Version 4.1.0

Date of diffusion : December 2012

Atelier B 4.1.0 is available in two versions:

- ***Community Edition***, usable by everyone without any restriction. This version is not maintained.
- ***Maintenance Edition***, acess restricted to Atelier B 4 maintenance contract holders (corrective maintenance, anticipated access to new features/tools). Some features are specific to this version (Ada, HIA and C++ code generators, mathematical rules proof tool.

Functionality	Atelier B 4.1 Community Edition	Atelier B 4.1 Maintenance Edition
Integrated Development Environment	✓	✓
Support of B Language Project	✓	✓
Support of Event-B Language Project	✓	✓
Support of Data Validation Project	✓	✓
Editor of B and Event-B Models	✓	✓
Automatic Refiner	✓	✓
Type Checker	✓	✓
Proof Obligations Generator	✓	✓
Automatic Prover	✓	✓
Interactive Prover	✓	✓
Predicate Prover	✓	✓
C Translator C4B	✓	✓
Ada Translator (MacOS, Linux)		✓
High Integrity Ada Translator (MacOS, Linux)		✓
C++ Translator (MacOS, Linux)		✓
Mathematical Rule Validator Tool		✓

New Functionalities / Characteristics:

Atelier B 4.1.0 (Community Edition and Maintenance Edition) has been released on December 11, 2012. This release brings 385 bug corrections (37 for 4.0.1, 79 for 4.0.2, and 269 for 4.1.0) and 56 improvements (5 for 4.0.1, 13 for 4.0.2, and 38 for 4.1.0).

Among these features, we can mention:

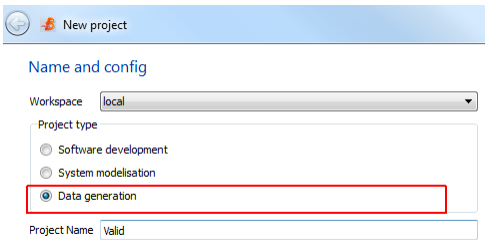
- Support for « data validation » project
- Improved support of integers : maximum implementable value (MAXINT) can be 16, 32 or 64 bit-wide, integer literals are not bounded (BigINT)
- Experimental support of real numbers
- Experimental support of floating point numbers
- Support of hexadecimal literals.
- New C code generator, C4B, with translation profile and makefile creation.
- Localization of the software in English, French, Japanese, and Portuguese.
- Support of Unicode. Identifiers, strings and comments may contain non-ASCII characters.
- Addition of a components status graphical view
- Parallel execution of proof tasks: firing a proof action several times leads to parallel execution of these tasks. Additional information, such as the order of execution of these actions, are displayed
- Improved Integrated Editor: Ctrl-D to delete a line, better indentation, addition of a list of open files, tuneable syntactic highlighting, model navigation (jump to definition, abstraction, or refinement).
- The interactive prover displays pending goals. Remaining proof work may now be evaluated with more precision.
- Generation of coverage and exclusivity proof obligations: these proof obligations allow to demonstrate the coverage of a system model, and to check that at most one event is enabled in each state of the system.
- Tool for validating user added mathematical rules: this tool provides a unique interface for managing and validating mathematical rules. Moreover manual demonstrations may be typed in which will be saved as comments in the pmm file. The tool provides a view per project and per component, and is also able to generate a validation report.
- Spell-checking B model comments: incorrectly spelled words are identified in the B model editor. A contextual menu allows choosing a correction among several proposals. Pre-installed languages are English (GB, US) and French (France, Belgium, and Canada).
- Mathematical rules displayed in the view “Theory List” are sorted according to their applicability (holding guards are written in bold).
- B model editor informs the user when the edited file has been modified on the disk.
- Addition of a search function “à la grep” enabling the searching of strings among all models of a project, with the help of regular expressions.
- Added rules when using the command “pc” are now displayed.

Data Validation/Generation[1]

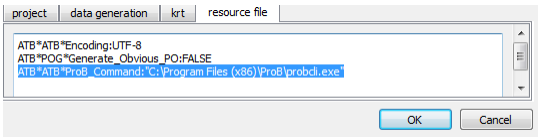
Data validation consists in verifying that it is possible to value a set of constants complying with a set of properties[2]. This validation is based on the model checker ProB that is now interfaced with Atelier B 4.1.

To use this new feature, it is mandatory to:

- install ProB[3]
- create a « data generation » project



- add a resource to this project, in order to be able to use ProB for validating data



Data generation may be applied to any machine containing only constants (or context machine) by using the « Generate Data » command.

If data generation is successful, results are store in a file in the directory <project translate directory>.

Project	Component	Action	Status	Messages	Server
P1	M0		Finished	Data generation for component M0 has successfully terminated	localhost

The file <machine>.probcst contains computed values for constants.

For example, for the machine:

```
MACHINE
    M0

CONSTANTS
    C0
PROPERTIES
    C0 : INT &
    C0: 0..4
END
```

All possible values of C0 are listed:

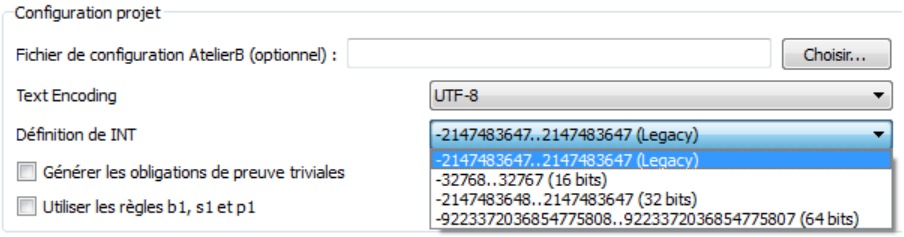
```
values(constants,[bind('C0',int(0))]).
values(constants,[bind('C0',int(1))]).
values(constants,[bind('C0',int(2))]).
values(constants,[bind('C0',int(3))]).
```

[1] This work has been funded by Alstom Transport Information Solutions – contract 04-4550001418.
[2] http://www.methode-b.com/2012/11/data-validationgeneration/?lang=en
[3] http://www.stups.uni-duesseldorf.de/ProB/index.php5/The_ProB_Animator_and_Model_Checker

Integer Improved Support

- Parameterisation of INT

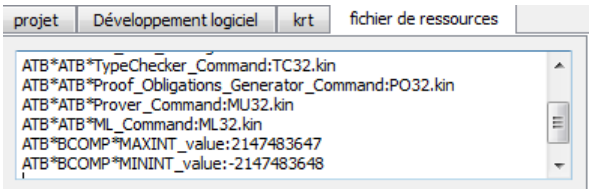
When creating a project, the size of implementable integers is tuneable (16, 32 or 64 bits).



The set of implementable integers, INT, is defined by INT = MININT..MAXINT. The value of the predefined constants MININT and MAXINT depends on this choice.

This choice is persisted in the AtelierB resources file, through the following resources:

- Binary filenames (*.kin) for typechecker, proof obligation generator and provers. File names depends on the number of bits used for integer (for example, typechecker binary file has three versions: TC16.kin, TC32.kin, and TC64.kin).
- MININT_value and MAXINT_value values.



To modify this parameterisation of a project once it has been created, it is mandatory to:

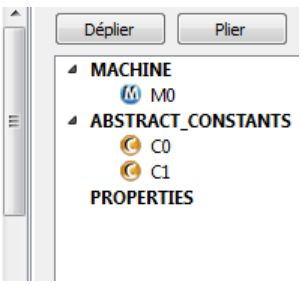
- modify the name of the four binary files
- modify the values of MININT_value and MAXINT_value,

with respect to the number of bits for implementable integers. It is also required to regenerate all proof obligations because of the different values of MININT and MAXINT.

- Big integer support (BigInt)

Integer literals are not limited to INT. Now they can be used in modelling and proof without any size restriction, thanks to the library BigInt::GMP.

```
1- MACHINE
2      M0
3
4- ABSTRACT_CONSTANTS
5      C0, C1
6- PROPERTIES
7      C0 : INTEGER &
8      C1 : INTEGER &
9      C0 + C1 <= 12345678901234567890
10 END
```



Real Numbers Experimental Support[4]

REAL type has been introduced in the B language, with the goal of eventually being able to specify algorithms manipulating real data and implemented using floating point numbers.

```
LIBPTF_types_init_data =
BEGIN
    V_ptf_data := rec ( pure_inertial_attitudes : rec ( Roll : 0.0 , Pitch : 0.0 , Azimuth : 0.0 ) ,
        T_vm : % xx . ( xx : 0 .. 2 | ( 0 .. 2 ) * { 0.0 } ) ,
        T_vb : % xx . ( xx : 0 .. 2 | ( 0 .. 2 ) * { 0.0 } ) ,
        deltaV_v : ( 0 .. 2 ) * { 0.0 } ,
        q_vm : ( 0 .. 3 ) * { 0.0 } )
END ;
```

The REAL type is a basic type that complements BOOL, INTEGER and STRING.

```
PROPERTIES
    tFloat32 = REAL &
    tFloat64 = REAL &

    tVect3_32 = 0 .. 2 --> tFloat32 &
    tVect3_64 = 0 .. 2 --> tFloat64 &

    tMat33_32 = ( 0 .. 2 ) --> ( ( 0 .. 2 ) --> tFloat32 ) &
    tMat33_64 = ( 0 .. 2 ) --> ( ( 0 .. 2 ) --> tFloat64 ) &

    tQuater_64 = 0 .. 3 --> tFloat64 &

    tPosHorizontal_64 = struct (
        Latitude : tFloat64 ,
        Longitude : tFloat64
    ) &

    tPosition_64 = struct (
        Coord : tPosHorizontal_64 ,
        Altitude : tFloat64
    ) &
```

Arithmetic operators +, -, * are kept as the properties of these operators are the same for integers and reals. The real division has different semantics and receives a specific operator: RDIV.
The comparison operators <=,> = are preserved.
Real literals, which must be syntactically distinguishable from integer literals to ensure type control, are postfixed with a ".". For example, 1.0 is the real number mathematically equal to the integer number 1.

```
PROPERTIES

    C_EARTH_RADIUS      : tFloat64 & C_EARTH_RADIUS      = 6378137.0                          &
    EXCENTRICITE_P2     : tFloat32 & EXCENTRICITE_P2     = 0.006694379989875978287281         &
    DGA_PAR_UN_MOINS_E2 : tFloat32 & DGA_PAR_UN_MOINS_E2 = 6335439.32729451239747469642450 3 &
    OMEGA_EARTH         : tFloat32 & OMEGA_EARTH         = 0.000072921151467                  &

    GE : tFloat32 & GE = 9.7803267714    &
    K1 : tFloat32 & K1 = 0.005279041381  &
    A0 : tFloat32 & A0 = 0.0000003134913 &
    K2 : tFloat32 & K2 = 0.00000000209427079
```

New typing rules have been added:

- In predicates x <= y and x> = y, the expressions x and y must be of the same type (INTEGER or REAL);
- The real literals are of type REAL;
- In the expressions x + y, x - y, x * y, the expressions x and y must be of the same type (INTEGER or REAL);
- In the expression x RDIV y, the expressions x and y must be of type REAL;
- In the expression x ** y, the expression x must be of type INTEGER or REAL, and the expression y must be an expression of type INTEGER;

[4] This work has been funded by SAGEM Défense Sécurité – contract SK-0000443958-02.

- In the expressions x mod y, succ (x) and pred (x), the expressions x and y must be of type INTEGER;
- In expressions max (E), min (E), the expression E must be of type POW(INTEGER) or POW(REAL);
- SIGMA expressions in (x). (P | E) and PI (x). (P | E), the expression E must be of type INTEGER or REAL;
- In the expression x .. y, the expressions x and y must be of type INTEGER.

New well-definedness conditions have been added:
- The well-definedness of x RDIV y is y / = 0.0;
- The well-definedness of min (x) and max (x) is x: FIN (x).

Atelier B 4.1 supports type checking and proof obligations generation for REAL numbers.

⚠ REAL numbers proof support should be slightly improved to ensure usability.

Floating Point Numbers Experimental Support[5]

FLOAT type has been introduced in the B language, with the goal of eventually being able to specify algorithms manipulating data with floating point numbers, prove and generate the corresponding code in accordance with IEEE 754 standard.

```
res <-- main ( xx ) =
VAR
    i_f, ii, ff, lf
IN
    i_f := ( xx -. start ) /. step ;
    lf := float0 ;
    IF i_f >=. lf
    THEN
        lf <-- get_floati ( nn - 1 ) ;
        IF i_f <. lf
        THEN
            lf <-- get_floor ( i_f ) ;
            ii <-- get_intf ( lf ) ;
            lf <-- get_floati ( ii ) ;
            ff := i_f -. lf ;
            lf := float1 ;
            lf := lf -. ff ;
            lf := lf *. yy ( ii ) ;
            res := ff *. yy ( ii + 1 ) ;
            res := lf +. res
        ELSE
```

Adding floating point numbers requires larger changes of the B language. Indeed, these new objects have special properties that have nothing in common with those commonly used in the language. So we have to add specific operators that do not appear in the rule database so as to maintain its integrity without requiring a complete validation. Only equality can be preserved. Indeed, in the B language, equality is already overloaded for sets and integers, so we decided to expand to both real and floating point numbers.
We chose not to "encode" floating point numbers in the Atelier B but to keep them as "neutral objects" that will never calculated but whose properties are given by the rule database.
All usual arithmetic operators are suffixed with a ". ".
Four comparison operators are added: "<=. "," <. ","> =. ""> . ".
Float expressions are constructed with the operators "+.", "-.", "*.", "/.".
New operators are added: minf, maxf, SIGMAF and PIF, being floating counterparts of integer operators min, max, SIGMA, and PI.

We chose not to provide a means to write floating point literals. If necessary, it is always possible to define constants valued in basic machines. This choice stems from the fact that we do not want to "code" floating point numbers in Atelier B but only model their properties.

New typing rules have been added:

- In predicates x <=. Y, x <. Y, x> =. Y and x>. Y, expressions x and y must be FLOAT
- In the expressions x +. Y, x-.y -. X, x *. Y, x / .y the expressions x and y must be of type FLOAT, the resulting expression is of type FLOAT.
- In the expression x **. y, the expression x must be of type FLOAT and y expression must be of type INTEGER. The resulting expression is of type FLOAT.
- In expressions maxf(E) and minf(E), the expression E must be of type POW(FLOAT), the resulting expression is of type FLOAT.
- In expressions SIGMAF(x).(P | E) and PIF(x).(P | E), the expression E must be of type FLOAT, the resulting expression is of type FLOAT
- In the expression Realf (x), x must be of type FLOAT, the resulting expression is of type REAL.

New well-definedness conditions have been added:

- Well-definedness of a /. b is realf (b) / = 0.0 /: {PLUSZEROF, MOINSZEROF}.
- Well-definedness of a **. b is b: NATURAL.
- Well-definedness of minf (a) and maxf (a) is a: FIN (a).

Atelier B 4.1 supports floating point numbers type checking and proof obligations generation.

[5] This work has been funded by SAGEM Défense Sécurité – contract SK-0000443958-02.

⚠ As of today, there is no proof support for floating point proof obligations. The R&D project BWare[6] would probably solve this issue in the future.
⚠ As of today, there is no support for code generation with floating point numbers. Proof support has to be ensured first.

[6] http://bware.lri.fr/index.php/BWare_project

Hexadecimal Literals Support[7]

Hexadecimal literals are used in modelling and proof, using the usual prefix 0x followed by a radix-16 representation (0 .. 9, A..F or a..f). Hexadecimal literals are of type integer. Combined with the support of large integers, this improvement can handle large hexadecimal literals.

```
1- MACHINE
2      ctx
3- CONSTANTS
4      BASE_MEMORY,
5      MASK1
6- PROPERTIES
7      BASE_MEMORY <: NAT &
8      MASK1 : NAT &
9      BASE_MEMORY = 0x1FF006FF .. 0x1FF02000 &
10     MASK1 = 0x1E
11 END
12
```

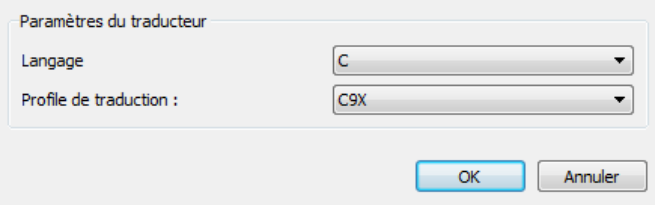
[7] This work has been funded by ST Microelectronics – grant 2008_0784.

New C Code Generator, C4B, with Translation Profiles and Makefile Generation

A new code generator, C4B, replacing ComenC, was developed based on the B compiler. He does not yet support the renaming of parameters and abstract machine language but supported B0 is larger than ComenC. Some known limitations are listed below.

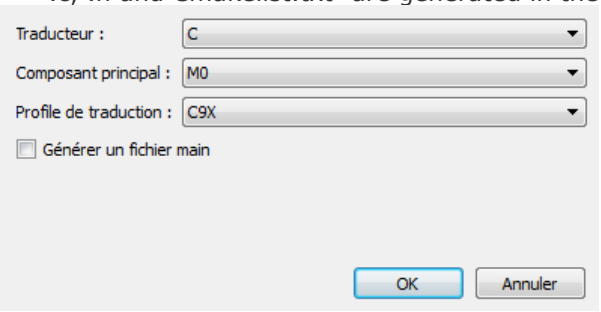
C4B may be used:

- In mode « component », to translate in C an implementation. One translation profile has to been selected among C9X, LIGHT, and PROJECT. .c and .h files are generated in the directory <project translate directory>/c.

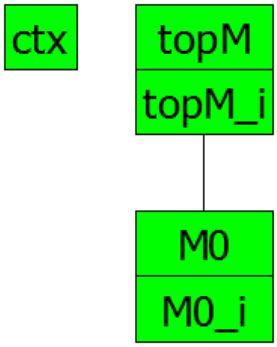


- In mode « project », to translate in C the whole project. The following information are required:
  - The name of the top level component, to choose among the implementations of the project. This implementation should contain only one parameter-less operation.
  - A translation profile: C9X, LIGHT or PROJECT.
  - Either a cmakefile have to be generated or not, in order to ease the compilation of the project.

  .c, .h and Cmakelist.txt[8] are generated in the directory <project translate directory>/c.



Example: for the project below, the following file Cmakelist.txt is produced:




```
project(P1)

cmake_minimum_required(VERSION 2.4)

SET(P1_SOURCES
    M0_i.c
    topM_i.c
    b_init.c
    b_main.c
)
SET(P1_HEADERS
    M0.h
    ctx.h
    topM.h
    b_init.h
)
add_executable(P1 ${P1_SOURCES} ${P1_HEADERS})
```

Component b_init.c initializes the components of the project, in a correct order.

[8] cmake (http://www.cmake.org) allows to automate project compilation with a definition file cmakelist.txt.

Component b_main is the top-level component: it triggers components initialization (b_init.c) then treatments contained in the single operation of the component topM_i.

Translation profiles allow tuning the translation of some elements:

Element	Profile C9X	Profile Light	Profile Project
boolean type	bool	unsigned char	user defined
boolean true literal	true	1	user defined
boolean false literal	false	0	user defined
integer type	int32_t	long	user defined
added headers	<stdint.h> <stdbool.h>	/	user defined

Profile « Project » allows choosing more precisely how to translate each element, through the use of resources set in the AtelierB file.

⚠ Version 4.1.0 includes profile "C9X" only.

Know limitations are related to arrays:

- Constants and variables of type "array of T" must be defined with the type:
  (0..MAX) --> T

where MAX is an integer literal or a concrete constant valued with an integer.

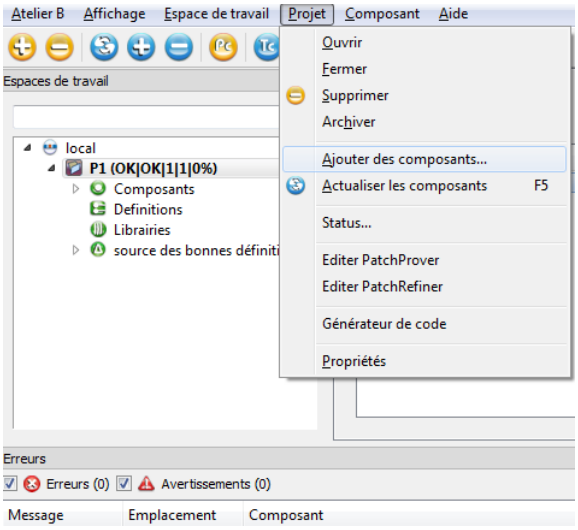
```
1- IMPLEMENTATION
2     M0_i
3- REFINES
4     M0
5- CONSTANTS
6     C0
7- PROPERTIES
8      C0: NAT
9- VALUES
10     C0 = 12
11- CONCRETE_VARIABLES
12     T1
13- INVARIANT
14     T1: (0..C0) --> INT
15- INITIALISATION
16     T1 := (0..C0) * {0xFF}
17+ OPERATIONS
20 END
21
```

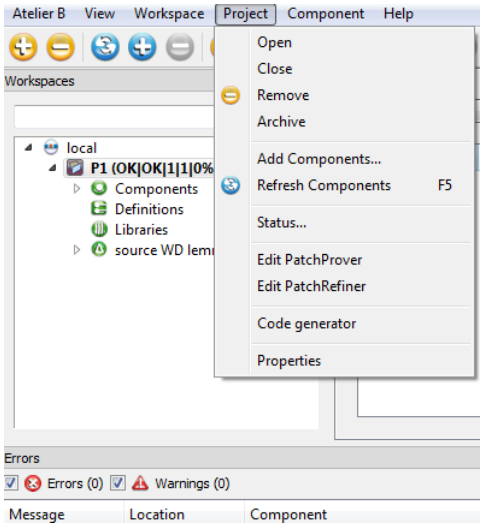
- Arrays of dimension 2 or greater are not supported.
- Concrete constants defining an array type are not supported (example: TT = 0..MM --> BOOL in the clause PROPERTIES where TT concrete constant).
- Constant arrays are not declared or initialized properly when arrays are of type SS --> TT where SS is an abstract set.

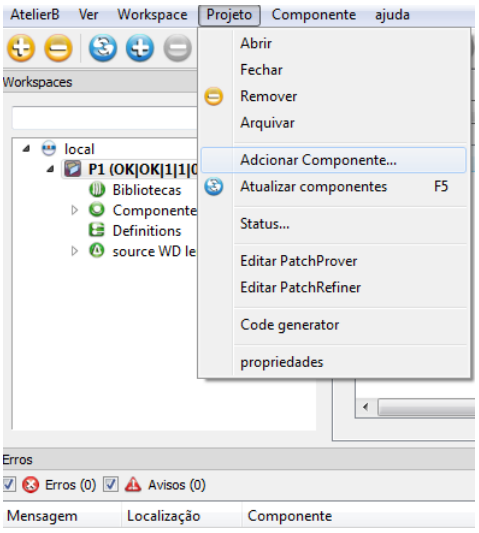
Localization of Graphical Interface in English, French, Japanese[9] and Portuguese[10]

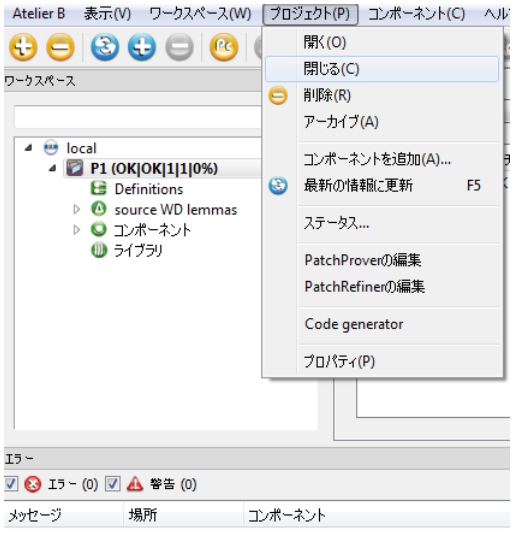
Language used by Atelier B graphical interface depends on the localization of the computer executing it.
To display another language (restarting Atelier B is mandatory):

- Select a different language in the menu « Preferences/main window/language »
- Modify the LANG variable (fr, en, br, ja) before starting Atelier B.





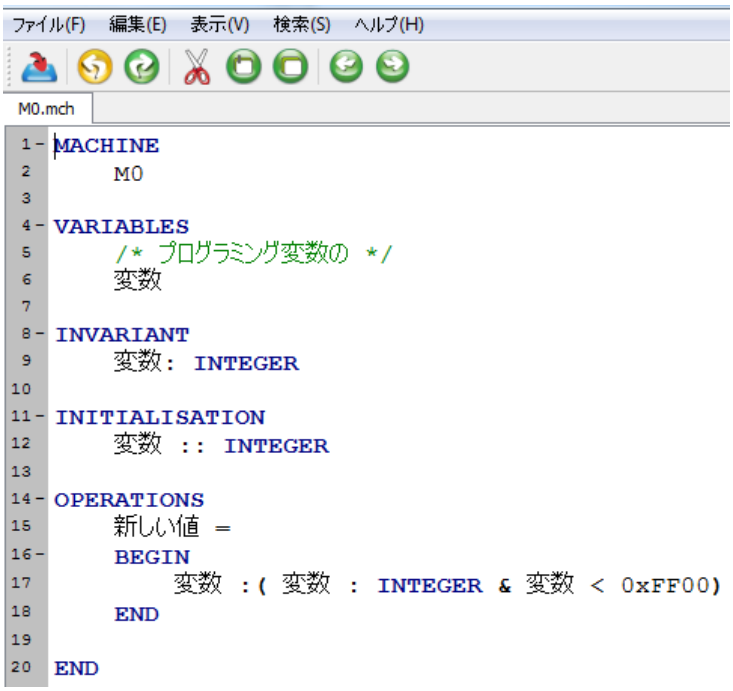




[9] Localization in Japanese has been completed by Takuya Sawada (Hokkaido Electronics Corporation, Sapporo).
[10] Localization in Portuguese has been completed by Aryldo Russo Jr (Aes, Sao Paulo) and Haniel Herbosa (UFRN, Natal).

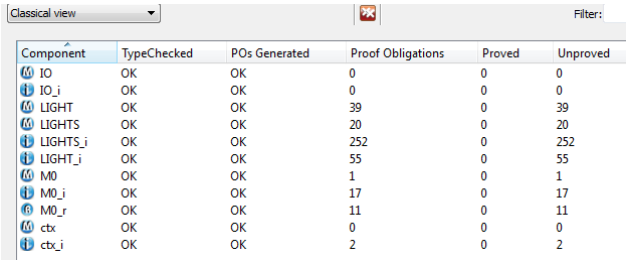
Unicode Support

Identifiers, strings and comments may contain non-ASCII characters.



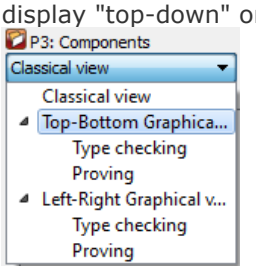
New Graphical Views for Components View

Atelier B components view lists the various components of a project, their status (typechecked, proof obligations generated) and the number of proof obligations (proven and unproven).

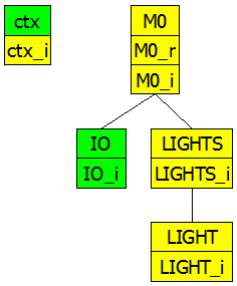


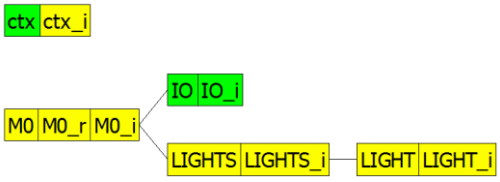
Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved
IO	OK	OK	0	0	0
IO_i	OK	OK	0	0	0
LIGHT	OK	OK	39	0	39
LIGHTS	OK	OK	20	0	20
LIGHTS_i	OK	OK	252	0	252
LIGHT_i	OK	OK	55	0	55
M0	OK	OK	1	0	1
M0_i	OK	OK	17	0	17
M0_r	OK	OK	11	0	11
ctx	OK	OK	0	0	0
ctx_i	OK	OK	2	0	2

Several graphical views have been added in order to better estimate the current status of a project. View can be selected through the menu displayed below. Note that the view can be set to display "top-down" or "left to right".

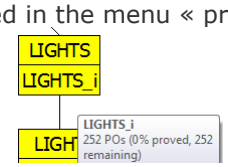


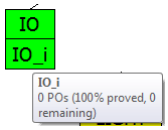
Simple graphical view displays the proof status of a project. Components are sorted according to their dependencies.



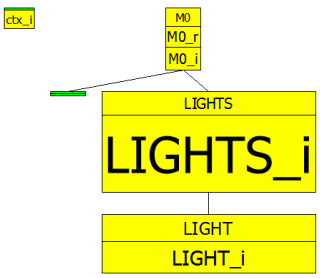


Colour indicates proof status: green for 100% proven, yellow for lower proof rate (these colours can be modified in the menu « preferences »).



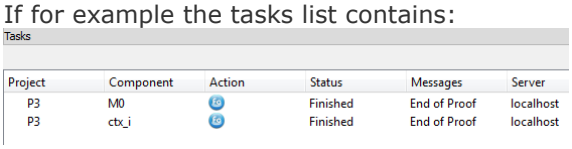


With the views « typecheck » and « proving », the size of the boxes depends on the number of proof obligations of the related component.

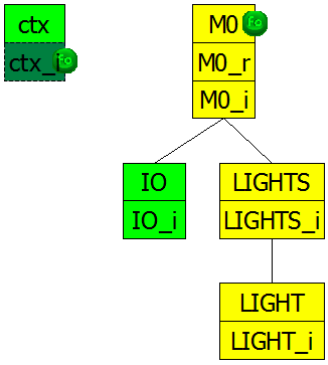


The graphical view finally allows displaying tasks associated to components.

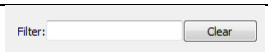
If for example the tasks list contains:



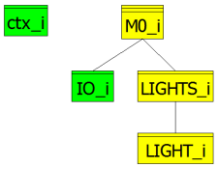
then the graphical view shows up:



Components can be filtered according to their name. If their name contains the characters typed in the field, selected component will be magnified (boxes are displayed with a bigger size).



Example :
When typing « _i », implementations are magnified.



Some widgets allow to:

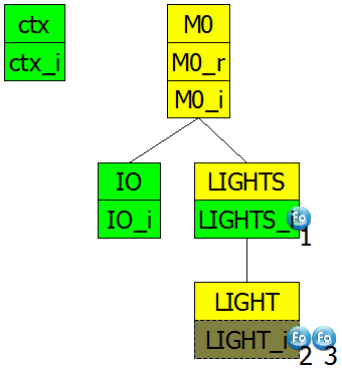
- remove tasks successfully completed,
- zoom on the whole project,
- zoom on filtered components,
- zoom on selected component and its children,
- select a set of components,
- print the graphical view,
- export the current graph as a graphviz file.

Executing Proof Tasks in Parallel

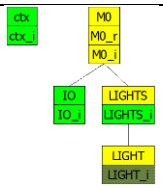
It is now possible to start several proof tasks in parallel on a single component.
You have to:

- modify the resource « maximum running tasks » with a value greater than 1 (menu *preferences* / *install*) ;
- select one or several components and use several times the buttons F0 or F1.

The sequence of initiated tasks is displayed in the tasks list textual window. The graphical view is updated accordingly. When more tasks are activated than the « maximum number of running tasks », some tasks are postponed and are granted an order number which appear on the graphical view.



These numbers will decrease when active tasks complete.

	If « Hide Finished Tasks » is ticked, only active tasks are displayed.	
	If not, history is displayed (past, current and pending tasks).