B2ViSiDiA - User Manual

Mohamed Tounsi
tounsi@labri.fr

Loïc Martin
loic.martin@labri.fr

June 22, 2010

Contents

1 Software installation **1**

2 B2ViSiDiA functionalities **3**
2.1 Graphical User Interface . . . 3
2.1.1 Global view . . . 3
2.1.2 Left menu and toolbars . . . 4
2.1.3 Edition panel . . . 5
2.1.4 Console panel . . . 5
2.1.5 Translate panel . . . 5

3 Developing your own algorithms **7**
3.1 General considerations . . . 7
3.2 Write an algorithm . . . 8
3.2.1 The name . . . 8
3.2.2 Invariants . . . 8
3.2.3 Events . . . 9
3.3 Examples . . . 12
3.3.1 Spanning Tree in LC0 . . . 12
3.3.2 3-coloring of a ring in LC1 . . . 13
3.4 Check . . . 15
3.5 Translate . . . 15
3.6 Run ViSiDiA . . . 16

A Install Java **17**
A.1 Troubleshooting . . . 17
A.1.1 GNU Java Compiler (GJC) . . . 17
A.1.2 OpenJDK . . . 18

B B2Visidia language **19**

Chapter 1

Software installation

B2ViSiDiA requires Java 6 or greater to run properly. Please refer to appendix A for information about Java installation.
You must have three archives nammed:

- visidia.jar
- JFlex.jar
- tom-ant.jar

These archives will be placed in the folder called `$B2Visidiaroot/WEB-INF/Lib/`.

Once a recent Java Virtual Machine (JVM) is installed, download `B2Visidia.jar` and run it either by double-clicking on it, or from the command line:

```
$ java -jar B2Visidia.jar
```

Please visit the website `http://visidia.labri.fr` or `http://www.labri.fr/perso/tounsi` for more information.

Chapter 2

B2ViSiDiA functionalities

2.1 Graphical User Interface

2.1.1 Global view

When B2ViSiDiA starts, the main window appears (figure 2.1). The Graphical User Interface (GUI) consists in four parts:

- at the center, a tabbed panel for displaying different code views (a single tab for Event-B edition, and a tab for each Java translation);
- at the left, a textual menu-like toolbar listing software functionalities relative to Event-B edition;
- at the top, a toolbar with icons representing the most useful functionalities associated to the current view;
- at the bottom, a secondary (optional) toolbar used for Event-B symbols.

The contents of toolbars change depending on B2ViSiDiA running mode. Either you are editing a Event-B specification, or you are visualizing a Java translation.

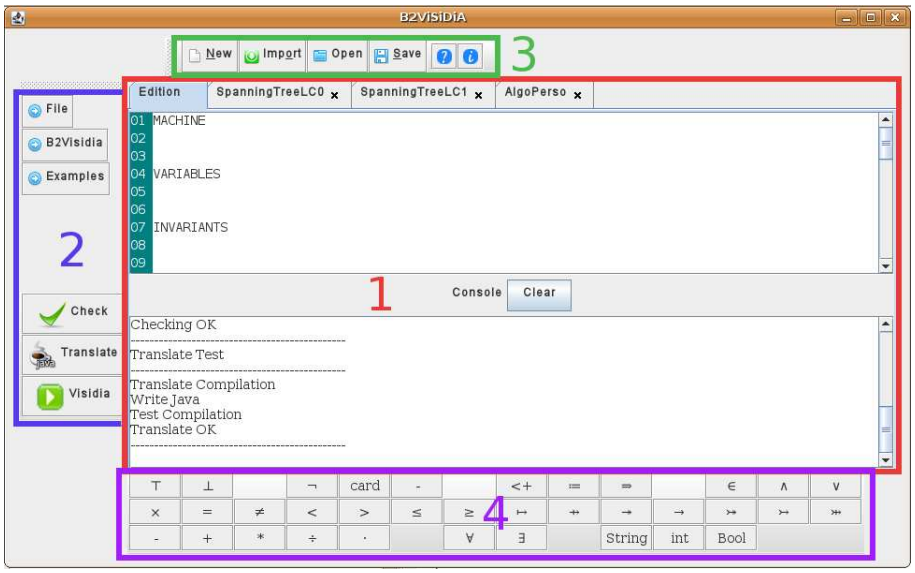

Figure 2.1: B2ViSiDiA main window
1: code viewer panels, 2: menus, 3: File Toolbar, 4: Edit Tools

2.1.2 Left menu and toolbars

Toolbar management

Toolbar position cannot be changed in B2ViSiDiA window. However, toolbars can be detached (floated) or attached (docked) to the window. To detach a toolbar from the window, just drag it outsine the window. To attach a toolbar, use the close button in the floating toolbar window.

Edition

You can create an empty new algorithm, load a algorithm from a file and save it to a file. The file extension used is b2v.

You can import a RODIN algorithm with “import” button. the import file must have the *bum* extension.

When importing, the filename is used for the name of the algorithm.

Figure 2.2: B2ViSiDiA File Menu

Edit Tools

This toolbar lets you insert special characters used in Event-B or the type of data manipulated.

It can be hidden via the menu.

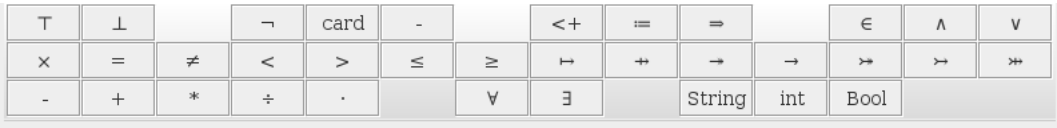

Figure 2.3: menu to insert special characters

Translate

You can check your algorithm. If the test is good you can turn it into a file readable ViSiDiA. It is possible to launch ViSiDiA.

Figure 2.4: The translates options

Examples Menu

To help you write your own programs, some examples tested are available. They are known and proven algorithms.

2.1.3 Edition panel

It is a simple text editor with line numbers.

Standard features of the edition are available as copy, paste, undo and redo. They are used with the usual keyboard shortcuts.

In the next chapter, we explain how to write an algorithm. (see chapter 3)

2.1.4 Console panel

All relevant information, allowing for example to locate a clerical error, are displayed in this context.

Also posted messages compilation to understand what the tool B2ViSiDiA.

It is possible to erase the history for readability.

2.1.5 Translate panel

After processing, the generated Java code is shown to help understand how the algorithm work in ViSiDiA.

Each transformation is associated with a tab, it is possible to have several open at once.

Chapter 3

Developing your own algorithms

Event-B is a formal method for developing software system. The two basic constructs in doing formal developments in Event-B consist in the machine and the context. The first one describes the dynamic properties of the specification and the second one defines how the machine is parameterized and can thus be instantiated.
For B2Visidia, only the machines which can be translated. They will constitute thereafter the relevant part of the resulting code. Hence, to avoid a course of context by B2Visidia, we fix the following parameters names:

- "g" to encode edges.
- "ND" to encode nodes.
- "ID" to encode a distinct identifier number of a node in the graph.
- "card" to encode the node degree (number of its neighbors).

In the other hand, an Event-B machine is considered as suitable for B2Visidia if and only if some considerations are taking into account in the modeling phase. In this section, we give details about how putting into practice these considerations.

3.1 General considerations

We recall that, machine may contain sections that serve only to prove the correctness of the design of distributed algorithm and which are not adequate for expressing the functional aspect of the algorithm. After doing a research study, we are able to point out that a suitable machine for B2Visidia is described by a sub-set of Event-B language containing the following sections :

MACHINE includes the name and the synchronization type of the algorithm.

VARIABLES contains the list of variables of the algorithm.

INVARIANTS defines the functions which encode nodes and edges labels.

EVENTS describes all the algorithm rules.

In the following, we show a straightforward description of a translatable machine of a distributed algorithm specification.

MACHINE
<machine_name> # synchronisation
VARIABLES
<variable_name>
.....
INVARIANTS
<label> : <invariant> # type
.....
EVENTS
<event>
.....
END

3.2 Write an algorithm

3.2.1 The name

The name of the machine is also the name of the produced Java file. It can contain numbers or letters, but it must begin with a letter. In the same section, it is possible to add synchronization annotation directly after the name. Rodin annotations are used to provide a further guidance for the tool and they are an integral part of the defined language. We defined annotation as a comment added to the specification to guide the translation process. It is differentiated from other comments by the "#" character. The three choices are $\#LCO$, $\#LC1$ and $\#LC2$.

3.2.2 Invariants

The definition of invariants is very important. They specify functions that allow to extract and to update states of nodes and edges in the graph. These functions are used in the Events section to define the rules of the algorithm. Invariant must follow a specific format[1] :

Definition

<invariant> ::= <ident> '∈' 'ND' <relational_set_op> <set_expression> # <type>
| <ident> '∈' 'g' <relational_set_op> <set_expression>
<relational_set_op> ::= ↔ | ⇜ | ⇿ | ⇻ | ↛ | → | ⤔ | ⤖ | ⤀ | ↣ | ⤗

<*ident*> that is placed on the left hand side of the operator '∈' represents the name of the function. The first form defines functions to control states of nodes and the second form defines functions to control states of edges. Considering the presence of <type>, <set_expression> is currently ignored and it is not analyzed. The type can take as values *String*, *int* or *Bool*. Due to some limitations of Visidia tool, <type> must be put only for nodes. Also, designer can specify as invariant the node that he want but there is a one on the edges. We give in the following some examples:

[1]We use the Extended Backus-Naur Form (EBNF) to describe syntax. In that notation, non-terminals are surrounded by angle brackets and terminals are surrounded by single quotes

Examples

- $inv1 : Mark \in g \rightarrow Bool$. Mark is a Boolean function. It expresses the state of edges, where the "true" value decrypts the state of a marked edge and the "false" value decodes the state of a non-marked edge.
- $inv2 : Lab \in ND \rightarrow nodeSet\#int$. Lab is a numerical function. It encodes the state of nodes.
- $inv3 : Color \in ND \rightarrow Set\#String$. Color is a function of String type. It encodes the state of nodes.

3.2.3 Events

Each event must be placed successively between the keywords $EVENTS$ and END already written in the editor. The event start with this name and stop with END. In the Event-B language, we count exactly three possible forms of an event :

(1) *ANY* **t** *WHERE* **G** *THEN* **S** *END*
(2) *SELECT* **G** *THEN* **S** *END*
(3) *BEGIN* **S** *END*

However, only the first event form can perfectly retrace a rewrite rule of a distributed algorithm. It can express through its guard the triggering condition as well as the forbidden context (if there exists) of a rule. Also, it can express through its substituting section the rule action. Formally, it is decomposed of local variables t, a guard G that specifies under which circumstances it might occur and then some generalized substitutions called actions S which specify how to change the variable states. When guards are evaluated to true, the event is triggered; it modifies the system state by executing actions on variables. The structure of an event in B2Visidia in presented in the following. Notice that guards and actions are labeled.

<event_name>
ANY
<variable_name>
.....
WHERE
<label> : <condition>
.....
THEN
<label> : <action>
.....
END

However, the events depends on the used type of synchronization. We explain in the following the particularity of each synchronization types.

LC0 algorithms

ANY: in this section, designer can declare names of adjacent nodes in the network that will perform a computation step. Also, he can declare the name of edge (if it is needed) which relies the two declared nodes.
WHERE: conditions must be written to one after the other beginning with "$grdN$:". Due to some limitations in Visidia tool, this section contains only conditions on the

nodes state. In order to extract the state of a node we have to use the following syntax :

Definition

<function> ::= 'ident' 'p_a_c' 'ident' 'p_a_c'

We mean by 'p_a_c' a series of brackets, braces or brackets and by 'ident' a character string. The first 'ident' represent the name of the function and the second one corresponds to the variable identification. For example "Color [{x}]" represents the label (having the type String) of the node x. A condition can be simple or complex (conjunction or disjunction of predicates). Other details are presented in Section B. However, only the set of explicit and deterministic relations operators are translatable:

Definition

<relop> ::= '>' | '<' | '≤' | '≥' | '=' | '≠'

We give in the following some examples of condition:

Examples

- "$grd1$: Color [{x}] = {Black} ∧ Lab (x)= 45" : This condition checks if the Color of the node x is Black and its Lab is equal to 45.
- "$grd2$: Color [{x}] = {Red} ∨ ¬ Lab (y) < 4 ": This condition verifies if the Color of the node x is Red or the Lab of its neighbor y is not lower than 4.

THEN: This section includes actions (an action must start with "$actN$:") which change some labels in the graph. An action is translatable by B2Visidia if and only if is deterministic. Formally, two cases are possible: in the first one, an action can be made of a function followed by ':=' and afterwards by an expression that must have the same type of the function. This type of action is strongly recommended if the function will change only one node (or edge) label. Otherwise, an action is defined by the name of a function followed by ':=' then the name of the function afterwards by the overwriting operator '⩤' and finally a set of nodes [resp. edges] identifiers given with their new labels.

Examples

- "$act1$: Color (x) := Green " : The new Color of x is Green.
- "$act2$: Lab := Lab ⩤ {x ↦ 12, y ↦ 13} " : This action changes two node states in the same time. The new Lab of x is 12 and the new Lab of y is 13.
- "$act3$: Mark (x ↦ y):= ⊤ " : The edge between x and y becomes marked.

LC1 algorithms

ANY: For LC1 and LC2 synchronization algorithms, we fix the choice of the name of the node center. We define "c" as the center of the star and so all the neighbors of "c" are specified by "g[{c}]". We distinguish this special node because it is considered as an essential information for the translation processes. In some case, designer needs to check properties on only one or a few neighbors nodes of "c", so he can set parameters as he wants. The designated parameters correspond to distinct nodes. Therefore, universal quantification predicate can be used to browse all neighbors of "c". Also, it allows to define new states of edges (and leaves for LC2) by means of local function. A local function must be declared in ANY section, defined in

WHERE section and used in THEN section.
WHERE: Compared to LC0 algorithm, LC1 gives other possibilities to control labels of neighbors nodes and edges of the star. For this end, we can use only the "for all" operator in the predicate. A predicate can be expressed by the following way:

Definition

<predicat> ::= '∀' 'ident_list_comma' '·' <ident_dec> '∧' <condition> '⇒' <condition>
| '∀' 'ident_list_comma' '·' <ident_dec> '⇒' <condition>
<ident_dec>::= <expression> '∈' <expression>
| <expression> '∈' <expression> '∧' <ident_dec>
<ident_list_comma>::= 'ident'
| 'ident' ',' <ident_list_comma>

<ident_list_comma> denotes a finite sequence of variables representing the list of neighbor's nodes identifiers. However this list must be different from the node center identifier "c" and from reserved words. The following are the list of the reserved word : ND, g, ID, card, MACHINE, VARIABLES, INVARIANTS, EVENTS, ANY, WHERE, THEN, END, LC0, LC1, LC2.

We give in the following some examples to illustrate the use of the predicate. In these examples, we suppose that we have a rule which will activate some edges in the star. To specify this rule, we declare in the "ANY" section a local function that we call "new_edge_label". After that, this function is defined and parameterized in the "WHERE" section. Formally, it is defined as a function mapped from each edges (or nodes for LC2 algorithm) in the star to a label set. For B2Visidia, only the parametrization is required.

Examples

- "$grd1 : \forall\ x \cdot x \in g[\{c\}] \Rightarrow$ Lab(x)= 14 " : This condition checks if Lab of all the neighbors are equal to 14.
- "$grd2 : \forall\ x \cdot x \in g[\{c\}] \wedge$ Color(x)=Green $\Rightarrow$ new_edge_label $(c \mapsto x) = \top$ " : This condition parameterizes the new_edge_label function. It activates edges connecting "c" with nodes having Green as a Color.
- "$grd3 : \forall\ x \cdot x \in g[\{c\}] \wedge$ Color(x)=Green $\Rightarrow$ Lab(x)= 8 " : This condition checks if Lab(s) of the neighbors nodes, that having a Green Color, are equal to 8.

For LC1 and LC2, edge must be defined by $c \mapsto X$ where X is the neighbor of the node center.
THEN: in this section, an other possibility to write actions is added. Now, designer can overwrite some edges labels by the new elements of local functions. The following example illustrates this possibility.

Example

- "$act1$: Mark:= Mark $\lhd$ new_edge_label " : The edge between "c" and Green neighbors becomes marked.

LC2 Algorithm

The specific feature of LC2 algorithms is that the labels attached to the leaves of the star [2] may be modified according to some rewriting rules. To specify this feature, designer must define new states of leaves by means of local functions. These functions

[2] We call a star, a node together with its neighbors. We refer to these neighbors as the leaves of the star

allow to hold the new states of the star leaves and to overwrite elements of functions declared in the Invariant section.

To explain how to use the universal quantification predicate to update states of leaves, we give an example of a local function called local_Lab2 :

Example

- " grd1: $\forall$ a $\cdot$ a $\in$ g{[c]} $\wedge$ Lab(a)=10 $\Rightarrow$ local_Lab2 (a) = 11" : This condition parameterizes the local_Lab2 function.

In Event-B, local_Lab2 is defined by the following guard : local_Lab2 $\in$ ND $\rightarrow$ nodeSet. We note that, only "grd1" is filtered by B2Visidia since the definition of local_Lab2 is not required for the translation. In the assignment component, this action overwrites the leaves states.

Example

- "act1 : Lab2 := Lab2 $\lhd$ local_Lab2" : This action changes Lab(s) of "c" neighbors from 10 to 11.

3.3 Examples

In this section, we present some examples of distributed algorithms specifications that are tested by our tool. These specifications are written by Rodin and filtered by B2Visidia.

3.3.1 Spanning Tree in LC0

This example shows a search spanning tree in a graph LC0.

Algorithm presentation

Assume that a unique given processor is in an "active" state (encoded by the label A), all other processors being in some "neutral" state (label N) and that all links are not marked. The tree initially contains the unique active node. At any step of the computation, an active node may activate one of its neutral neighbors and mark the corresponding link. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the marked edges.

Algorithm specification (b2v file)

MACHINE
SpanningTreeLC0 #LC0
VARIABLES
lab1
Mark
INVARIANTS
inv1 : Lab1 $\in$ ND $\rightarrow$ VisualLabel #String
inv2 : Mark $\in$ g $\rightarrow$ VisualEdgeMark

EVENTS
Spanning
ANY
X
Y
u
WHERE
grd1 : Lab1[{X}] = {A}
grd2 : Lab1[{Y}] = {N}
THEN
act1 : Lab1(Y) := A
act2 : Mark(u) := ⊤
END
END

3.3.2 3-coloring of a ring in LC1

This example shows how to color ring with three colors (A, N, E). In other words, the algorithm can ensure such that no two successive nodes are colored the same. We notice that initially nodes are labeled at random.

Algorithm specification (b2v file)

MACHINE
Coloring #LC1
VARIABLES
Color
INVARIANTS
inv1 : Color ∈ ND → VisualLabel #String
EVENTS
SameNColor
ANY
c
X
Y
WHERE
grd1 : Color[{c}] = Color[{X}] ∧ Color[{c}] = Color[{Y}]
grd2 : Color[{c}] = {N} ∨ Color[{c}] = {E}
THEN
act1 : Color(c) := A
END

SameAColor
ANY
c

```
            X
            Y
    WHERE
            grd1 : Color[{c}] = Color[{X}] ∧ Color[{c}] = Color[{Y}] ∧ Color[{c}] = {A}
    THEN
            act1 : Color(c) := E
    END

    ANColor
    ANY
            c
            X
            Y
    WHERE
            grd1 : Color[{X}] = {N}
            grd2 : Color[{Y}] = {A}
    THEN
    act1 : Color(c) := E
    END

    AEColor
    ANY
            c
            X
            Y
    WHERE
            grd1 : Color[{X}] = {E}
            grd2 : Color[{Y}] = {A}
    THEN
            act1 : Color(c) := N
    END

    ENColor
    ANY
            c
            X
            Y
    WHERE
            grd1 : Color[{X}] = {N}
            grd2 : Color[{Y}] = {E}
    THEN
            act1 : Color(c) := A
    END

END
```

3.4 Check

The verification of the algorithm can only detect syntax errors. It is the user to pay attention to names and labels they use values (replace int by a string). When an error is detected, the line is displayed for easier tracking.

3.5 Translate

When you launch the translation, you need three configuration steps to transform the event-B code into ViSiDiA code. The first step (figure 3.1) is to choose the type of synchronization required, if specified in the algorithm does not need to reselect. The "broken symmetry" option allows you to run LC0 algorithms need to differentiate the two neighbors nodes (such as the election in a tree).

Figure 3.1: Choose synchronization

It is possible to define in the second stage the initial state of the nodes, see figure 3.2. Each type of label taking a different syntax, the boolean is true or false as value and the *int* takes a mathematical formula with the possibilities of using keywords *id* and *card*.

Figure 3.2: Instantiate Label

In the end, you must choose the location of a file will be created and you can give a small description of your algorithm. See figure 3.3.

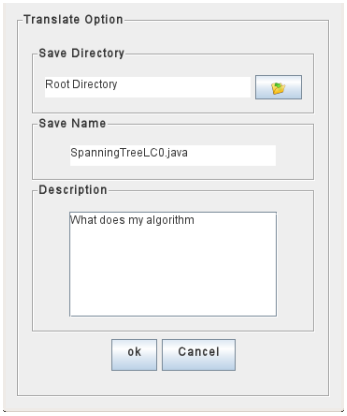

Figure 3.3: File Chooser

The transformation produces two files, one with java extension and the other in ".class". If there were no errors in compiling, the result is displayed in a new tab. Examples of possible errors while compiling the code has passed the ckeck:

- **Error in the type of the variable:** compares or instantiates a label with a type other than state.
  Display console:
  ERROR
  cannot find symbol
  symbol : constructor String(int)
  location: class java.lang.String

- **Error in the use of nodes:** Use a node not tested in the *WHERE* clause
  Display console:
  ERROR
  cannot find symbol
  symbol : variable door_Y_0
  location: class Algo

- **Error in updating a neighbor node label:** Update the neighboring nodes, so that one is in sync LC1
  Display console:
  ERROR
  cannot find symbol
  symbol : variable neighboursLabel_copy
  location: class Algo

3.6 Run ViSiDiA

The documentation available on the website of the project ViSiDiA can help you to run ViSiDiA. To load an algorithm must use the "add new" option in the window scheduled for the algorithms and go to the folder where the recorded files generated by our tool B2ViSiDiA.

Appendix A

Install Java

You must have either a Java Runtime Environment (JRE) or a Java Development Kit (JDK) in version 6 installed on your computer. A JRE is enough to run ViSiDiA. If you plan to develop algorithms, please install a JDK.
If you do not have a JRE/JDK, of if its version is prior to version 6, please download and install JRE/JDK 6.

You can get the last official Sun JRE/JDK :

`http://java.sun.com/javase/downloads/index.jsp`

or get an open source JDK, such as OpenJDK :

`http://openjdk.java.net/`

Under linux Ubuntu, you can for example get openjdk-6:

```
$ sudo apt-get install openjdk-6-jdk
```

Important note: Depending on your operating system, you may encounter problems with some open source JRE/JDK. Please refer to section A.1.
You may need to adjust your JAVA_HOME environment variable, for example:

```
$ export JAVA\_HOME=/usr/lib/jvm/java-6-openjdk/
```

Under Windows, you may have to set the PATH environment variable to the JDK binaries.

A.1 Troubleshooting

A.1.1 GNU Java Compiler (GJC)

Please check that you DO NOT USE a JDK from the GJC. To check, run on command line:

```
$ java -version
$ javac -version.
```

If your system is configured to use GJC, please download and install another JDK, and use it as default.

A.1.2 OpenJDK

You may encounter problems using OpenJDK with Fedora distribution: Java does not execute, or ViSiDiA graphical interface is altered (buttons are misalignmed for example). In this case, please install and use the official Sun JRE/JDK.
Here is the recipe to install Sun JDK on Fedora (example is given for JDK 6 Update 16):

* Get the last JDK version on `http://java.sun.com/javase/downloads/index.jsp` Get the file with `.bin` extension, not the one with `.rpm.bin` extension.

* Intall the JDK (you need root privileges)

```
$ chmod a+x jdk-6u16-linux-i586.bin
$ mv jdk-6u16-linux-i586.bin /usr/lib/jvm/
$ cd /usr/lib/jvm
$ ./jdk-6u16-linux-i586.bin
$ alternatives --install /usr/bin/java java \
       /usr/lib/jvm/jdk1.6.0_16/bin/java 3
$ /usr/sbin/alternatives --config java
(select 3 and enter)
```

* install the Java plugin for firefox (you need root privileges)

```
$ usr/sbin/alternatives --install /usr/lib/mozilla/plugins/libjavaplugin.so \
       libjavaplugin.so \
       /usr/lib/jvm/jdk1.6.0_16/jre/plugin/i386/ns7/libjavaplugin_oji.so 2
$ /usr/sbin/alternatives --config libjavaplugin.so
(select 2 and enter)
```

Appendix B

B2Visidia language

<*invariant*> ::= <*ident*> '∈' <*type_fonction*>

<*type_fonction*> ::= 'ND' <*relational_set_op*> <*set_expression*> # <*type*>
| 'g' <*relational_set_op*> <*set_expression*>

<*relational_set_op*> ::= ↔ | ⇿ | ↔ | ⇻ | ⇸ | → | ⤔ | ↣ | ⇸ | → | ⤖

<*set_expression*> ::= 'Z' | 'N' | 'bool' | < *expression* >

<*fonction*> ::= 'ident' 'p_a_c' 'ident' 'p_a_c'
| 'ident' 'p_a_c' 'ident' '↦' 'ident' 'p_a_c'

<*condition*> ::= <*literal*>
| <*literal*> ∧ <*condition*>
| <*literal*> ∨ <*condition*>
| <*predicat*>

<*literal*> ::= <*atomic_predicat*>
| '¬' <*atomic_predicat*>

<*atomic_predicat*> ::= 'p_a_c' <*condition*> 'p_a_c'
| <*expression1*> <*relop*> <*expression*>

<*expression1*>::= <*fonction*>
| 'card' 'p_a_c' 'g' 'p_a_c' 'ident' 'p_a_c'
| 'ID' 'p_a_c' 'ident' 'p_a_c'

<*predicat*> ::= '∀' 'ident_list_comma' '·' <*ident_dec*>
'∧' <*condition*> '⇒' <*condition*>
| '∀' 'ident_list_comma' '·' <*ident_dec*>
'⇒' <*condition*>

<*ident_dec*>::= <*expression*> '∈' <*expression*>

| <expression> '∈' <expression> '∧' <ident_dec>

<ident_list_comma>::= 'ident'
| 'ident' ',' <ident_list_comma>

<relop> ::= '>' | '<' | '≤' | '≥' | '=' | '≠'

<action> ::= <fonction> ':=' <expression>
| 'ident' ':=' 'ident' '⩤' 'p_a_c' <nouvel_etat> 'p_a_c'
| 'ident' ':=' 'ident' '⩤' 'ident'

<nouvel_etat> ::= 'ident' '↦' <expression>
| 'ident' '↦' <expression> ',' <nouvel_etat> <expression> ::= <term>
| '-' <term>
| 'p_a_c' <expression> 'p_a_c'

<term> ::= <term1>
| <term1> '+' <term>
| <term1> '-' <term>

<term1> ::= <factor>
| <factor> '*' <term1>
| <factor> '/' <term1>
| <factor> 'mod' <term1>

<factor> ::= '⊤'
| '⊥'
| 'number'
| <commun>

<commun> ::='ident'
| <fonction>
| 'card' 'p_a_c' 'g' 'p_a_c' 'ident' 'p_a_c'
| 'ID' 'p_a_c' 'ident' 'p_a_c'
| 'g' 'p_a_c' 'ident' 'p_a_c'