _ok = 1;
    public static final int _out_of_bound = 2;
    public static final int _already_full = 3;

    public static int valid(String string) throws IOException {
        if (string.compareTo("ok") == 0) return _ok;
        if (string.compareTo("out\_of\_bound") == 0) return _out_of_bound;
        if (string.compareTo("already\_full") == 0) return _already_full;
        throw (new IOException());
    }

    public static String toString(int value) {
        switch (value) {
            case 1 : return "ok";
            case 2 : return "out\_of\_bound";
            case 3 : return "already\_full";
            default: return "";
        }
    }
}
```

26
The set class is created according to the standard discussed earlier. There are two methods to convert between each element of the class and integer and vice versa. If one variable is assigned to one element of the set \( RESPONSE \), for example, in \( B \) implementation, the following substitution is used to assign a value for output:

\[
res : = \text{out}_\text{of}_\text{bound};
\]

in \texttt{JAVA} the substitution become:

\[
res = \_RESPONSE\_out\_of\_bound;
\]

By importing the \_RESPONSE class, the \texttt{JAVA} translation of \( B \) implementation can access the set \( RESPONSE \).

- The ninth part of the analysed file is associated with the \texttt{PROPERTIES} clause. The properties of all constants and sets appearing in the development are listed here. The properties are listed as a normal predicate, contained the information about the constants and set:

\[
(Predicate1 \& Predicate2 \& ...)
\]

Each predicate is a constrain about the type of a constant, the value of a constant or the value of a set.

The translation ignores the predicate about type of the variable since every set is integer. The value of constant or set is recorded by the translation.

- If the predicate is constrained to the value of a constant having the form:

\[
constant = value
\]

the name and the value of the constant are add to global list of constant.

- If the predicate is constraint about the value of a set, there are three case:

* If it is equal to other set: \( Set = AnotherSet \), the set is added to global list of imported classes if it is not there. A new class for the \( Set \) is created as follow:

\[
\text{import AnotherSet;}
\]

\[
\text{public class Set} \\
\quad \{ \\
\quad \text{public static int valid(String string) throws IOException} \\
\quad \}
\]
```java
int value;
try {
    value = AnotherSet.valid(string);
} catch (IOException e) {
    throw (new IOException);
}
return value;

public static String toString(int value) {
    return AnotherSet.toString(value);
}
```

The new Set created is the same as AnotherSet by using the methods valid and toString of that AnotherSet.

* If the set is defined as a subrange of natural number, such as:
  
  \[ Set = 1..1000 \]

  The Set is added to the global list of imported classes and a new class is created. Since the members of the class are integer, so the toString method of the class just returns the string representing the number. The valid method need to check if the string entered represents a number within the range or not. It is also straightforward.

* If the set is an enumerated set, such as:
  
  \[ Set = \{1, 2, 3, 4\} \]

  The Set is added to the global list of imported classes and a new class is created. The members of this class must be a number, or a member of another known set. That is the difference between the enumerated sets in the PROPERTIES clause and the SETS clause. The valid method needs to compare with each member of the set to return the correct value associated with the input string and vice versa for the toString method. If the members of the set are members of another known set, the new class must use the method valid and toString of the known class. For example, if the new set is built from the members of _RESPONSE set:

  \[ newSet = \{ok, out-of-bound\} \]

The newSet class in JAVA is as below:
import _RESPONSE;
import java.io.*

public class _newSet {

  public static int valid(String string) throws IOException {
    if (string.compareTo("ok") == 0) return _RESPONSE._ok;
    if (string.compareTo("out_of_bound") == 0) return _RESPONSE._out_of_bound;
    throw (new IOException());
  }

  public static String toString(int value) {
    return _RESPONSE.toString(value);
  }
}

The members of this set are either numbers or members of another set which is already known, therefore the set is not added to list of sets.

• The tenth part of the analysed file is associated with the IMPORTS clause. The format of this part is:

  ((Import1), (Import2), ...)

For each imported machine, the name and the parameters of the machine are listed according to the following format:

  Name(Para1, Para2, ..):(Predicate):((?),(?),(?))

where Predicate is the constraint about the parameter of the machine.

The JAVA translator needs to extract the name and the parameters of the imported machine and add it to the global imported machine list. The parameters of the imported machine are the expressions consisting of constants, implementation parameters or members of sets. The expressions are translated token by token follow the rule:

- If the token is a implementation's parameter, input of the operation or a constant of the development, one underscore prefixed to the token.
- If the token is a member of set named Set for example, the translation for token become _Set._token.
- If the token is a constant of another development named Development, the translation becomes _Development._token.
- Otherwise, each token in an expression is translated to itself (including "(" and ")").

For example, in B one expression is:
\[(bb + 2) * ii + \text{max.size}\]
where \(bb\) is a local variable, \(ii\) is an input, \(\text{max.size}\) is implementation’s parameter. The JAVA code equivalents to that is:
\[(_bb[0] + 2) * _ii + _\text{max.size}\]

- The seventeenth part of the analysed file is associated with INITIALISATION clause. The substitution in the INITIALISATION of the implemenetation is listed here between "(" and ")":

(Substitutions)
The detail of the substitutions in the INITIALISATION is discussed in section 3.2.2. Creating constructor method.

- The last part of the analysed file is the copies of all operations in the implementation with all DEFINITIONS parsed and replaced. The format is:
\[((\text{Operation1}), (\text{Operation2}), ...\)]

The operation format also has different parts separated by ";", where the JAVA translator can extract the name of the operation, the input, output of the operation and the body of the operation which is a set of substitutions.

3.2.1 Creating the header

From the INITIALISATION part of the analysed file, the translation start writing into actual output file after collecting all the information needed. The output file name is the name of the development with underscore before that. In the \texttt{Student} array development, the name of the output file is \texttt{Student_array.java}

First it produces the header part of the output:

- For each set in global enumerated set list, create the classes for these enumerated sets. In the \texttt{Student} array example, the \texttt{RESPONSE} class is created.

- For each imported class in imported class list, import these classes in the output file. For example, in the implementation of the \texttt{Student} array the following classes are imported:

\begin{verbatim}
import _Bool_TYPE;
import _RESPONSE;
import Varr;
\end{verbatim}
• For each imported machine in the global imported machine list, declare an object of that class. For example, if the library machine Student Vari is used in the implementation, a private variable created as follow:

\[
\text{public class } \_\text{Student} \_\text{array} \{ \\
\quad \text{private } \_\text{Varr} \_\text{Student} \_\text{Varr}; \\
\quad \ldots \\
\}\n\]

• For each input of the implementation machine, if this is not a set (upper case word), a private variable is created. For example, \text{Student} \_\text{array} has two parameters, \text{max} \_\text{size} and \text{STUDENT}. \text{STUDENT} is a set and is ignored. A private variable is created for \text{max} \_\text{size} as follow:

\[
\text{private int } \text{max} \_\text{size};
\]

• For each constant in the list of constants, create a public final variables for it. For example, if a constant is defined as:

\[
\text{aConstant} = 2
\]

in the implementation, the \text{JAVA} translation for thatis:

\[
\text{public final static int } \_\text{aConstant} = 2;
\]

The naming convention is applied here and all the constants are integer (Type checking ignorance).

3.2.2 Creating constructor method

The constructor method's parameters are the parameters of the implementation, excepting set parameters. In the \text{Student} \_\text{array} example, the parameters of the implementation are \text{max} \_\text{size} and \text{STUDENT}, but only \text{max} \_\text{size} is the parameter of the constructor method.

\[
\text{public } \_\text{Student} \_\text{array}(\text{int } \_\text{max} \_\text{size}) \{
\}
\]

Within the constructor, the inputs of the constructor are save into the private variables:

\[
\text{this.} \_\text{max} \_\text{size} = \_\text{max} \_\text{size}
\]

The private variables are created in the header part of the translation.
Next, the objects for imported machine need to be created. By searching through the list of imported machines, the names and the parameters of these imported machines are found. For example, in `Student_array` implementation, `Student_Varr` library is used. The object associated with that library is created as follow:

```
.Student_Varr = new Varr("Student", "Student_array", _max_size);
```

where the names of the instance and the development are passed to the constructor method of `Varr` library class.

Finally, the `INITIALISATION` substitution of the implementation is translated in the constructor method. For translating substitution, see part 3.2.4. Translating substitution.

### 3.2.3 Translating operations

After translating constructor methods, the other methods in the development are translated. By getting the operation’s name, input and output (from the analysed file), the JAVA translator can translate the function name as follow:

```
public void Name(int[] output1, int[] output2, ..., int input1, int input2)
```

where `Name` is the name of the operation in the B implementation. The rest of the operation is the translation of the substitutions in the body of the implementation operation.

### 3.2.4 Translating substitutions

A block of sequential substitutions are set of substitutions seperated by ",;"

```
Substitutions -- > Sub;Substitutions
               -- > Sub
```

If `J` is the translation function from B into JAVA, the translation of Substitutions is:

```
J(Substitutions) = J(Sub);J(Substitutions)
```

```
= J(Sub)
```

Each substitution belongs to one of these five substitutions:

- Var substitution
- Begin-End substitution
- If substitution
- While substitution
- Case substitution
- Assignment

32
The grammar for single substitution is

\[ \text{Sub } \rightarrow \text{ Var} \\
\rightarrow \text{ Begin-end} \\
\rightarrow \text{ If} \\
\rightarrow \text{ While} \\
\rightarrow \text{ Case} \\
\rightarrow \text{ Assignment} \]

From the given grammar, the translation of one substitution is:

\[
J(\text{Sub}) = J(\text{Var}) \\
= J(\text{Begin-end}) \\
= J(\text{If}) \\
= J(\text{While}) \\
= J(\text{Case}) \\
= J(\text{Assignment})
\]

The translator need to understand the grammar for each of the above substitution in B implementation.

- **Var substitution**
  The grammar for each Var substitution is:

\[
\text{Var } \rightarrow \text{ VAR} \\
\rightarrow \text{ LocalVariables} \\
\text{ IN} \\
\text{ Substitutions} \\
\text{ END}
\]

Where the **LocalVariables** is list of the local variable used in the substitution.

\[
\text{LocalVariables } \rightarrow \text{ variable,LocalVariables} \\
\rightarrow \text{ variable}
\]

For the local variable, the translator need to keep track of the local variables along with the input and the output of the operation. Since the behaviour of the local variable is the same as output of the operation, these variables are stored in the same list. Similar to the output of the operation, the local variables are also arrays of integer. The reason for this is that local variables frequently receive the result of an operation. For example, in implementation, the local variables are declared as:

\[
\text{VAR} \\
\text{ ii, bb} \\
\text{ IN}
\]
The translation in **JAVA** is:

\[
\text{int \[1\] \(ii\) = new \text{int}[1];}
\]
\[
\text{int \[1\] \(bb\) = new \text{int}[1];}
\]

The translation for the **Local Variables** can be summarised as:

\[\begin{align*}
J(\text{LocalVariables}) &= \text{int \[1\] \_variable = new \text{int}[1];} \\
J(\text{LocalVariables}) &= \text{int \[1\] \_variable = new \text{int}[1]}
\end{align*}\]

The reason for that is the ignorance of type checking. Overall the translation of the **Var substitution** need to be placed between "{" and "}". The translation of the **Var substitution** can be summarized as:

\[J(\text{Var}) = \{J(\text{LocalVariables})\ \ J(\text{Substitution})\}\]

- **Begin-End substitution**
  The grammar for Begin-End substitution is:

\[
\begin{align*}
\text{Begin-end} & \quad -- \quad \rightarrow \quad \text{BEGIN} \\
& \quad \text{Substitutions} \\
& \quad \text{END}
\end{align*}\]

As before, if the **J** is translation from **B** to **JAVA**

\[J(\text{Begin-end}) = \{J(\text{Substitutions})\}\]

where **Substitutions** are the substitution in the **Begin-End** structure.

- **If substitution**
  The grammar for If substitution is:

\[
\text{If} \quad -- \quad \rightarrow \quad \text{IF Predicate THEN} \\
& \quad \text{Substitutions} \\
& \quad \text{Else} \\
& \quad \text{END}
\]

Consider the translation of If substitution, we can have:
\[
J(\text{IF Predicate THEN} \quad \text{if } (J(\text{Predicate})) \quad \{ \\
\quad \text{Substitutions} \\
\quad J(\text{Substitutions}) \\
\text{Else} \\
\quad J(\text{Else}) \\
\text{END}) \quad \}
\]

The \text{Else} part of the If substitution can be empty or an ELSE statement or ELSIF statement:

\[
\text{Else} \quad \text{-- > } \\
\quad \text{ELSE Substitutions} \\
\quad \text{ELSIF Predicate THEN Substitutions Else}
\]

If the \text{Else} part is empty, the translation of it is also empty. For the second case of \text{Else} the translation of it will be

\[
J(\text{Else Substitutions}) \quad = \quad \}
\quad \text{else} \quad \{ \\
\quad J(\text{Substitution}) \\
\text{END}
\]

If the \text{Else} part has \text{ELSIF} clause, the translation of that is:

\[
J(\text{ELSIF Predicate THEN Substitutions Else}) \quad = \quad \}
\quad \text{else if } (J(\text{Predicate})) \quad \{ \\
\quad \text{else } J(\text{Substitution}) \\
\quad J(\text{Else})
\]

From the grammar and translation formula above, the translation of \text{If substitution} can be done if the translation of predicate is done.

- \text{While substitution}

  The grammar for While substitution is:

  \[
  \text{While} \quad \text{-- > } \quad \text{WHILE} \\
  \quad \text{Predicate} \\
  \quad \text{DO} \\
  \quad \text{Substitutions} \\
  \quad \text{VARIANT} \\
  \quad \text{Expression} \\
  \quad \text{INVARIANT} \\
  \quad \text{Predicate} \\
  \quad \text{END}
  \]

  The translation of the \text{VARIANT} and \text{INVARIANT} parts can be ignored since it is only used for proving the correctness of the loop. The translation of the While
substitution can be expressed as:

\[
J(\text{While}) = \text{while } (J(\text{Predicate})) \{ \\
\quad J(\text{Substitutions}) \\
\} \\
\]

- **Predicate**
  For the translation of predicate, **JAVA** translation can translate tokens by tokens according to the following mapping:

  - `/=` becomes `!=`
  - `=` becomes `==`
  - `&` becomes `&&`
  - `or` becomes `||`
  - `not` becomes `!`

  - If the token is a implementation’s parameter, input of the operation or a constant of the development, adding one underscore before the token.

  - If the the token is local variable or output of the operation, since in **JAVA** it is an array of integer, so the translation for `token` becomes `token[0]`.

  - If the token is a member of set named `Set` for example, the translation for `token` become `_Set_token`.

  - If the token is a constant of other development named `Development`, the translation becomes `_Development_token`.

  - Otherwise, the token in predicate translated as itself (including "(" and ")").

For example, if the implementation has one predicate as follow:

\[
((\text{ii} > 1) \& (\text{bb} = \text{TRUE}))
\]

where `ii` is an input of the operation and `bb` is a local variable. The translation
of the above predicate is:

\[ ((i > 1) \&\& (bb[0] == \_BOOL\_TRUE)) \]

- **Case substitution**
  The grammar for Case substitution is:

  ```
  Case  -->  CASE Expression OF
           EITHER Expression THEN
           Substitutions
           Or
           END
  ```

  The translation for the Case substitution can be written as follow:

  ```
  J(Case) = switch (J(Expression)) {
    case (J(Expression)): J(Substitutions)
    break;
    J(Or)
  }
  ```

  For the Or part of the Case substitution, there are three possibilities:

  - **Or = Empty**
    If the Or part is empty, its translation is also empty:

    ```
    J(Empty) =
    ```

  - **Or = ELSE Substitutions END**
    In this case, the translation of Or part can be written as follow:

    ```
    J(ELSE Substitutions END) = default: J(Substitutions)
    ```

  - **Or = OR Expression THEN Substitution Or**
    The translation in this case is:

    ```
    J(OR Expression THEN Substitution Or) =
    case (J(Expression)): J(Substitutions)
    break;
    J(Or)
    ```
The translation of the Case substitution now reduce to the translation of Expression.

- **Expression**
  Similar to Predicate, the Expression is translated tokens by tokens. The mapping for tokens in an Expression as follow:

  - If the token is a implementation's parameter, input of the operation or a constant of the development, adding one underscore before the token.
  - If the the token is local variable or output of the operation, since in JAVA it is an array of integer, so the translation for token becomes \texttt{.token[0]}.  
  - If the token is a member of set named \texttt{Set} for example, the translation for token become \texttt{.Set._token}.
  - If the token is a constant of other development named Development, the translation becomes \texttt{.Development._token}.
  - Otherwise, the token in expression translated as itself (including "). (" and ")").

- **Assignment**
  There are two types of assignments: Assignment from an expression and assignment from Operation.

  - Assignment from an expression. The grammar for this assignment is simple as follow:

    $\textbf{Assignment} \rightarrow \textbf{Variable} ::= \textbf{Expression}$

    The \texttt{Variable} can be the local variable, output of the operation. Since this type of variable is an array of integer, so the translation of the assignment is written as:

    \[ J(\texttt{Variable ::= Expression}) = \texttt{.Variable[0]} = J(\texttt{Expression}) \]

  - Assignment from Operation. The grammar for this assignment is:

    $\textbf{Assignment} \rightarrow \textbf{ListVariables} <--- \textbf{OperationName(Parameters)}$

    The list of variables is list of local variables or output of the operations separated by ",":

    $\textbf{ListVariables} \rightarrow \textbf{Variable, ListVariables}$
    $\rightarrow \textbf{Variable}$
The parameter of the operation is set of expression seperated by ",":

\[\text{Parameters} \quad \rightarrow \quad \text{Expression, Parameters} \quad \rightarrow \quad \text{Expression}\]

Since the function in JAVA has no output, the output of the operation need is modelled as the array input of the function. The translation of the assignment of this type is:

\[J(\text{ListVariables} \leftarrow \quad \text{OperationName(Parameters)}) = J(\text{OperationName})(J(\text{List})J(\text{Parameters}))\]

By searching through the list of operation, the translator knows the translation for the operation name \((J(\text{OperationName}))\). The translation \(J(\text{ListVariable})\) is exactly the same as ListVariable . It is because the variables in the list already are arrays of integer. The translation of the Parameters are base on the translation of Expression.

\[J(\text{Parameters}) = J(\text{Expression}), J(\text{Parameters})\]

or

\[J(\text{Parameters}) = J(\text{Expression})\]

For example, an assignment from the operation \(\text{Student}_\text{SCH}_\text{LO}_\text{EQL}_\text{ARR}\) is:

\[bb, \text{index} \leftarrow \quad \text{Student}_\text{SCH}_\text{LO}_\text{EQL}_\text{ARR}(1, \text{max}_\text{index}, \text{student});\]

where \(\text{max}_\text{index}\) is implementation's parameter and \(\text{student}\) is input of the opearation. The JAVA translation for this assignment is:

\[\_\_\text{Student}_\_\text{Varr}_\text{SCH}_\text{LO}_\text{EQL}_\text{ARR}(\_\_\text{bb}, \_\_\text{index}, 1, \_\_\text{max}_\text{index}, \_\_\text{student});\]

The JAVA translation of the implementation is done by collecting information about sets, constants, implementation's parameters. The translation is done according to the translation function \(J\) applied to the grammar of the operations and substitution in the operations.
3.3 Linking the JAVA code

After translating the implementation into JAVA code, the code generated by the translator is only the class with no main method. An interface for the class is written, compiled with the generated code to produce the executable code. The interface for the implementation are based on the .itf file of the development. The format of the interface file is:

```
INTERFACE Name(Parameters)
OPERATIONS
    List of operations
END
```

In the interface file only the operations in the list of operations will appear in the translated interface. For example, the interface file for the `Student_array.itf` can be:

```
INTERFACE Student_array(5, SCALAR)
OPERATIONS
    Student_Read
END
```

Although the implementation of `Student_array` has two operations but only `Student_Read` will be in the interface according to the above `Student_array.itf` file.

Another point to notice is the parameters of the implementation are instantiated. The machine `Student_array` has two parameters. `STUDENT` is instantiated to set `SCALAR` while `max_size` is set to 5.

In order to create the interface for the implementation, the JAVA linker need to know which operations are to be translated and the values for the parameters of the implementation. By parsing the simple interface file, the linker knows the values of the machine parameters and the list of operations that needed to have an interface.

For the parameters of the machine, if it is a set, a new set is created in the same way as for constant set. The value assigned to a set can be another set, subrange of the natural numbers, or enumerated set.

The interface class for the development is an extension of Frame class. In the interface class, an object for the implementation is created. The parameter for the constructor to create this object is obtained from the interface file. For example, the object for `Student_array` implementantion is:

```
  Student_array = new Student_array(5);
```

The interface class need to get the input argument from the user, then call the correct operation of the implementation, passing the input to the operation, and finally display the output for that operation. Therefore, the JAVA linker need to know the format of the operations of the implementation. The format of the operations is in `.mri.typ` file in the
/TYP folder. For example, below is the type file for Student_array:

(Student_Read(resp , student;index) == (student : STUDENT & resp : RESPONSE & index : NAT));
(Student_Store(resp;index , student) == (resp : RESPONSE & index : NAT & student : STUDENT));

|... Ignored

the last part of the file is ignored since it is not related to the interface. The format of the operation in the type file is:

Name(outputs; inputs) == (Predicate)

Where predicate is the constraint to specify the type of the inputs and outputs of the operation. Since the operations of the implementation do not have any type checking. The interface class needs to check the type of the input before passed it to the operation.

All the inputs from user are strings. The interface class can use the method valid to check the correctness of the input. For example, if the input from user is string. The interface wants to get the input index for operation Student_Read. The following code attempts to get the index input until correct:

while (true) {
    try {
        .index = .NAT:valid(string);
    } catch (IOException e) {
        continue;
    }
    break;
}

The above code invokes the loop until valid input is entered. Similar code can be used to get other input.

For the output, the operations of the implementation only return integer number. By using the toString method of the type class, the interface can print the value of the output correctly. Consider the following code to print the value of the output resp for the operation Student_Read:

System.out.println("resp: " + RESPONSE.toString(_resp));

By knowing the format of the operations and using the methods valid and toString, a nice interface is generated for the implementation using the Frame class. Below is the picture of the interface created by JAVA frame.
Figure 5: JAVA frame interface
3.4 Changing in B-Toolkit interface

The interface of B-Toolkit need to be modified to include the new JAVA code generation. The changes are to the menus and environments as discussed above.

3.4.1 Changing in Translators/Compilers Option

A Programming Language Flag need to be defined for JAVA. In the source code, all the Programming Language Options are defined in two files:

B_MotifGlobals.c and
B_MotifGlobals.h

A new define clause for the JAVA language is added as below:

#define Prog_Language_Option_JAVA 5

along with other Programming Language Option definitions such as C, ANSI C, PAS-CAL

The Translators/Compilers Options in the interface are create in the file B_Motif_Widget.c. The function to create these options are:

Popup_NewTranslatorsCompilers_Options_Dialog() and
Create_NewOption_Process_TranslatorsCompilers()

The JAVA options can be added in between a #ifdef and #endif directive in order to be enabled only when the JAVA flag is set during compiling process of the B source code. The new Options is display below:
Figure 6: Translator/Compiler Option
3.4.2 Changing the translate button in Translator environment

By pressing the translate button, a function `Trl()` and subsequently `Pre_Trl()` function in `B_Motif.c` are called. The programming language option is checked in `Pre_Trl()` function. In `Pre_Trl()` function, added the checked for JAVA language. A toolkit containing theories is run after `Pre_Trl()` by calling `Invoke_B()` function. For translation process, Toolkit 1 is called. In Toolkit 1, by checking the language flag option, the B-Toolkit knows which of the toolkit theories should be call next in order to complete the translation process.

```c
SetWriteToXTerm &
LoadToolkit(35)
=>
run_Bplf1(Prog_Language_Flag_PASP);
SetWriteToXTerm &
LoadToolkit(7)
=>
run_Bplf1(Prog_Language_Flag_C);
```

Originally in the Toolkit, if the PASCAL language is chosen, the Toolkit 35 is loaded. Otherwise, Toolkit 7 is called to translate the C code.

As discussed before, since the syntax of the theories is unknown, new Toolkit 40, the JAVA theory is made as a copy of the C theory file. The other theory files `Translators-JAVA.src` is a copy of `Translators.src`, `Trans-JAVA.src` is a copy `TransC.src`, `Utils-JAVA.src` is a copy of `Utils.src`. They all support to implement Toolkit 40. Added the following rules in Toolkit 1 to redirect the translation process to load Toolkit 40 when the language is set to JAVA:

```c
SetWriteToXTerm &
LoadToolkit(40)
=>
run_Bplf1(Prog_Language_Flag_JAVA);
```

The Programming Language JAVA flag needs to be defined for in the theory file. As for other language option, the JAVA flag is defined in the script file name `GlobalsDef.src`. The following define directive is added into the global definition source file:

```c
#define Prog_Language_Flag_JAVA 5
```

The different from C toolkit translation within the JAVA toolkit is the redirection of calling external script. In the C toolkit, in particular `TransC` script, the `TransCFile` shell script is called as below:

45
/***/
 TransCFile:
$1 name (eg, BoolTYPE)
$2 1=>TransCFilter/NLFilter - from Translators 0=>skip - from Enum.src
$3 compiler string
$4 0=>Motif 1=>Non-Motif
$5 0=>ANSIC 1=>non_ANSIC
***/
WriteDot &
brule(TransC.parse_implementationX.6,z) & /* 0=>Motif 1=>Non-Motif */
brule(TransC.parse_implementationX.3,c) & /* cc flags etc. */
brule(TransC.parse_implementationX.2,n) & /* 0=>ANSIC 1=>non_ANSIC */
check_error(a)
=>
trans_c2_1(a.imp)(M);

TransCFile takes inputs such as the name of the machine, language flag, etc. Similar
for TranslatorsJAVA.src, the external script can be called from inside the toolkit. A new
shell script named TransJAVAFile is created. The TranslatorsJAVA.src is modified to
called TransJAVAFile by replacing TransCFile with TransJAVAFile. The external JAVA
translator is called by the shell script TransJAVAFile. Therefore the input of shell script
depends on the external translator. Since the translator needs the name of the implementa-
tion and the name of the development as inputs. The script can be modified as follows:

/***/
TransJAVAFile:
$1 name of the implementation(eg, ArrayI)
$2 name of the top level machine (eg, Array)
***/
WriteDot &
bshell(bcatl("$BKIT/BLIB/TransJAVAFile ",a," ",M)) &
check_error(a)
=>
trans_c2_1(a.imp)(M);

After code is generated, shell script CreHtmlc is called to document the C code into
Html. CreHtmlc is called in the TransC.src script. In order to have the new shell script
CreHtmlJAVA called when the chosen language is JAVA, the corresponding script Trans-
java.src is modified with CreHtmlJAVA replaced CreHtmlc. In the shell script Cre-
HtmlJAVA, Html documents are generated by running javadoc command. The generated
documents are placed in **HTX/JAVA** directory.

3.4.3 Changing the interface button in Generator environment

Another change required in the **B-Toolkit** interface is the interface button. When an interface of a machine is implemented, the interface button is used to translate and link the code together to produce the executable code. Similarly for **JAVA**, by pressing interface button (\(itf\)) when the flag is set to **JAVA**, the toolkit needs to generate a top-level interface for the **JAVA** class and compile the development. Currently, in the **B-Toolkit** during the linking process, the shell script *LinkCFile* is run eventually. The **JAVA** linker is plugged into this shell script. By checking the existence of the **C** code, the shell script can determine to run the **JAVA** linker or **C** linker accordingly. The **JAVA** linker produce a top level interface class for the development. After that, in the shell script, the *javac* command is run to compile the whole development.

3.4.4 Changing the execute button in Translator environment

After generating and compiling, the executable code is invoked by execute button (*exe* in **Translator** environment. When the button is pressed, the executable code is invoked in the **EXE()** function in **B.Motif.c**. By checking the *Programming Language* flag, the toolkit can be redirected to run the external **JAVA** code instead of executable **C** code.
4 Conclusion

In conclusion, the objective of the project is met. JAVA code can be generated after the B development is fully implemented. The generated code is also executable by an interface produced by the toolkit. The interface is written using JAVA’s Frame class. A nice document for the JAVA code is generated for maintenance purpose. There is still some limitation to the current version of the JAVA translator and hence some possible improvement can be done in the future.

4.1 Limitation

The first limitation of the JAVA translator is due to the lack of documentation on the source code. The current version of the translator is really external to the B-Toolkit. The Toolkit might be improved by creating an internal translator, using theories. The internal translator much more consistent than the external translator and it would be more consistent with the rest of the B-Toolkit.

It would also reduce the overhead of translating C code in the current system. Before translating JAVA, since the theory is copied from the theory for translating C code, so the C code is still generated before deleting by the external script.

The JAVA translator is only tested on quite small number of developments. Since it is not using the theory, therefore there are no formal proof for the correctness of the translator. If the internal translator is produced, the correctness can be guaranteed.

4.2 Suggestion

Clearly, the first possible project is to improve the JAVA translation process by creating an internal translator using theories. The internal translator can get over the limitation of the current translator. Since all the JAVA libraries are already implemented, only the translator and the linker are needed. The grammar for the JAVA translator is also known. So with the knowledge about the format of the theories, the internal translator should be straightforward.

Another improvement for the B-Toolkit is the extension of the library, not only for the JAVA version but for all the B-Toolkit implementations. More library machines can be added into the B-Toolkit’s SLIB to support the implementors.

From the knowledge about the B-Toolkit source code, other programming languages can be generated along the current set of programming languages. C++ code generator should be easy supported in addition to since it is close to C and has the object-oriented advantage of JAVA. Other types of programming languages can be generated such as functional programming languages like Haskell.

Since the code generated contains only the functions of the implementation, an interface for the code needs to be generated. Currently, the integration between different programming languages can be done. Therefore the interface for the JAVA code might not be JAVA’s Frame class but might be C code using the Motif library. Moreover,
current interfaces for the C code is either text base (non-Motif) or graphical (Motif). Unfortunately, the Motif library is copyrighted and not every machine can have this library installed. A graphical interface for the C code could be built using JAVA’s Frame. This would give the users a nice graphical interface over the top of the C code even though they do not have the Motif library.

There are some difference between the C code and the JAVA code generated. The first difference is the handling of strings. In C, the string is regarded as list of bytes. Therefore C code only supports string of limited length. By using the global string file, the JAVA code can handle strings of any length. This could be an advantage but it also creates a new problem when the run-time program depends on the physical storage of the computer. The C code generation could be modified to handle strings as in the JAVA code generation (using the global string file) and vice versa. The advantages and disadvantages need to be considered for each method and the code generation can be changed to the better method.
A  B-Toolkit interface
Figure 7: Utils menu
Figure 8: Introduce menu
Copyright(c) 1985–2001 B-Core(UK) Ltd

Current Directory:

/home/htson/DEM01/StudentArray

Figure 9: Construct menu
Figure 10: Remake menu
Figure 11: Browse menu
Figure 12: Options menu
Figure 13: Help menu
Figure 14: Main environment
Figure 15: Proof environment
B-Toolkit
Release 5.0.18

Copyright (c) 1985–2001 B-Core(UK) Ltd

Current Directory:
/home/htson/DEMO1/StudentArray

Generating Motif INTERFACE Student_array .......
Generated executable interface code
Execute in the Translators Environment

Figure 16: Generators environment
Figure 17: Translators environment

61
Figure 18: Document environment
B  Student_array example

B.1  B implementation

IMPLEMENTATION Student_array_imp
SEES
  Bool_TYPE
REFINES
  Student_array
IMPORTS
  Student_Varr(STUDENT, max_size)
INVARIANT
  array = Student_Varr
INITIALISATION
  skip
OPERATIONS
  resp, student <--- Student_Read(index) =
  BEGIN
    IF (index <= max_size) & (index >= 1)
    THEN
      resp:=ok;
      student <--- Student_VAL_ARR(index)
    ELSE
      resp:=out_of_bound;
      student <--- Student_VAL_ARR(max_size)
    END
  END

  resp <--- Student_Store(index, student) =
  BEGIN
    IF (index <= max_size) & (index >= 1)
    THEN
      resp := ok;
      Student_STO_ARR(index, student)
    ELSE
      resp := out_of_bound
    END
  END

  resp <--- Student_Exchange(index1, index2) =
  BEGIN
    IF (index1 > max_size) or (index1 < 1)
    THEN

63
resp := out_of_bound
ELSIF (index2 > max_size) or (index2 < 1)
THEN
  resp := out_of_bound
ELSIF index1 = index2
THEN
  resp := same_index
ELSE
  resp := ok;
  Student_SWP_ARR(index1, index2)
END
END;

resp, index <----- Find(max_index, student) =
VAR
  bb
IN
  IF (max_index <= max_size) & (max_index >= 1)
  THEN
    resp := ok;
    bb, index <----- Student_SCH_LO_EQL_ARR(1, max_index, student);
    IF bb = FALSE
    THEN
      index := 0
    END
  ELSE
    resp := out_of_bound;
    index := 0
  END
END
END
B.2 C code

```c
#include "Student_array.h"

#include "Student_Var.h"

#include "BoolTYPE.h"

void INLStudent_array()
{
    INLStudent_VAR();
    ;
}

void Student_Read(*resp, *student, _index)
int *resp, *student, _index;
{
    if (_index <= Student_arrayP1 && _index >= 1 ) {
        *resp = ok;
        Student_VAL_ARR(_student, _index);
    } else {
        *resp = out_of_bound;
        Student_VAL_ARR(_student, Student_arrayP1);
    }
}

void Student_Store(*resp, _index, *student)
int *resp, _index, *student; {
    if (_index <= Student_arrayP1 && _index >= 1 ) {
        *resp = ok;
        Student_STO_ARR(_index, *student);
    } else {
        *resp = out_of_bound;
    }
}

void Student_Exchange(*resp, _index1, _index2 int *resp, _index1, _index2; {
```
if ( _index1 > Student_arrayP1 || _index1 < 1 ) {
    *resp = out_of_bound;
}
else if ( _index2 > Student_arrayP1 || _index2 < 1 ) {
    *resp = out_of_bound;
}
else if ( _index1 == _index2 ) {
    *resp = same_index;
}
else {
    *resp = ok;
    Student_SWP_ARR(_index1, _index2);
}

void Find(*resp, _index, _max_index, _student)
int *resp, *index, _max_index, _student;
{
    int bb;
    if ( _max_index <= Student_arrayP1 && _max_index >= 1 ) {
        *resp = ok;
        Student_SCH_LO_EQL_ARR(&bb, _index, 1, _max_index, _student);
        if ( bb == FALSE ) {
            *index = 0;
        }
    }
    else {
        *resp = out_of_bound;
        *index = 0;
    }
}
B.3 JAVA code

```java
import _Bool_TYPE;
import _RESPONSE;
import Varr;

class _Student_array {
    private Varr _Student_Varrr;
    private int _max_size;

    _Student_array (int _max_size) {
        _max_size = _max_size;
        _Student_Varrr = new Varr("Student", "Student_array", _max_size);
    }

    void _Student_Read(int [] _resp, int [] _student, int _index) {
        if (_index <= _max_size && _index >= 1) {
            _resp[0] = _RESPONSE_ok;
            _Student_Varrr.VAL_ARR(_student, _index);
        } else {
            _resp[0] = _RESPONSE_out_of_bound;
            _Student_Varrr.VAL_ARR(_student, _max_size);
        }
    }

    void _Student_Store(int [] _resp, int _index, int _student) {
        if (_index <= _max_size && _index >= 1) {
            _resp[0] = _RESPONSE_ok;
            _Student_Varrr.STO_ARR(_index, _student);
        } else {
            _resp[0] = _RESPONSE_out_of_bound;
        }
    }

    void _Student_Exchange(int [] _resp, int _index1, int _index2) {
        if (_index1 > _max_size || _index1 < 1) {
```

67
\_resp[0] = \_RESPONSE\_out\_of\_bound;
}
else if ( \_index2 > \_max\_size || \_index2 < 1) {
    \_resp[0] = \_RESPONSE\_out\_of\_bound;
}
else if ( \_index1 == \_index2) {
    \_resp[0] = \_RESPONSE\_same\_index;
}
else {
    \_resp[0] = \_RESPONSE\_ok;
    \_Student\_Varr\_SWP\_ARR( \_index1 , \_index2);
}
}

public void \_Find(int [ ] \_resp, int [ ] \_index, int \_max\_index, int \_student) {
{
    int [ ] \_bb = new int[1];
    if ( \_max\_index <= \_max\_size && \_max\_index >= 1) {
        \_resp[0] = \_RESPONSE\_ok;
        \_Student\_Varr\_SCH\_LO\_EQL\_ARR( \_bb, \_index, 1, \_max\_index, \_student);
        if ( \_bb[0] == \_BOOL\_FALSE) {
            \_index[0] = 0;
        }
    }
    else {
        \_resp[0] = \_RESPONSE\_out\_of\_bound;
        \_index[0] = 0;
    }
}
}
C  Manual on JAVA translation

Below are the listing of the B source code directory:
B2DOORS
BDEMO
BEESRC
BHELP
BPALETTE
C
ERROR_LOG
INFO
INSTALL_SCRIPTS
LEX
LIB
MOTIF
SCRIPTS
SRC
UTIL
WWW_DEMOS

The B source code is compiled by running the shell script BUILD in UTIL directory.
To include the JAVA version of the translator, the flag for the JAVA version need to be set. To set the flag, edit the environment variable JAVA_VERSION in the shell script set_compile_variables to YES.
setenv JAVA_VERSION YES

The compiled B-Toolkit is place in the folder BKIT in the B source code directory.
The compilation of the B-Toolkit source code requires the MOTIF library. The environment variables in the set_compile_variables need to be adjusted for different machine.
The new files added in the B source code directory are:

- In MOTIF folder, the modified files are: BMotif.c, BMotif_globals.c, BMotif_globals.h, BMotif_widgets.c, BMotif_displaystate.c.

- In SRC folder, the modified files are: Toolkit1.src, GlobalDef.src, CIOM.src. The new files are: Toolkit40.src, TransJAVA.src, TranslatorsJAVA.src, UtilsJAVA.src.

- In SCRIPTS folder, the script LinkCFile is modified and the new scripts are: TranssJAVAFile, CreJAVADir, CreHtmlJAVA, CreHTXDir

- In C folder, the new external translator is added in including: linker.c, translation.c, utils.c, utils.h

- In LIB/CDE folder, the new directory contains the library for JAVA is added.
• In UTIL folder, some scripts is modified for compiling new JAVA translator: *set_compile_variables, BUILD, _CRE_toolkit_binary_raw, CRE_toolkit_binary_all*. Only one new script is *CRE_JAVA_translator*.

All the B-Toolkit source code is zipped in the file named *bsource.zip* in the directory *bsource* directory of the CD submitted along with this report.

Since the source code is placed on the CD so it needs to be copied into the local disk before compiling. Also in the CD, the DEMO directory contains the development that is successfully testing in remake-able form. Copy these demos on the local disk, run the B-Toolkit on each development and used the Remake option in the Remake menu. The JAVA code can be generated by changing the Translator/Compiler option in Option menu to JAVA. The code is generated by button *trl* in Translators environment of the development. An interface is created by *itf* in Generators environment and can be run by *exe* button in Translators environment.

This report is contained in the report directory of the CD.