Version 4.2

Date of diffusion : December 2014

Atelier B 4.2 is available in two versions:

- ***Community Edition***, usable by everyone without any restriction. This version is not maintained.
- ***Maintenance Edition***, access 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.

Fonctionality	Atelier B 4.2 Community Edition	Atelier B 4.2 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 4.2 (Community Edition and Maintenance Edition) has been released on December 19, 2014. This release brings 151 bug corrections and 47 improvements.

Among these features, we can mention:

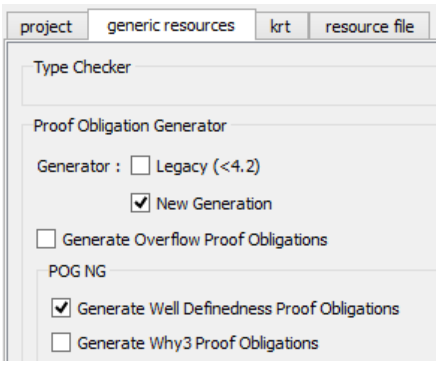
- Full 64-bit support
- A new traceable, generic, proof obligation generator
- A better integration of real and floating point numbers
- Accessors added to BART in order to solve refinement conflicts
- Boolean and integer translation fine-tuning in C4B code generator
- Addition of a proof server, in order to speed up proof

Generic Proof Obligation Generator

A new proof obligation generator (POG) has been developed[1] in order to bring new functionalities:

- Proof obligation traceability, through an integrated GUI associating model and proof;
- Simplified proof obligations par by modifying formulas normalisation principles;
- Ability to define and add your own proof obligations, through xsl files defining theoretical proof obligations for B method, Event-B and well-definedness.

This new proof obligation generator produces about the same number of proof obligations, while their content may vary. It is highly probable that a project proved with a previous version of Atelier B would not be completely proved (automatically or replaying saved demonstrations) with Atelier B 4.2. The new proof obligation generator is selected by default when creating a project (*New Generation*). To use the previous proof obligation generator, it is required to change project configuration and to select « *Legacy (<4.2)* ». From this configuration window, it is also possible to trigger the generation of arithmetic overflow, well-definedness or Why3[2] proof obligations.



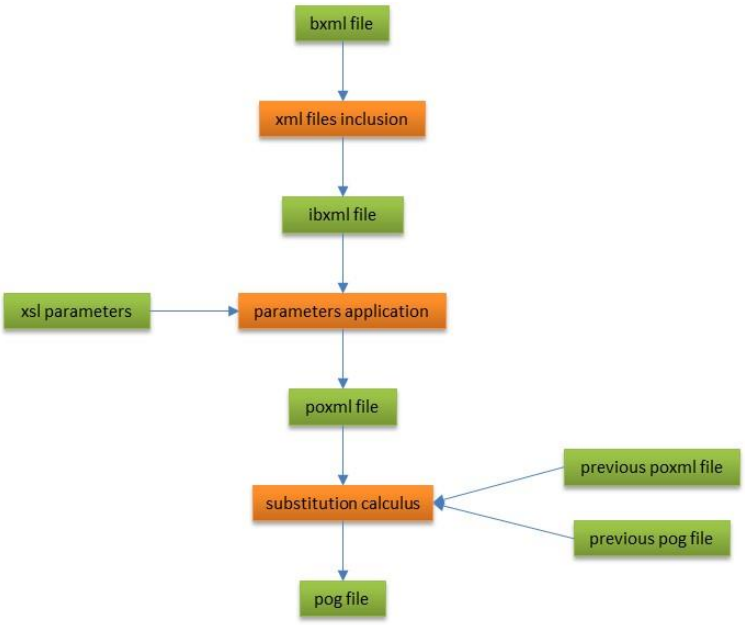


Figure 1: files used by Atelier B and their relationships

Files used internally by Atelier B 4.2 have changed (see Figure 1), mostly all in xml formal. B models are saved in bxml format, proof obligations are making use of poxml format to ensure differential generation: only proof obligations related to modified parts of models are recomputed, non-modified proof obligations (and their associated demonstrations are kept).

[1] Within the framework of project Cercles-2 with the support of Agence Nationale pour la Recherche

[2] Resulting file is named like the component from where it is used but with .why extension. It is located in the bdp directory of the project.

Traceability

Proof obligations (PO) are now linked with the models they are issued from. For each PO, in the interactive prover, related models are displayed on the pane right to the initial proof obligation (see Figure 2), one section per model part (invariant, variable, operation clauses for example). Expressions contained in the proof obligations are linked with model code: when selecting expressions in the initial proof obligation, corresponding expressions in the model are selected (inverse video).

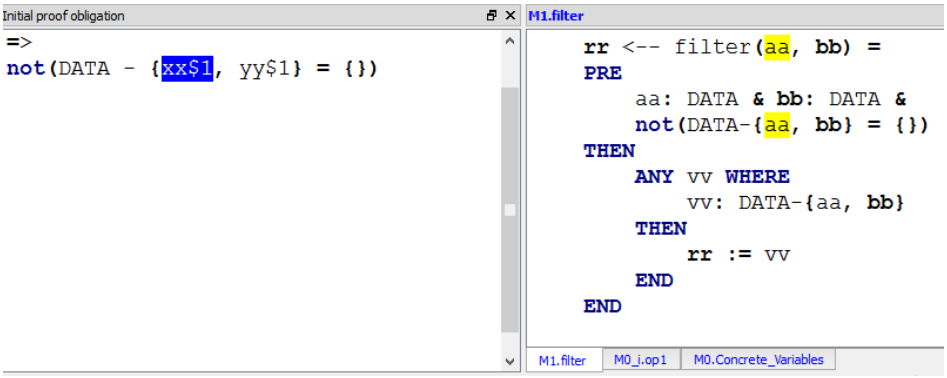
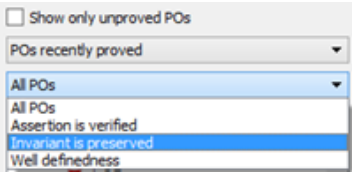


Figure 2: proof obligation with traceability information

PO can be filtered according to their type in the interactive prover GUI, by using a dynamic contextual menu which contains only the types of the POs of the current component.



Versatility

Theoretical proof obligations are POG parameters. They are located in Atelier B installation directory, in the sub-directory press/include:

- *paramGOPSoftware.xsl* : proof obligations for B method
- *paramGOPSystem.xsl* : proof obligations for Event-B
- *wellDefinedness.xsl* : proof obligations for well-definedness

These proof obligations may be extended by modifying these files.

```
514     <Proof_Obligation>
515       <Tag>WellDefinednessProperties</Tag>
516       <Definition name="B definitions"/>
517       <Definition name="ctx"/>
518       <Definition name="mchcst"/>
519       <Definition name="aprp"/>
520       <Hypothesis><xsl:copy-of select="Sets/*"/></Hypothesis>
521       <Goal>
522         <Well_Definedness>
523           <xsl:copy-of select="Properties/*"/>
524         </Well_Definedness>
525       </Goal>
526     </Proof_Obligation>
```

Figure 3: proof obligation of the well-definedness of constant properties of a machine

Otherwise in the directory, the file *bxml.xsd*, contains the definition of the grammar of bxml, new file format for B models, enabling interoperability.

Simplicity

New normalisation principles have been defined and are listed in the three following tables.

Goal and hypothesis normalisation	Normalised predicate (expression)
a /= b	not(a = b)
a /: b	not(a : b)
a <: b	a : POW(b)
a <<: b	a : POW(b) & not(a=b)
a /<: b	not(a : POW(b))
a /<<: b	a : POW(b) => a=b
a <= b (real)	a rle b
a <= b (float)	a <=. B
a >= b (int)	b <= a
a >= b (real)	b rle a
a >= b (float)	b <=. a
a < b (int)	a+1 <= b
a < b (real)	a rle b & not(a=b)
a < b (float)	a <=. b & not(a=b)
a > b (int)	b+1 <= a
a > b (real)	b rle a & not(b=a)
a > b (float)	b <=. a & not(b=a)
a + b (real)	a rplus b
a + b (float)	a +. b
a - b (real)	a rminus b
a - b (float)	a -. b
a * b (real)	a rmul b
a * b (float)	a *. b
a / b (real)	a rdiv b
a / b (float)	a /. b
a ** b (real)	a rpow b
-a (real)	0.0 rminus a
max(a) (real)	rmax(a)
min(a) (real)	rmin(a)
SIGMA(a).(b|c) (real)	rSIGMA(a).(b|c)
PI(a).(b|c) (real)	rPI(a).(b|c)
{a|b}	SET(a).(b)
a <=> b	(a => b) & (b => a)
bool(a) = TRUE	A
NAT1	NAT-{0}
NATURAL1	NATURAL-{0}
[]	{}
{a_1,...,a_n}	{a_1}\/...\/{a_n}
FIN1(a)	FIN(a)-{{}}
POW1(a)	POW(a)-{{}}

Table 1: Predicate and expression normalisation in the goal and hypotheses (predicates and expressions are replaced by their normalised form)

Valuation normalisation (when POs are generated)	Normalised predicate
a(b) := c	a := a <+ {b |-> c}
a'b := c	a := a <<< {b$8888 = c}

Table 2: normalised valuation (predicates replaced by their normalised form)

Hypothesis normalisation (new hypotheses added)	Normalised predicate
a : NATURAL	a : INTEGER & 0 <= a
a : b --> c	a : b +-> c & dom(a)=b
a : b >+> c	a : b +-> c & a~ : c +-> b
a : b >-> c	a : b +-> c & a~ : c +-> b & a : b --> c & dom(a)=b
a : b +->> c	a : b +-> c & ran(a) = b
a : b -->> c	a : b +-> c & ran(a) = b & a : b --> c & dom(a)=b & a : b +->> c
a : b >+>> c	a : b +-> c & ran(a) = b & a~ : c +-> b & a : b >+> c & a : b +->> c
a : b >->> c	a : b +-> c & ran(a) = b & a~ : c +-> b & a : b --> c & a : b >+> c & a : b +->> c
a : seq(b)	a : NATURAL-{0} +-> b
a : seq1(b)	a : seq(b) & a : NATURAL-{0} +-> b & not(a={})
a : iseq(b)	a : seq(b) & a : NATURAL-{0} +-> b & a~ : b +-> NATURAL-{0}
a : iseq1(b)	a : seq(b) & a : NATURAL-{0} +-> b & a~ : b +-> NATURAL-{0} & a : iseq(b) & a : seq1(b) & not(a={})
a : perm(b)	a : seq(b) & a : NATURAL-{0} +-> b & a~ : b +-> NATURAL-{0} & a : iseq(b) & a : seq1(b) & not(a={}) & ran(a) = b

Table 3: normalised predicate in hypothesis (new hypotheses are created)

A new, internal, update operator has been added[3] that allows for avoiding situations where record modification could lead to memory exhaustion and large expressions difficult to manipulate. All known bugs related to records are solved, including name captures between record labels and model variables.

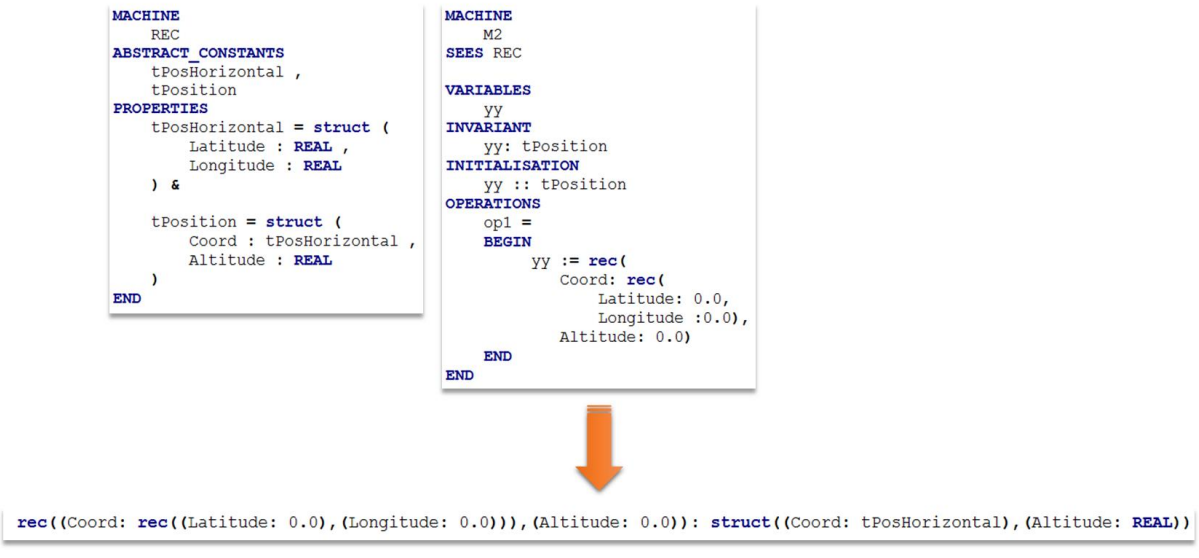


Figure 4: example of proof obligation issued from "record in a record" valuation – all labels have to be made explicit

This proof obligation generator has not yet been qualified for a safety critical software development. Caution is required in case of a SIL3 or SIL4 software development.

[3] Comparable to *with (field update)* of the why3 language)

Real and floating point numbers

Since release 4.1, real and floating point numbers are supported (see Release Notes 4.1.0). Real numbers are of type **REAL**, floating point numbers are **FLOAT**. With release 4.2, real and floating point numbers support has been improved and slightly modified.

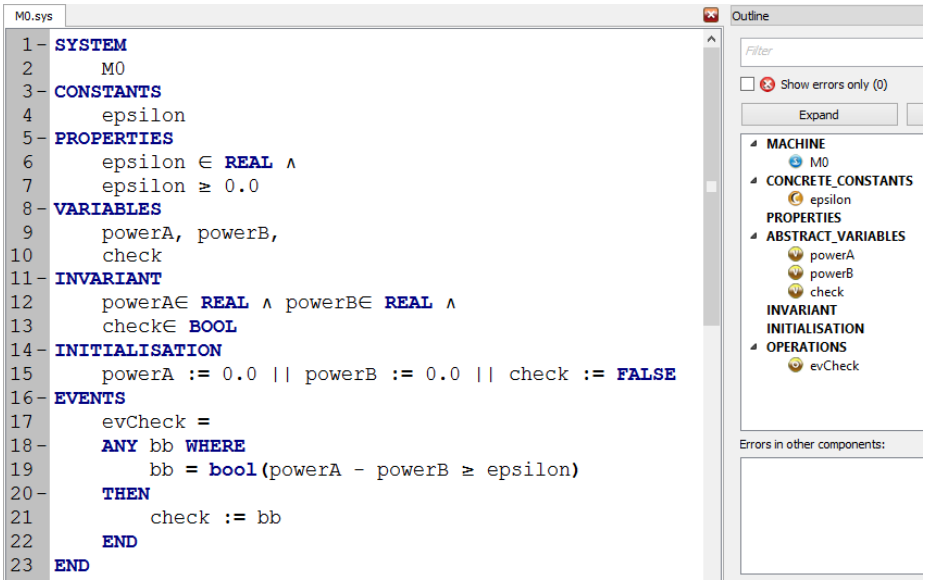


Figure 5: Event-B model including real variables

There are several important points to mention:

- non integer numbers are only taken into account by the **new** proof obligation generator;
- Unlike release 4.1, arithmetic operators are unified among integer, real and floating operators (see Table 4, Table 5 and Table 6);
- On the other side, during proof phase, operators are different because their semantic is different. Hence languages in the models and in the prover differ (this apply also for mathematical rules in the pmm files).

When proof obligations are generated, unified syntax is transformed into type-specific dedicated syntax. Operand type is used to determine the operator to use. There is no conversion nor no implicit coercion.

Unified	Integer	Real	Float
x <= y	x <= y	x rle y	x <=. y
x < y	x < y	x rlt y	x <. y
x >= y	x >= y	x rge y	x >=. y
x > y	x > y	x rgt y	x >. y
x + y	x + y	x rplus y	x +. Y
- x	- x	0.0 rminus x	-. X
x - y	x - y	x rminus y	x <=. y
x * y	x * y	x rmul y	x *. Y
x / y	x / y	x rdiv y	x /. y
x ** y	x ** y	x rpow y	***Invalid***
min(x)	min(x)	rmin(x)	***Invalid***
max(x)	max(x)	rmax(x)	***Invalid***
SIGMA(x).(y | z)	SIGMA(x).(y | z)	rSIGMA(x).(y | z)	***Invalid***
PI(x).(y | z)	PI(x).(y | z)	rPI(x).(y | z)	***Invalid***

Table 4: predicate conversion, from model (left column) to proof, based on their type

Well-definedness conditions are equivalent to integer operators ones.

Integer/real and real/integer conversion is performed by using operators in the next table.

Meaning resulting type	Syntax	Operand type
Embedding integers in reals **real(x) : REAL**	real(x)	x : INTEGER
Integer part **floor(x) : INTEGER**	floor(x)	x : REAL
Smallest following integer **ceiling(x) : INTEGER**	ceiling(x)	x : REAL

Table 5: integer/real and real/integer conversion operators

Floating point numbers are considered as implementable type and as such they do not have specification operators like `min`, `max`, `SIGMA` and `PI`.

Floating point literals are not accepted: a basic machine has to be used instead.

There is no predefined operator for converting float into real and float into integer (and vice-versa).

Real and float partial order normalisation (used in the proof obligations) is summarized on the following table.

Unified	Integer	Real	Float
x < y	`x+1 <= y`	`x rle y & x /= y`	`x <=. y & x /= y`
x >= y	`y <= x`	`y rle x`	`y <=. x`
x > y	`y+1 < x`	`y rle x & y /= x`	`y <=. x & y /= x`

Table 6: partial order normalisation for integers, reals and floats

BART

- ***Addition of accessors to solve variable implementation conflict***

  Classical refinement substitution rules (THEORY_OPERATION) may require the implementation or export of variables. Export of variables means here that the implementation of a variable is transferred to an imported machine. Resulting constraints on how variables have to be implemented in a refinement column may lead to conflicts when constraints are incompatible (a variable needs to be both implemented and exported), preventing the refinement process to complete successfully.

  A solution to avoid conflicts is to use a special type of substitution rule for each rule requiring the implementation of a variable: accessor rules (THEORY_ACCESSOR). These rules are elementary rules describing the refinement of a substitution requiring a variable to be implemented.

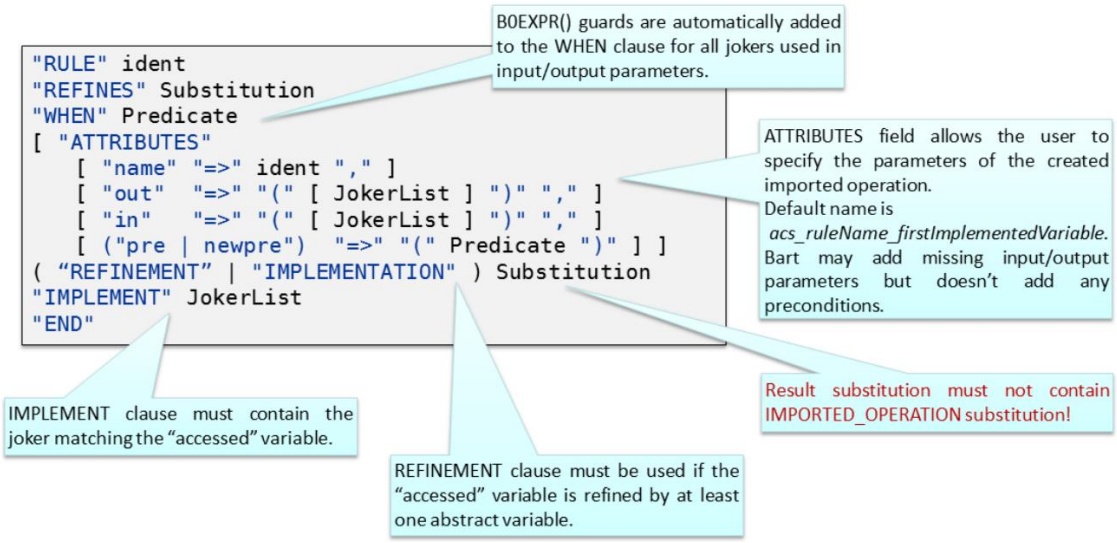


Figure 6: syntax of an accessor rule

In practice, this refinement is automatically imported in an "accessor" rule, which contributes to avoid conflicts when allocating operations in the refinement column. In short, the theory THEORY_OPERATION should not contain any rule requiring the implementation of a variable; similarly the theory THEORY_ACCESSOR should only contain rules implementing variables in order to read or modify them. In the operation rules, transformations of variables should be performed using abstract substitutions (the ones that are refined by accessor rules). Please refer to BART User Manual for more details about accessor rules.

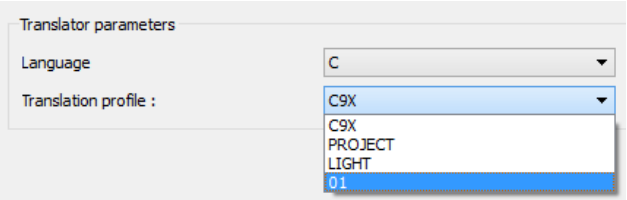
- Components generated automatically appear differently in the project status: their names are in italic.

- BART generated models are not any more systematically suppressed then re-added to the project: only new components are added and suppressed components are removed. This way proof status, demonstrations and proof rules are not lost. For unchanged models, typecheck status and proof obligations are kept.

Fine-tuning Boolean and integer type translation for C4B

Since release 4.1, C4B C code generator has superseded ComenC (see Release Notes 4.1.0).

A new code generation mode has been added: "01". It allows to generate C code complying with the C9X profile with constants and variables names not prefixed with the component name.



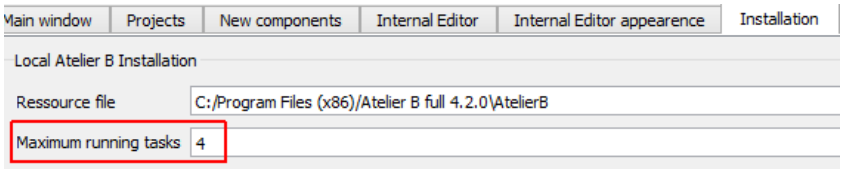
Translations produced are summarized in the following table.

B model (machine M3)	Profile C9X	Profile light	Profile 01
I1 : INT	static int32_t M3__i1;	static long M3__i1;	static int32_t i1;
B1 : BOOL	static bool M3__b1;	static unsigned char M3__b1;	static bool b1;

Proof server

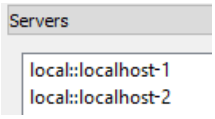
Since release 4.1, parallel execution of proof tasks has been introduced (see Release Notes 4.1.0). This ability is extended with Atelier B 4.2 with the use of:

- Several cores of the local machine,
- A remote proof server (**Linux only**).



Parallel task execution is determined by the number by a maximum number of running tasks, strictly greater than 1 (see parameter "*maximum running tasks*"). The number of tasks really executed in parallel is always greater or equal to the number of cores available locally. If this parameter is 0, then no core of the local machine is used and all proof effort is transferred to remote proof server, if any.

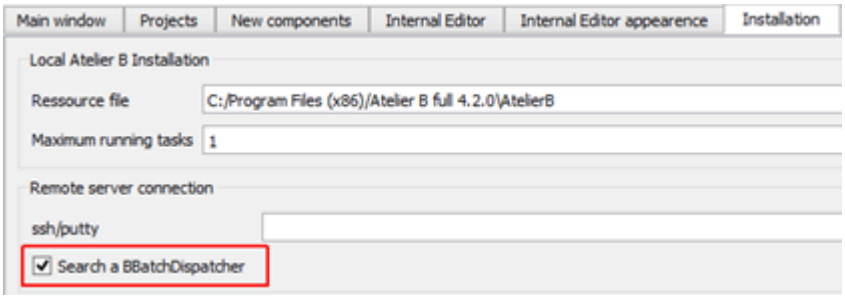
The list of available cores is displayed in the window « servers », starting with the cores of the local machine. Cores are named « *local::<IP address>-<core index>* » where *<IP address>* is equal to the IP address of the machine executing the task on its core # *<core index>*. IP address of the local machine is localhost.

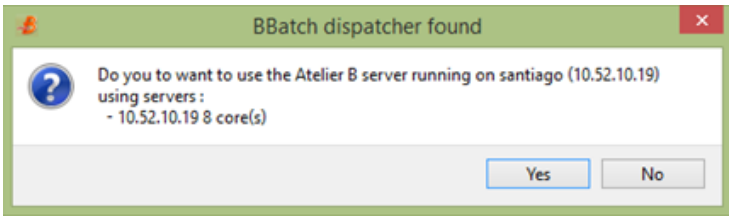


To trigger parallel execution of task, one component of an open project has to be selected. The tasks number value has to be set up (see Figure 7). Selecting proof action (F0, F1, etc.) initiates a number of parallel tasks corresponding to that value.

Figure 7: tasks number value (here 4)

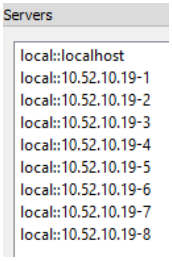
To connect to a proof server and have more computation power available, Atelier B configuration has to be modified first to enable the search for proof servers. For this, "search a BBatch Dispatcher" has to be selected in the "installation" preferences page. Then Atelier B has to be restarted.





If at least one proof server is available on the local network, a window would show up, inviting to connect to this server (indication: IP address, number of cores).

In case of positive answer, the list of cores available in the "servers" window will be extended. Local machine cores are always listed before remote cores. When executing tasks in parallel, the search of an available core is always performed according to this order.



A proof server is in fact a "proof concentrator": it links Atelier B clients with mono-core or multi-cores machines performing proof tasks. There is no constraint on the localization of a proof sever that can execute on any Linux computer.

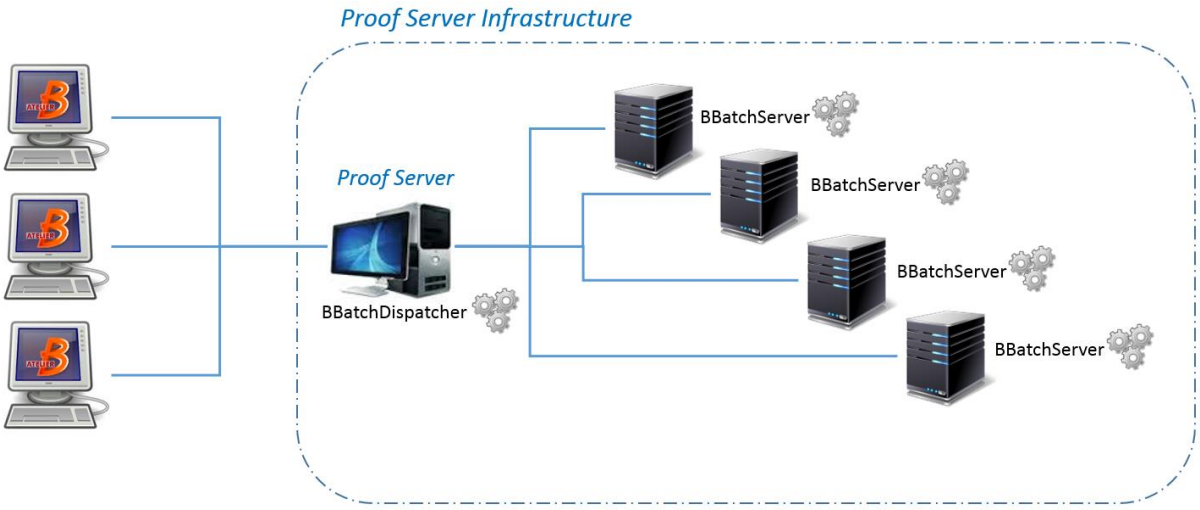


*Figure 8: Proof infrastructure – **Atelier B has to be installed on all these machines***

It is mandatory to execute (see Figure 8):

- a processus ***BBatchDispatcher*** on the proof server,
- a processus ***BBatchServer*** on each machine having cores(s) available for the proof tasks.

These executable files are located in the Atelier B install directory.

To execute a ***BBatchDispatcher***, type the following command:

```
./bbatchdispatcher <hostname> <hostaddress>
```

where <*hostname*> is the name of the computer and <*hostaddress*> its IP address.

A web server is started and reachable at the address http://localhost:<port>/servers.html (the port number, <*port*>, is returned when launching the BBatchDispatcher). It provides an information and command interface (see Figure 9) which lists associated BBatchServers and their number of cores.

Figure 9: BBatchDispatcher web server indicating BBatchServers status

With this interface, cores can be allocated to proof (action « *reserve* ») or released (action « *release* »)[4].

[4] Cores use is only limited within BBatchServer. This feature doesn't prevent the execution of third party processes on these cores.

To execute a ***BBatchServer***, type the following command:

```
./bbatchserver <hostaddress> <cores> -d <dispatcheraddress>
```

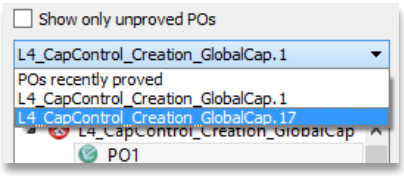
where <*hostaddress*> is the IP address of the computer, <*cores*> the number of available cores and <*dispatcheraddress*> the IP address of the proof server executing ***BBatchDispatcher***.

Diverse improvements

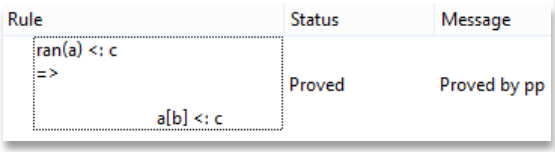
The main new features of Atelier B 4.2 are listed below by category:

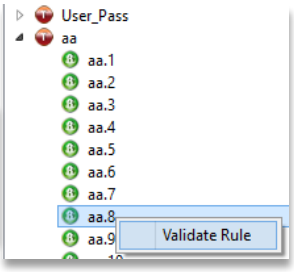
Prover :

- When proving interactively, a proof history is stored, enabling to navigate through proof obligations previously demonstrated manually. This history is local to the computer executing the interactive prover.

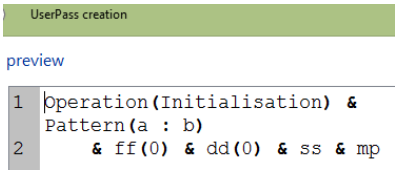


- The pmm editor offers to validate a rule in the outline view, with a context menu. Select a rule on the outline and right click on it.

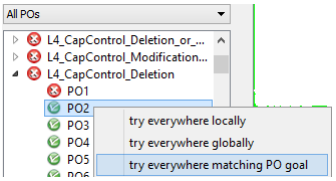




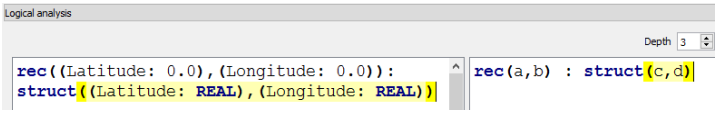
- When a proof obligation is demonstrated and saved, the demonstration is saved in the "*User Pass*". The formatting separates the name of the operation on the first line, the pattern of the proof obligation on the second line and the list of commands starting on the third line.



- The TryEverywhere command, allowing to try successful demonstrations on other proof obligations, has a new mode of execution: with a context menu ("try everywhere matching goal"), a demonstration can be tried on all unproved proof obligations of a component that have exactly same goal.



- Interactive prover logic formula analyser supports new real and float operators.

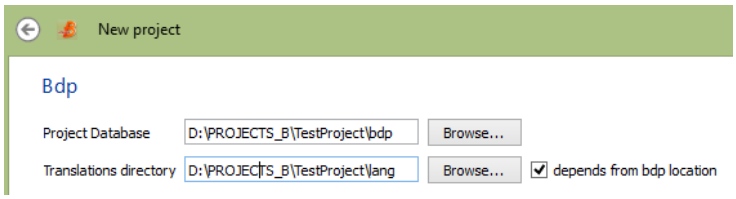


- When triggering the interactive prover "reset" button (return to the proof tree root of the current proof obligation), a confirmation is required in order to avoid losing the ongoing demonstration.

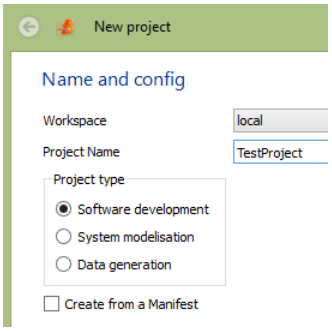
- Rules *EqualityXY.148* and *EqualityXY.149* were not generic enough. Their typing guards (*binhyp*) have been deleted.

Project management:

- Project creation has been simplified since now only one directory is required (the root directory of the project). If this directory contains *lang* and *bdp* directories (or equivalent specified in the preferences) that are selected as « project database » and « translation directory», then these directories are associated to the project. If these directories do not exist, they are created.



- To improve project management, it is now possible to associate to a *Manifest* file to a project. *Manifest* file is an xml file containing the list of files to add to the project with their relative path (see Figure 10). To create this file, an open project has to be selected then the « Synchronize with Manifest » context menu action has to be triggered. A directory and a filename have to be chosen for this *Manifest* file. Warning! It should not be saved in the *bdp* directory which already contains a MANIFEST file used for archiving projects. To create a project from a Manifest file, it is required to select « Create from a Manifest » during its creation.




```
<project>
  <add_file path="../../___LIBRAIRIE/B/L4B4/CapId.mch"/>
  <add_file path="../../___LIBRAIRIE/B/L4B4/L4ceKernelAPI.mch"/>
  <add_file path="../../___LIBRAIRIE/B/L4B4/ThreadId.mch"/>
  <add_file path="../../___LIBRAIRIE/B/L4B4/Time.mch"/>
  <add_file path="../../___LIBRAIRIE/B/L4B4/Types.mch"/>
</project>
```

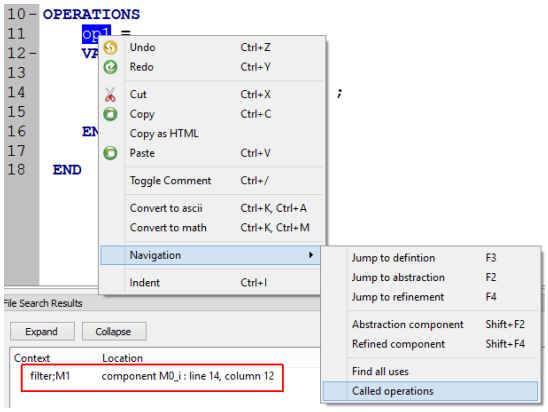
Figure 10: example of MANIFEST file

Rules proof tool (Atelier B Maintenance Edition):

- Rules containing dangerous list patterns ([a] or {a}) are more easily identifiable:
  - User can trigger an action which find rules containing [a] or {a} patterns (whatever the wildcard) and which change proof rule statuses to "Invalid".
  - The Html report displays a warning for each rule containing [a] or {a} pattern
  - The GUI displays a warning when a rule contains [a] or {a} pattern
- Rule selection is performed with a single-click, instead of a double-click
- The timestamp used to tag a verified rule in a pmm file now complies with UTC date ISO format.
- A rule can be de-verified.
- When a rule is verified, its status is displayed (« verified OK » or « verified NOK »). Status is displayed in the navigator (files, theories and rules list) and in the visualisation area of the current rule.
- In the navigator, colours have been replaced by icons in order to quickly identify verified, partly verified or non-verified elements.
- Tooltips make precise what the various numbers in the navigator mean.
- Current mode name (specification, design) is displayed in the window title.

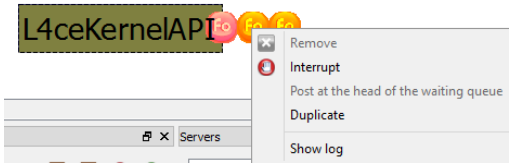
Ergonomics:

- The context menu associated to an operation now shows up two new actions: « *Find all uses* » and « *Called operations* ». The former lists all implementations calling this operation. The latter lists all operations that are called in the implementation of this operation. These lists are displayed in the window « *File search results* ».



- Long error messages are now displayed on several lines.

- In the component view (graphic view), tasks can be rescheduled, interrupted or removed (actions available on context menu). When a task is rescheduled, all the tasks of the component are rescheduled accordingly.



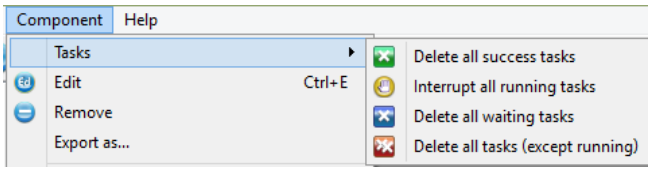
- B Compiler error messages are more understandable: when displaying type errors, types are now displayed in accordance with B models and not using B Compiler internal type (more precise but less clear).

- A context menu of the model editor allows to close all tabs or only the current one.

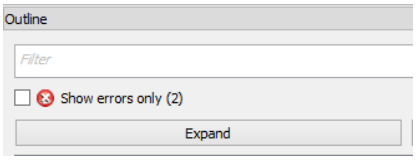
- Action "Open folder" in the project menu allows to open the project root directory in the system file explorer.

- The content of the « outline » view is generated on the background and doesn't delay the opening of the editor.

- In the component menu, a sub-menu, "tasks", has been added, enabling the interruption or deletion of a tasks of a component. The last action (« delete all tasks (except running ») deletes waiting tasks, error tasks and completed tasks from the tasks list.



- Elements in the « outline » view can be filtered, that is useful if the component contains a large number of elements. The filter can be set up to only display errors.



- B0Check verifications are not displayed by default, as these verifications are sometimes not related to code generator used (some verifications are related to Atelier B Maintenance Edition code generators).

- Accents and spaces in filenames and path are better supported.