Version 4.2

Date de diffusion : Décembre 2014

L'Atelier B 4.2 est disponible en deux versions :

- La version ***Community Edition***, utilisable par tous sans restriction. Cette version n'est pas maintenue.
- La version ***Maintenance Edition***, dont l'accès est réservé aux possesseurs d'un contrat de maintenance Atelier B 4 qui disposent ainsi d'un support technique (maintenance corrective) ainsi que d'un accès anticipé aux nouvelles fonctionnalités. Certaines fonctionnalités (traducteurs Ada et C++, outil de preuve de règles mathématiques) sont uniquement accessibles dans cette version, comme détaillé dans le tableau ci-dessous.

Fonctionnalité	Atelier B 4.2 Community Edition	Atelier B 4.2 Maintenance Edition
Environnement de développement	✓	✓
Support projet langage B	✓	✓
Support projet langage Event-B	✓	✓
Support projet validation de données	✓	✓
Editeur de modèles B et Event-B	✓	✓
Raffineur Automatique	✓	✓
Vérificateur de type	✓	✓
Générateur d'obligations de preuve	✓	✓
Prouveur Automatique	✓	✓
Prouveur interactif	✓	✓
Prouveur de prédicats	✓	✓
Traducteur C C4B	✓	✓
Traducteur Ada (MacOS, Linux)		✓
Traducteur High Integrity Ada (MacOS, Linux)		✓
Traducteur C++ (MacOS, Linux)		✓
Outil de validation de règles mathématiques		✓

Nouvelles fonctionnalités / Caractéristiques :

L'Atelier B 4.2 a été mis à disposition de tous le 19 Décembre 2014.

Cette version corrige 151 anomalies et propose 47 améliorations.

Parmi ces améliorations, on notera :

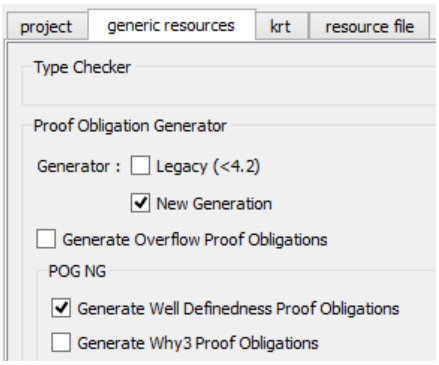
- Le support complet 64-bit
- Un nouveau générateur d'obligations de preuves, traçable et paramétrable
- Une meilleure intégration des types réels et flottants
- L'ajout des accesseurs dans Bart permettant de traiter les conflits de raffinement
- Le paramétrage fin des types booléen et entier pour C4B.
- L'introduction d'un serveur de preuve, pour paralléliser l'effort de preuve

Générateur d'Obligations de Preuve Générique

Un nouveau générateur d'obligations de preuve (GOP) a été développé[1] afin d'apporter de nouvelles fonctionnalités :

- traçabilité des obligations de preuve, au travers d'une interface graphique intégrée permettant d'associer modèle et preuve ;
- simplification des obligations de preuve par modification des principes de normalisation des formules ;
- possibilité de définir et d'ajouter ses propres obligations de preuve, au travers des fichiers xsl définissant les obligations de preuve théoriques pour le B logiciel, le B événementiel et la bonne-définition de ces modèles.

Ce nouveau générateur produit sensiblement le même nombre d'obligations de preuve que précédemment. Celles-ci peuvent varier toutefois dans leur forme et il est probable qu'un pprojet prouvé avec une version antérieure de l'Atelier B ne soit plus complétement prouvé, en preuve automatique ou par rejeu, avec la version 4.2. Le nouveau générateur d'obligation de preuve est sélectionné par défaut (*New Generation*). Pour utiliser l'ancien GOP, il est nécessaire de modifier la configuration des projets et sélectionner « *Legacy (<4.2)* ». Il est par ailleurs possible de choisir de générer ou non les obligations de preuve de débordement arithmétique, de bonne définition ainsi que celles au format Why3[2].



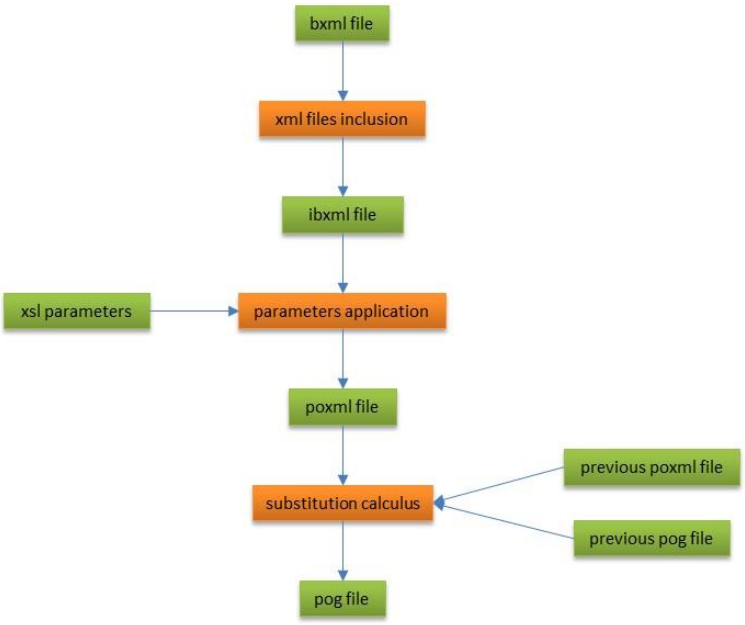


Figure 1: les fichiers manipulés par l'Atelier B et leurs dépendances

Les fichiers manipulés par l'Atelier B 4.2 ont changé (voir Figure 1) et sont aujourd'hui au format xml. Les modèles B sont sauvegardés au format bmxl. Les obligations de preuve utilisent le format

[1] Dans le cadre du projet Cercles-2 grâce au support de l'Agence Nationale pour la Recherche

[2] Le fichier résultat a comme nom celui du composant et comme extension why. Il est situé dans le répertoire « bdp » du projet.

poxml afin d'assurer la génération différentielle : seules les portions de modèles modifiées nécessitent un recalcul, les démonstrations existantes sont sauvegardées.

Traçabilité

L'origine des obligations de preuve (OP) est enfin connue avec précision. Pour chaque obligation de preuve, les portions de modèles en rapport sont affichées dans un onglet à droite de celle-ci (voir Figure 2).Les expressions contenues dans les POs sont liées avec le code, lors du surlignage d'une expression de l'obligation de preuve, avec l'affichage et surlignage du code correspondant.

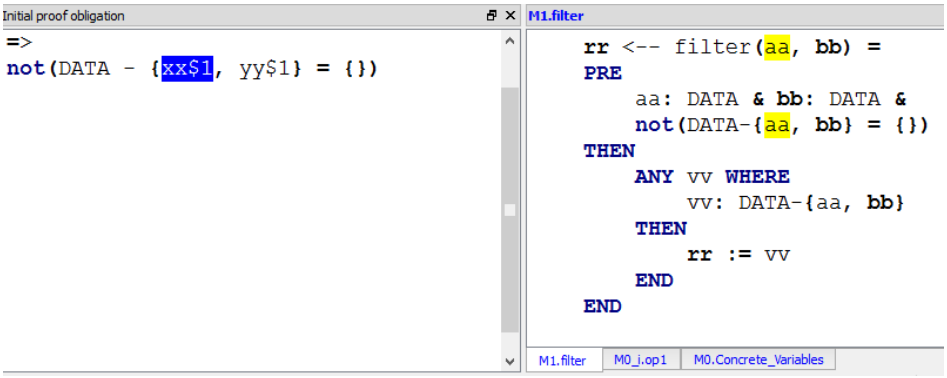
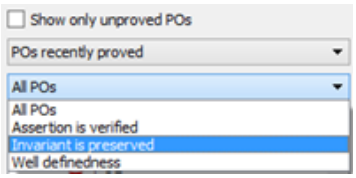


Figure 2: obligation de preuve avec ses informations de traçabilité

Les OP peuvent être filtrées par type dans l'interface de preuve interactive, grâce à un menu déroulant dynamique qui ne contient que la liste des types d'obligations de preuve du composant.



Versatilité

Les obligations de preuve théoriques sont des paramètres de l'outil. Elles sont disponibles dans le répertoire d'installation de l'Atelier B, dans le sous-répertoire press/include :

- *paramGOPSoftware.xsl* : obligations de preuve pour le B logiciel
- *paramGOPSystem.xsl* : obligations de preuve pour le B système
- *wellDefinedness.xsl* : obligations de preuve pour la bonne définition

Ces obligations de preuve peuvent être étendues en modifiant ces fichiers.

```
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: obligation de preuve pour la bonne définition des propriétés des constantes d'une machine

Par ailleurs, dans ce même répertoire, on trouve la définition de la grammaire du bxml (*bxml.xsd*), nouveau format de représentation des modèles B permettant l'interopérabilité.

Simplicité

De nouveaux principes de normalisation des prédicats ont été définis. Ils sont listés dans les 3 tableaux ci-dessous.

Normalisation but et hypothèses	Prédicat (expression) normalisé(e)
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)-{{}}

Tableau 1: normalisation des prédicats et expressions dans le but et hypothèses (remplacement de prédicat/expression par leur forme normalisée)

Normalisation des affectations (lors de la génération des Pos)	Prédicat normalisé
a(b) := c	a := a <+ {b |-> c}
a'b := c	a := a <<< {b$8888 = c}

Tableau 2: normalisation des affectations (remplacement de prédicat par sa forme normalisée)

Normalisation hypothèses (ajout nouvelles hypothèses)	Prédicat normalisé
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

Tableau 3: normalisation des prédicats dans les hypothèses (de nouvelles hypothèses sont créées)

Un nouvel opérateur de mis à jour des records a été introduit lors de la génération des obligations de preuve[3]. Il permet d'éviter les explosions des expressions et de la mémoire lors de la modification de record. Il corrige aussi les anomalies connues concernant les records, notamment des captures de nom entre les labels des records et les variables du modèle.

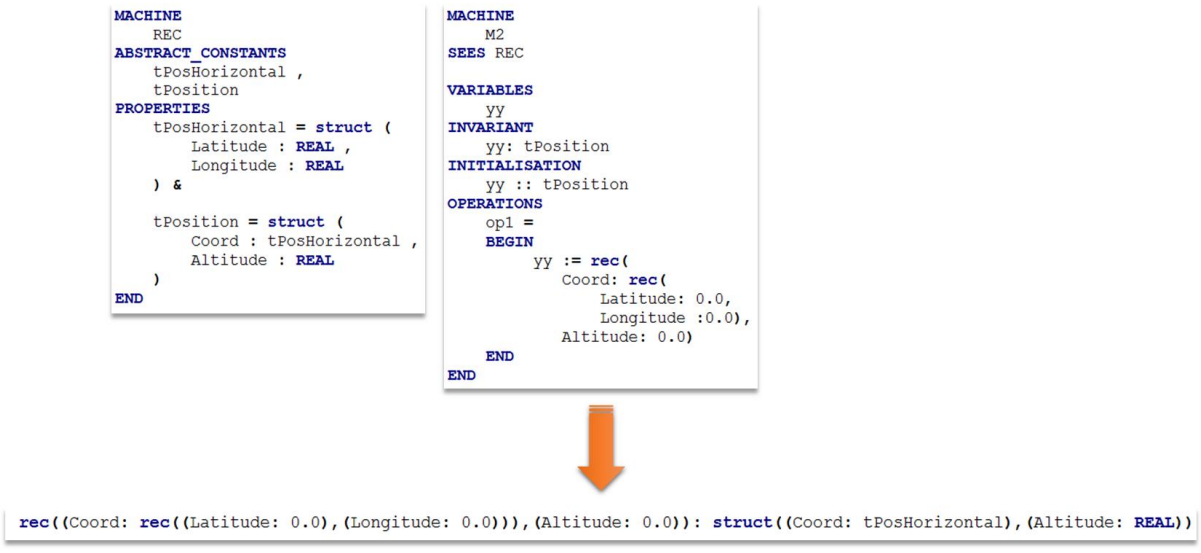


Figure 4: exemple d'obligation de preuve générée lors de la valuation d'un record imbriqué - tous les labels doivent être indiqués

Ce générateur d'obligations de preuve n'a pas encore été qualifié dans le cadre d'un développement sécuritaire. Son utilisation requiert des précautions avant toute utilisation pour un développement de niveau SIL3 ou SIL4.

[3] équivalent du *with (field update)* du langage why3)

Réels et flottants

Depuis la version 4.1, il existe un support des nombres réels et nombres flottants (voir la Release Notes 4.1.0 pour cette version). Le type des réels est **REAL**, celui des flottants est **FLOAT**. Avec la version 4.2, la gestion des nombres réels et nombres flottants a été améliorée et a subi des modifications.

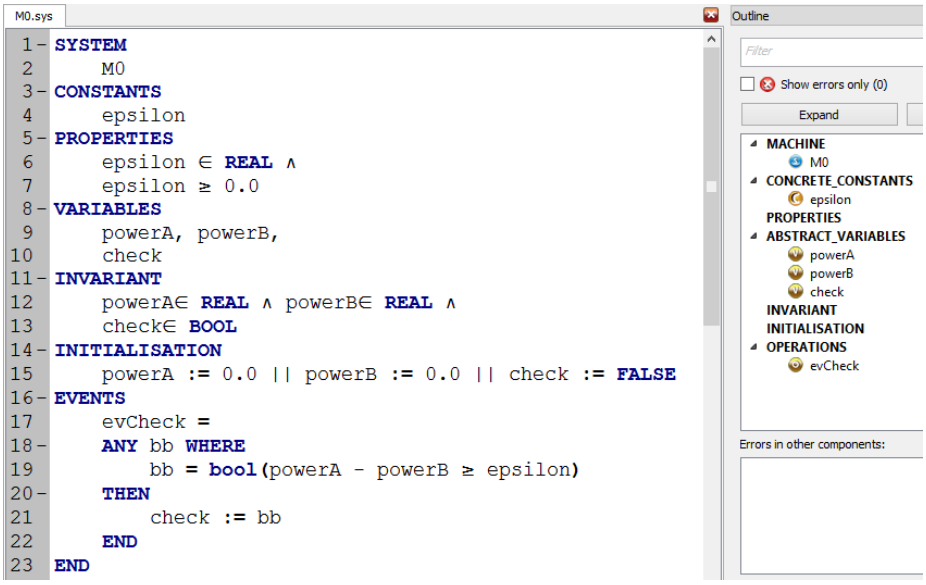


Figure 5: modèle événementiel incluant des variables réelles

Il y a plusieurs points importants à noter :

- les nombres non entiers sont uniquement gérés par le **nouveau** GOP ;
- Contrairement à la version 4.1 qui distinguait les opérateurs entiers, réels et flottants, les opérateurs utilisés dans les modèles B sont unifiés (voir Tableau 4, Tableau 5 et Tableau 6)
- Par contre, en phase de preuve, les opérateurs sont distingués car leur sémantique est différente selon leur type. Le langage utilisé n'est pas le même dans le source des composants et dans le prouveur (et donc aussi dans les règles des fichiers `.pmm`)

Lors de la génération des obligations de preuves, il y a conversion depuis la syntaxe unifiée vers une syntaxe dédiée aux types non-entiers. Le type des opérandes est utilisé pour déterminer quel type d'opérateur utiliser. Il n'y a pas de conversion ni de coercition implicite.

Unifié	Entier	Réels	Flottants
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***

Tableau 4: conversion des prédicats des modèles (colonne de gauche) pour la preuve, en fonction de leur type

Les conditions de bonne définition sont équivalentes à celle des opérateurs entiers historiques.

Il est possible de passer des nombres entiers aux réels (et vice-versa) avec les opérateurs ci-dessous.

Signification type résultant	Syntaxe	Type de l'opérande
plongement entier dans réel **real(x) : REAL**	real(x)	x : INTEGER
partie entière **floor(x) : INTEGER**	floor(x)	x : REAL
partie entière par excès **ceiling(x) : INTEGER**	ceiling(x)	x : REAL

Tableau 5: opérateurs de conversion entier/réel et réel/entier

Les flottants sont considérés comme un type implémentable et ils ne possèdent donc pas des opérateurs de spécification comme `min`, `max`, `SIGMA` et `PI`.

Il n'y a pas de littéraux flottants, il faut passer par une machine de base.

Il n'y a pas d'opérateur prédéfini pour passer des flottants aux réels, ou des flottant aux entiers (et vice-versa).

La normalisation (dans les obligations de preuves) de la relation d'ordre des réels et des flottants est résumée ci-dessous.

Unifié	Entier	Réels	Flottants
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`

Tableau 6: normalisation de la relation d'ordre pour les entiers, réels et flottants

BART

- ***Ajout des accesseurs pour traiter les conflits d'implantation de variables***.

  Les règles classiques de raffinement de substitution (THEORY_OPERATION) peuvent nécessiter l'implémentation ou l'exportation de variables. On entend par exportation, le report de l'implémentation d'une variable dans une machine importée. Ceci engendre des contraintes sur l'ordre dans lequel doivent être implémentées les différentes variables dans la colonne de raffinement générée. Lorsque ces contraintes sont incompatibles, un conflit apparaît et il est impossible de générer le raffinement.

  La solution pour éviter les conflits est d'utiliser un type spécial de règles de substitution pour chaque règle nécessitant l'implémentation d'une variable : les règles dites d'accesseur (THEORY_ACCESSOR). Ces règles sont des règles élémentaires décrivant le raffinement d'une substitution nécessitant l'implémentation d'une variable.

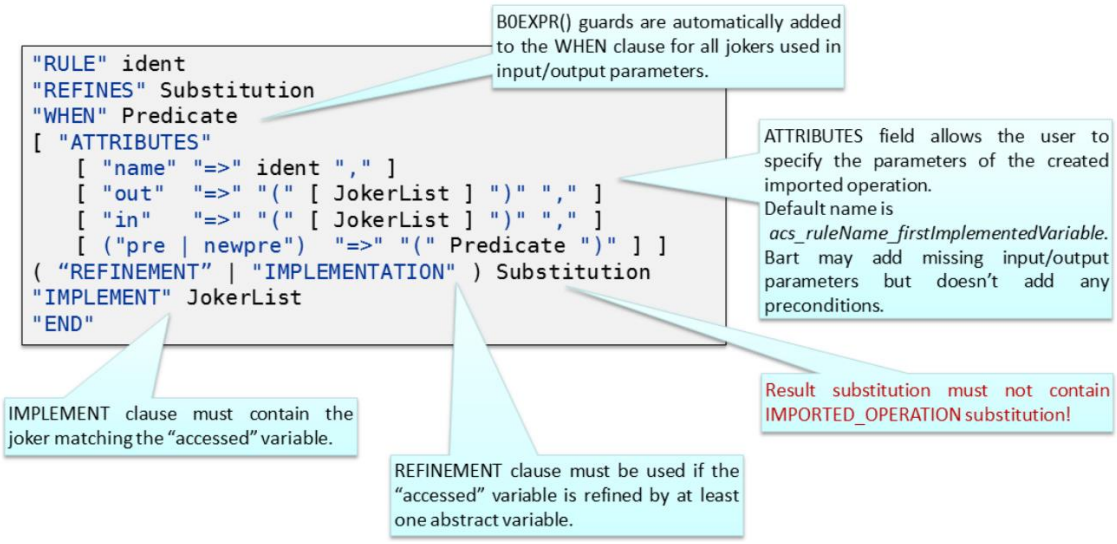


Figure 6: syntaxe d'une règle d'accesseur

  En pratique, ce raffinement est automatiquement importé dans une opération "accesseur", ce qui permet à l'algorithme qui répartit les opérations dans la colonne de raffinement d'éviter les conflits. En résumé, la théorie THEORY_OPERATION ne doit contenir aucune règle imposant l'implémentation d'une variable et la théorie THEORY_ACCESSOR doit contenir les règles implémentant une variable afin de la lire ou de la modifier. L'utilisation d'une variable dans les règles d'opération doit se faire à l'aide de substitutions abstraites (les substitutions raffinées par les règles d'accesseur).

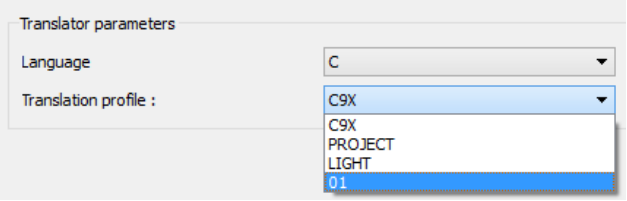
- Les composants générés automatiquement apparaissent différemment des composants développés manuellement : leur nom est affiché en italique.

- La mise à jour des fichiers générés par BART est réalisée sans suppression systématique : les fichiers qui ont disparus par rapport à la liste précédente sont supprimés, les nouveaux fichiers par rapport à cette liste sont ajoutés. Le statut des composants non modifiés est ainsi conservé (preuve, typecheck, etc...).

Paramétrage des types booléen et entier pour C4B

Depuis la version 4.1, le générateur de code C C4B a remplacé ComenC (voir la Release Notes 4.1.0 pour cette version).

Un nouveau mode de génération de code a vu le jour : « 01 ». Il permet de générer du code conforme au profil C9X sans que les noms des variables et constantes soient préfixés par le nom du composant les contenant.



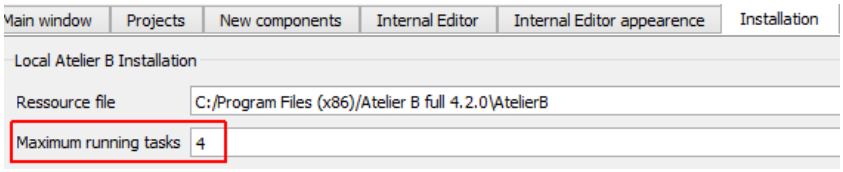
Les résultats de traduction sont résumés dans le tableau ci-dessous.

Modèle B (machine M3)	**Profil C9X**	**Profil light**	**Profil 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;

Serveurs de preuve

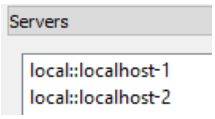
Depuis la version 4.1, il est possible d'exécuter des tâches de preuve en parallèle (voir la Release Notes 4.1.0 pour cette version). Cette fonctionnalité a été étendue avec la version 4.2 grâce à l'utilisation:

- de plusieurs cœurs de la machine locale,
- d'un serveur de preuve distant (**uniquement sur Linux**).



L'exécution de tâches en parallèle est déterminée par un nombre maximal de tâches strictement supérieur à 1 (voir paramètre "*maximum running tasks*"). Le nombre de tâches réellement exécutées en parallèle sera toujours inférieur ou égal au nombre de cœurs disponibles. Si ce paramètre vaut 0, alors aucun cœur de la machine locale ne sera utilisé, l'effort de preuve sera alors reporté sur le serveur de preuve distant, s'il existe.

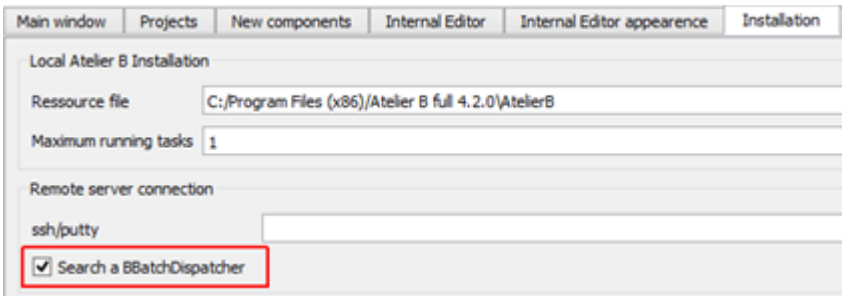
La liste des cœurs disponibles est affichée dans la fenêtre « servers », en commençant par les cœurs de la machine locale. Les cœurs sont nommés « *local ::<adresse ip>-<index cœur>* » avec *<adresse ip>* égal à l'adresse IP de la machine exécutant la tâche sur son cœur n° *<index cœur>*. L'adresse IP de la machine locale est *localhost*.

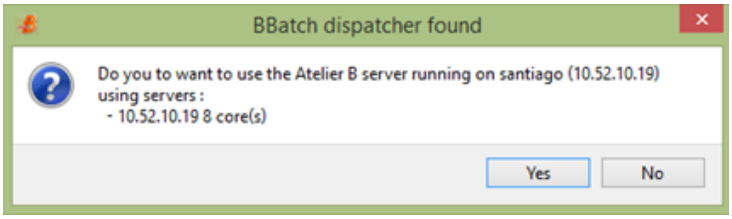


Pour déclencher l'exécution de tâches de preuve en parallèle, il suffit de sélectionner au moins un composant d'un projet, de positionner le sélecteur du nombre de tâches (voir Figure 7) à la valeur voulue puis de sélectionner l'action de preuve (F0, F1, etc.).

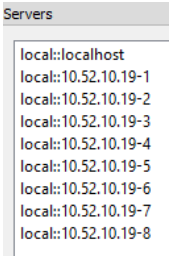
Figure 7: sélecteur du nombre de tâches en parallèle (ici 4)

Pour pouvoir se connecter à un serveur de preuve et disposer de bien plus de puissance de calcul, il faut au préalable modifier la configuration de l'Atelier B en autorisant la recherche de serveur de preuve. Pour cela, il convient de cocher la case « *search a BBatchDispatcher* » dans la page de configuration « *installation* » de l'Atelier B. Il faudra ensuite redémarrer l'Atelier B.





Si au moins un serveur de preuve est actif sur le réseau local, une fenêtre apparaitra, invitant à se connecter à ce serveur, avec l'indication de son adresse IP et du nombre de cœurs.



En cas de réponse positive, la liste des cœurs disponibles sera étendue avec ceux du serveur de preuve. Les cœurs de la machine locale sont toujours listés avant ceux du serveur de preuve (distant). Lors de l'exécution des tâches en parallèle, la recherche de cœur disponible se fera selon cet ordre.

Un serveur de preuve est un « concentrateur de preuves » : il met en relation des clients Atelier B avec des machines mono ou multi-cœurs réalisant ces preuves. Il n'y pas de contrainte de localisation d'un serveur de preuve qui peut s'exécuter sur n'importe quelle machine Linux.

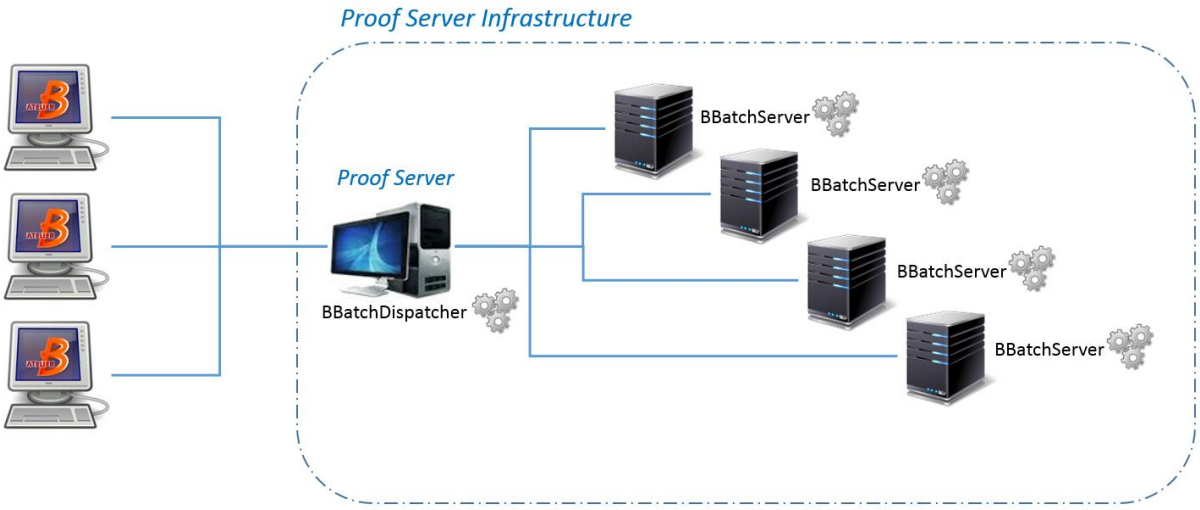


Figure 8: infrastructure de preuve- ***l'Atelier B doit être installé sur toutes ces machines***

Il est nécessaire d'exécuter (voir Figure 8) :

- Un processus ***BBatchDispatcher*** sur le serveur de preuve
- Un processus ***BBatchServer*** sur chacune des machines mettant à disposition leurs cœurs pour la réalisation des preuves.

Ces fichiers exécutables se trouvent dans le répertoire d'installation de l'Atelier B.

Pour exécuter un ***BBatchDispatcher***, il convient d'exécuter la ligne de commande:

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

où <*hostname*> est le nom de la machine et <*hostaddress*> son adresse IP.

Un serveur web est alors démarré à l'adresse http://localhost:<port>/servers.html (le numéro du port, <*port*>, est indiqué lors du lancement). Il fournit une interface d'information et de commande (voir Figure 9) qui indique la liste des BBatchServers associés, ainsi que le nombre de cœurs associés.

Figure 9: serveur web du BBatchDispatcher indiquant l'état des BBatchServers associés

Il est possible de ne pas utiliser tous les cœurs (action « *reserve* ») ou bien d'autoriser à en utiliser plus (action « *release* »)[4].

[4] L'utilisation des cœurs n'est limitée que dans le cadre du BBatchServer. Ce mécanisme n'empêche pas l'exécution de processus autres sur ces cœurs.

Pour exécuter un ***BBatchServer***, il convient d'exécuter la ligne de commande:

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

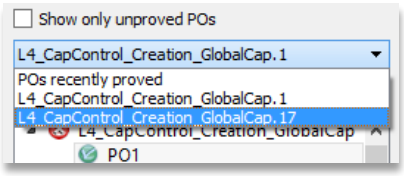
où <*hostaddress*> est l'adresse IP de la machine, <*cores*> le nombre de cœurs disponibles et <*dispatcheraddress*> l'adresse IP du serveur de preuve exécutant ***BBatchDispatcher***.

Améliorations diverses

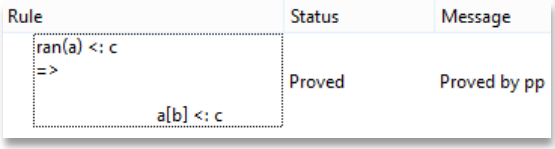
Les principales améliorations apportées à l'Atelier B 4.2 sont regroupées ci-dessous par catégorie :

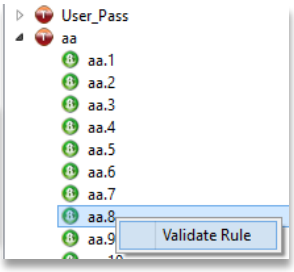
Prouveur :

- En preuve interactive, il est possible de revenir à une OP qui a été démontrée de manière interactive et manuellement. Un menu déroulant apparait avec la liste des obligations de preuve traitées de manière manuelle. L'historique est propre au poste de travail.

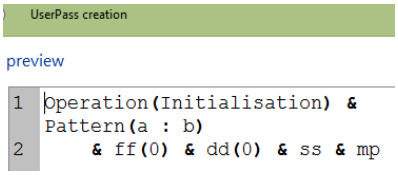


- Au sein de l'éditeur de fichier pmm, un menu contextuel accessible dans la vue *outline* permet de valider une règle.

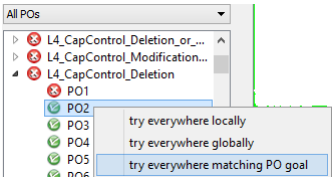




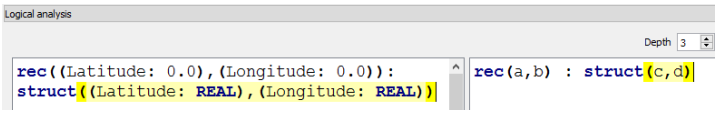
- Pour la génération de « *User Pass* » lors de la sauvegarde d'OP, celle-ci est mise en forme par décomposition de la ligne de commande avec le filtre sur le nom de l'opération d'abord, puis le filtre sur la forme et enfin la démonstration sous la forme d'une liste de commandes.



- La commande TryEverywhere qui permet d'appliquer une démonstration réussie à d'autres obligations de preuve non prouvées dispose d'un nouveau mode d'exécution : il est possible, à partir de la liste des obligations de preuve prouvées (menu contextuel), d'appliquer globalement la démonstration d'une obligation de preuve à toutes les obligations de preuve de même forme, non prouvées.



- L'analyse logique de formule du prouveur interactif prend en compte les nouveaux opérateurs réels et flottants.

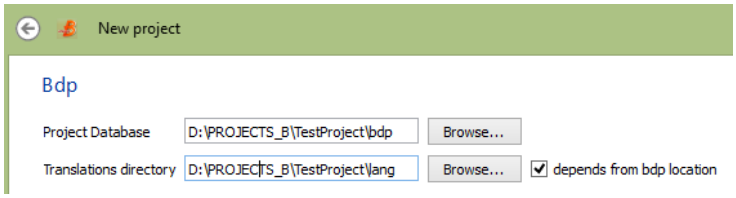


- Une confirmation a été ajoutée lorsque l'on actionne le bouton *reset* du prouveur interactif (retour à la racine de l'arbre de preuve de l'obligation de preuve courante), afin d'éviter de perdre la démonstration interactive en cours.

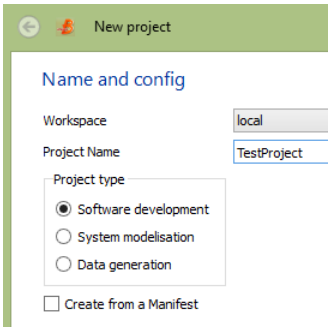
- Les règles *EqualityXY.148* et *EqualityXY.149* n'étaient pas assez générales. Les gardes de typage (*binhyp*) ont été supprimées.

Gestion de projet :

- La création de projet a été simplifiée puisqu'elle ne requiert plus qu'un répertoire unique (répertoire racine du projet). Si ce répertoire contient des sous-dossiers lang et bdp (ou équivalents spécifiés dans les préférences), qui sont sélectionnés comme base de données projet et répertoire de traduction, alors ces répertoires sont associés au projet. S'ils ne sont pas présents, ils sont créés.



- Afin de permettre une meilleure gestion en configuration des projets, il est maintenant possible d'associer à un projet un fichier *Manifest*. Le fichier *Manifest* est un fichier xml contenant la liste des fichiers à ajouter au projet avec un chemin relatif (voir Figure 10). Pour créer ce fichier, il suffit de sélectionner un projet ouvert puis de choisir dans le menu contextuel l'action « Synchronize with Manifest ». Il vous faut alors choisir le répertorie et le nom du fichier *Manifest*. Attention ! Il ne doit pas être sauvegardé dans le répertoire bdp du projet qui contient déjà un fichier MANIFEST utilisé pour l'archivage des projets. Pour pouvoir créer un projet à partir d'un fichier *Manifest*, il suffit de cocher la case « Create from a Manifest » lors de la création de celui-ci.




```
<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>
```

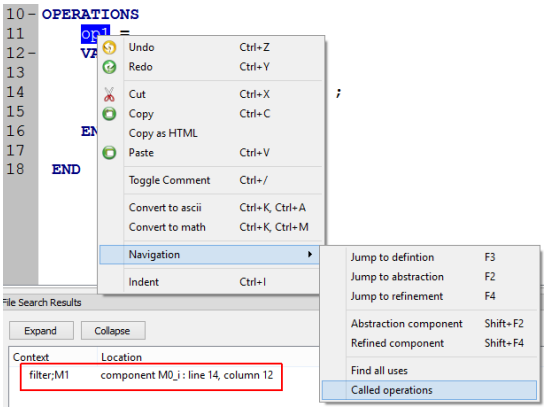
Figure 10: exemple de fichier MANIFEST

Outils de preuve de règles (Atelier B Maintenance Edition):

- Les règles contenant des patterns de liste dangereux ([a] ou {a}) sont plus facilement identifiables :
  - l'utilisateur peut lancer une action qui trouve les règles possédant des motifs en "[a]" ou "{a}" (quel que soit le joker) et qui passe le status de preuve de la règle à invalide.
  - l'export html fait apparaître une ligne de texte pour chaque règle contant un motif en [a] ou {a}
  - au sein de l'IHM, la vue d'une règle affiche un avertissement quand la règle visualisée contient un motif en [a] ou {a}
- Le changement de règle se fait avec un simple clic, au lieu d'un double clic.
- L'horodatage utilisé pour marquer une règle vérifiée dans le fichier pmm utilise maintenant la date UTC au format ISO.
- Une règle peut être dé-vérifiée.
- Lorsqu'une règle est vérifiée, son statut est affiché ("vérifiée OK" ou "vérifiée NOK"). Ce statut apparaît dans le navigateur (liste des fichiers, théories et règles) et dans la zone de visualisation de la règle courante.
- Dans le navigateur, les couleurs ont été remplacées par des icônes pour pouvoir identifier, a minima, les règles, théories ou fichiers, qui sont vérifiés, partiellement vérifiés ou non vérifiés
- Des info-bulles précisent la signification des nombres qui se trouvent dans le navigateur.
- Le nom du mode courant est affiché (conception ou vérification) dans le titre de la fenêtre.

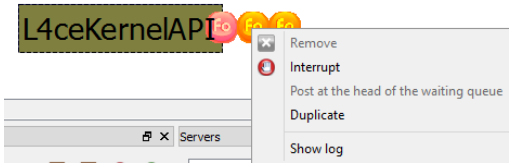
Ergonomie :

- Dans le menu contextuel associé à une opération, deux nouvelles actions ont été ajouées : « *Find all uses* » et « *Called operations* ». La première permet de trouver toutes les implémentations faisant appel à cette opération. La seconde permet de trouver toutes les opérations qui sont appelées dans l'implémentation de cette opération. Dans chaque cas, le résultat est une liste apparaissant dans la fenêtre « *File search results* ».



- Les longs messages d'erreur sont dorénavant affichés sur plusieurs lignes

- Dans la listes des tâches, il est possible de ré-agencer les tâches ou pouvoir leur donner une priorité d'exécution. en mettre en pause. Lorsqu'on reschedule automatiquement une tâche, c'est toutes les taches du composant qui sont reschedulées dans l'ordre.



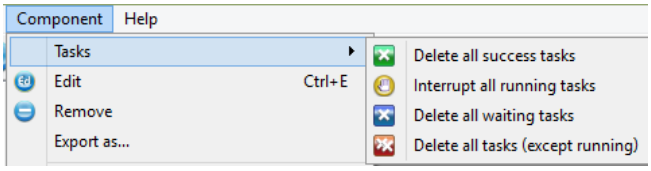
- Les messages d'erreurs du Compilateur B sont plus compréhensibles : lors de l'affichage des erreurs de types, le format de type tel qu'il apparaît dans les sources B est utilisé plutôt que le format de type du Compilateur B (plus riche mais moins lisible).

- L'éditeur de modèle permet, par menu contextuel, de fermer tous les onglets ou juste l'onglet courant.

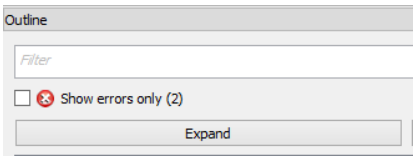
- Dans le menu projet, l'action « Ouvrir le répertoire » permet d'ouvrir le répertoire du projet dans l'explorateur de fichiers.

- La génération de la vue « *outline* » est lancée en tâche de fond lors de l'ouverture de l'éditeur et ne retarde plus l'ouverture de l'éditeur

- Dans le menu composant, un menu tâches a été ajouté afin de pouvoir interrompre ou supprimer les tâches d'un composant. La dernière action du menu (« suppression de toutes les tâches ») permet de supprimer les tâches en attente, les tâches terminées et les tâches en erreur.



- Il est maintenant possible de filtrer les éléments qui sont affichés dans la vue « *outline* » de l'éditeur. Cela peut être utile lorsqu'on veut aller à une variable ou un événement dans le fichier, et que le fichier contient un grand nombre d'éléments.



- L’affichage des vérifications B0Check dans l’éditeur est désactivé par défaut, du fait que les vérifications B0 ne correspondent pas forcement aux véritables contraintes des traducteurs utilisés.

- Les accents et espaces dans les chemins ou les machines sont mieux gérés.