Version 4.1.0

Date de diffusion : décembre 2012

L'Atelier B 4.1.0 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.

Fonctionnalité	Atelier B 4.1 Community Edition	Atelier B 4.1 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 4.1.0 a été mis à disposition de tous le 11 Décembre 2012.

Cette version corrige 385 anomalies (37 pour la version 4.0.1, 79 pour la version 4.0.2 et 269 pour la version 4.1.0) et propose 56 améliorations (5 pour la version 4.0.1, 13 pour la version 4.0.2 et 38 pour la version 4.1.0).

Parmi ces améliorations, on notera :

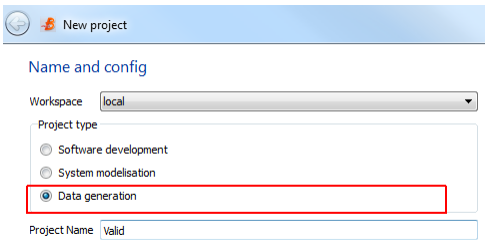
- Ajout du type de projet « validation de données »
- Support amélioré des entiers : choix de la valeur maximale implémentable (MAXINT) sur 16, 32 ou 64 bits, les littéraux entiers ne sont plus limités (BIGINT).
- Support des nombres réels
- Support expérimental des nombres flottants
- Support des nombres littéraux en hexadécimal.
- Nouveau générateur de code C, C4B, avec profils de traduction et génération de makefile
- Localisation de l'interface graphique en Anglais, Français, Japonais et Portugais.
- Support de l'unicode. Les identificateurs, chaines de caractères et commentaires peuvent contenir des caractères non ASCII.
- Ajout d'une vue graphique pour le status des composants (points de vue preuve, obligations de preuve).
- Exécution de tâches de preuve multiples en parallèle : déclencher la même action de preuve plusieurs fois conduira à l'exécution en parallèles de ces actions. Des informations supplémentaires (ordre d'exécution des actions sont affichées sur la vue graphique).
- Amélioration de l'éditeur textuel intégré : Ctrl-D pour supprimer une ligne, meilleure indentation, ajout de la liste des fichiers ouverts récemment, paramètrage de la colorisation syntaxique, navigation dans le modèle (saut vers la définition, vers l'abstraction, vers le rafinement).
- Ajout d'une fonctionnalité de recherche dans tous les fichiers du projet, de type grep, autorisant l'emploi d'expression régulière
- Outil de preuve de règles : amélioration de la signalisation de l'état de vérification des règles
- Affichage des règles rechargées lors de l'utilisation de la commande « pc »
- Le prouveur interactif affiche désormais les branches de preuve en attente. Il est alors possible de savoir avec précision où l'on en est de la démonstration interactive.
- Génération des obligations de preuve de couverture et d'exclusivité : ces obligations de preuve permettent de démontrer la couverture d'un modèle événementiel, et d'affirmer ou d'infirmer qu'au plus un événement est ouvert dans chaque état d'un modèle événementiel.
- Outil de validation des règles mathématiques ajoutées par l'utilisateur. Cet outil propose une interface unique pour la gestion et la validation des règles mathématiques. Il est par ailleurs possible de saisir des démonstrations manuelles qui seront sauvegardées dans le corps du fichier pmm sous la forme de commentaires. L'outil fournit une vue synthétique par projet et par composant, ainsi que la possibilité de générer un rapport de validation.
- Correcteur orthographique pour les commentaires de l'éditeur de modèles B : les mots mal orthographiés sont identifiés. Un menu contextuel permet de choisir une correction parmi plusieurs possibilités. Les langages pré-installés sont l'Anglais (GB, US) et le Français (France, Belgique, Canada).
- Les règles mathématiques affichées dans la vue « Theory List » sont maintenant triées en fonction selon leur applicabilité (les gardes qui sont vérifiées sont affichées en gras)
- L'éditeur de modèles B indique désormais si le fichier en cours d'édition a été modifié par ailleurs.

Validation/génération des données[1]

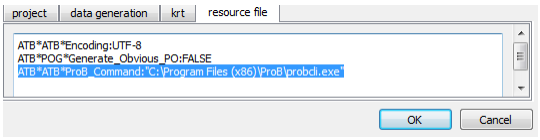
La validation de données consiste à vérifier qu'il est possible de valuer un ensemble de constantes en regard d'un ensemble de propriétés[2]. Cette validation s'appuie sur le model-checker ProB qui est maintenant interfacé avec l'Atelier B 4.1.

Pour pouvoir utiliser cette nouvelle fonctionnalité, il est nécessaire de:

- avoir installé ProB[3]
- avoir créé un projet de type « validation de données »



- avoir ajouté une ressource à ce projet, afin de pouvoir utiliser ProB pour réaliser la validation de données



La génération de données s'applique à toute machine de constantes (ou machine de contexte) grâce à la commande « Generate Data ».

Si la génération de données se termine avec succès, le résultat du calcul de ProB se trouve dans le répertoire <project translate directory>.

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

Le fichier <machine>.probcst contient les valeurs calculées pour les constantes.

Par exemple, pour la machine :

```
MACHINE
    M0

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

On obtient toutes les valeurs possibles pour C0:

```
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] Ces travaux ont été réalisés grâce au support de Alstom Transport Information Solutions – contrat 04-4550001418.

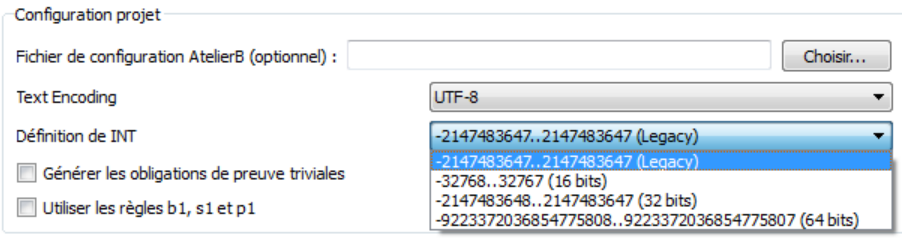
[2] http://www.methode-b.com/2012/11/validation-generation-de-donnees/

[3] http://www.stups.uni-duesseldorf.de/ProB/index.php5/The_ProB_Animator_and_Model_Checker

Support amélioré des entiers

- Paramétrage de INT

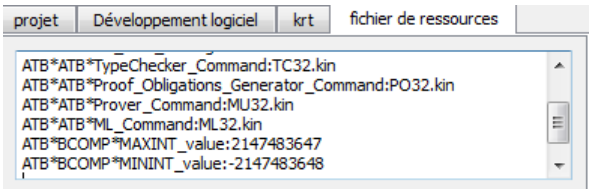
Lors de la création d'un projet, la taille des entiers implémentables est désormais paramétrable (16, 32 ou 64 bits).



L'ensemble des entiers implémentables est défini par INT = MININT .. MAXINT. La valeur des constantes prédéfinies MININT et MAXINT sont ainsi modifées selon ce choix.

Ce choix est concrétisé dans le fichier de ressources AtelierB du projet, au travers des ressources définissant :

- Les binaires des outils de vérification de type, génération d'obligations de preuves et de preuve (*.kin). Ces fichiers binaires diffèrent selon le nombre de bits utilisés pour les entiers TC16.kin, TC32.kin, TC64.kin par exemple pour le vérificateur de type).
- Les valeurs MININT_value et MAXINT_value.



Pour modifier ce paramétrage une fois le projet créé, il est nécessaire de :

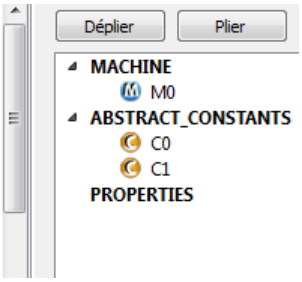
- modifier les noms des 4 fichiers binaires,
- modifier les valeurs de MININT_value et MAXINT_value,

en conformité avec le nombre de bits retenus. Il est d'autre part nécessaire de regénérer complètement toutes les obligations de preuve, du fait de la valuation différente de MININT et MAXINT.

- Support des entiers de grande taille (BigInt)

Les entiers litéraux ne sont plus limités à l'ensmble INT. Dorénavant, ils peuvent être utilisés en modélisation et en preuve, sans limite de taille, grâce à la librairie 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
```



Support expérimental des nombres réels[4]

Le type REAL a été introduit dans le langage B, dans l'objectif de pouvoir spécifier à terme des algorithmes manipulant des données réelles et implémentées avec des nombres flottants.

```
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 ;
```

Le type REAL est un type de base language qui complète BOOL, INTEGER et 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
    ) &
```

Les opérateurs arithmétiques +, -, * sont conservés car les propriétés de ces opérateurs sont les mêmes pour les entiers et les réels. La division réelle a une sémantique différente et reçoit un opérateur spécifique : rdiv.
Les opérateurs de comparaison <=, >= sont conservés.
Les littéraux réels, qui doivent être distinguables syntaxiquement des littéraux entiers de manière à assurer le contrôle de type, s'écrivent avec un «.». Ainsi 1.0 est le réel mathématiquement égal à l'entier 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 = 6335439.32729451239747469664245 03 &
    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
```

De nouvelles règles de typage ont été ajoutées :

- dans les prédicats x<=y et x>=y, les expressions x et y doivent être du même type qui peut être INTEGER ou REAL ;
- les réels littéraux sont de type REAL ;
- dans les expressions x+y, x-y, -x, x*y, les expressions x et y doivent être du même type qui peut être INTEGER ou REAL ;

[4] Ces travaux ont été réalisés grâce au support de SAGEM Défense Sécurité – contrat SK-0000443958-02.

- dans l'expression x rdiv y, les expressions x et y doivent être de type REAL ;
- dans l'expression x ** y, l'expression x doit être de type INTEGER ou REAL et l'expression y doit être de type INTEGER ;
- dans les expressions x mod y, succ(x) et pred(x), les expressions x et y doivent être de type INTEGER ;
- dans les expressions max(E), min(E), l'expression E doit être de type POW(INTEGER) ou POW(REAL) ;
- dans les expressions SIGMA(x).(P | E) et PI(x).(P | E), l'expression E doit être de type INTEGER ou REAL ;
- dans l'expression x..y, les expressions x et y doivent être de type INTEGER.

De nouvelles conditions de bonne définition ont été ajoutées :

- la bonne définition de x rdiv y est y /= 0.0 ;
- la bonne définition de min(x) et max(x) est x : FIN(x).

L'Atelier B 4.1 supporte la vérification de type et la génération des obligations de preuve des REAL.

⚠ Le support de la preuve pour les nombres réels doit être amélioré avant de pouvoir être véritablement utilisable.

Support expérimental des nombres flottants[5]

Le type FLOAT a été introduit dans le langage B, dans l'objectif de pouvoir spécifier à terme des algorithmes manipulant des données en nombres flottants, les prouver et générer le code correspondant, en conformité avec la norme IEEE 754.

```
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
```

L'ajout des flottants nécessite des modifications plus importantes du langage, en effet, ces nouveaux objets ont des propriétés particulières qui n'ont rien de commun avec celles usuellement utilisées dans le langage. Nous sommes donc obligés d'ajouter des opérateurs spécifiques qui ne sont pas connus de la base de règles de manière à préserver son intégrité sans refaire une validation complète. Seule l'égalité peut être conservée. En effet dans le langage B, l'égalité est déjà surchargée pour les ensembles et les entiers, nous avons donc décidé de l'étendre aux réels et aussi aux flottants.
Nous avons choisi de ne pas « encoder » les flottants dans l'Atelier B mais de les garder comme des objets « neutres » qui ne seront jamais calculés mais dont les propriétés seront donnés via la base de règles.
Le choix, syntaxiquement, a été de suffixeré tous les opérateurs arithmétiques usuels avec un « . ».
Ainsi on ajoute quatre opérateurs de comparaison : « <=. », « <. », « >=. », « >. ».
On construit des expressions flottantes avec les opérateurs : « +. », « -. », « *. », « /. ».
On ajoute aussi les équivalents à min, max SIGMA et PI qui sont minf, maxf, SIGMAF et PIF.

Nous avons choisi de ne pas fournir de moyen d'écrire des littéraux flottants. En cas de besoin, il sera toujours possible de définir des constantes valuées dans des machines de base. Ce choix provient du fait que l'on ne veut pas « coder » les flottants dans l'Atelier B mais seulement modéliser leurs propriétés à l'aide de règles de preuve.

De nouvelles règles de typage ont été ajoutées :

- dans les prédicats x<=.y, x<.y, x>=.y et x>.y, les expressions x et y doivent être de type FLOAT
- dans les expressions x+.y, x-.y, -.x, x*.y, x/.y, les expressions x et y doivent être de type FLOAT, l'expression résultante est de type FLOAT.
- dans l'expression x **. y, l'expression x doit être de type FLOAT et l'expression y doit être de type INTEGER. , l'expression résultante est de type FLOAT
- dans les expressions maxf(E), minf(E), l'expression E doit être de type POW(FLOAT) , l'expression résultante est de type FLOAT.
- dans les expressions SIGMAF(x).(P | E) et PIF(x).(P | E), l'expression E doit être de type FLOAT, l'expression résultante est de type FLOAT.
- dans l'expression realf(x), x doit être de type FLOAT, l'expression résultante est de type REAL.

De nouvelles conditions de bonne définition ont été ajoutées :

- la bonne définition de a/.b est realf(b) /= 0.0 / : {PLUSZEROF, MOINSZEROF}.
- la bonne définition de a **. b est b : NATURAL.
- la bonne définition de minf(a) et maxf(a) est a:FIN(a).

[5] Ces travaux ont été réalisés grâce au support de SAGEM Défense Sécurité – contrat SK-0000443958-02.

L'Atelier B 4.1 supporte la vérification de type et la génération des obligations de preuve des FLOAT.

- ⚠ Il n'y a à ce jour aucun support pour la preuve avec des nombres flottants. Le projet ANR BWare[6] devrait pouvoir combler ces manques.
- ⚠ Il n'y a à ce jour aucun support pour la génération de code supportant les nombres flottants. Le support de la preuve devra être suffisant avoir de pouvoir envisager cette fonctionnalité.

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

Support des nombres littéraux en hexadécimal[7]

Les littéraux hexadécimaux sont utilisables en modélisation et en preuve, en utilisant le préfixe usuel 0x suivi d'une représentation en base 16 (0..9, A..F ou a..f). Les littéraux hexadécimaux sont de type entier. Combinée au support des entiers de grande taille, cette amélioration permet de manipuler des littéraux hexadécimaux de grande taille.

```
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
```

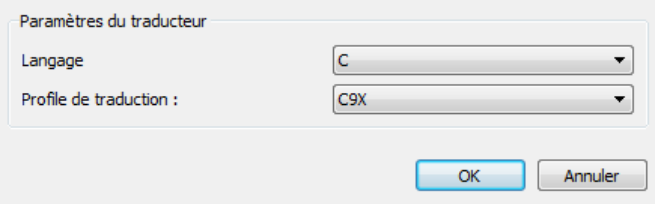
[7] Ces travaux ont été réalisés grâce au support de ST Microelectronics – convention 2008_0784.

Nouveau générateur de code C, C4B, avec profils de traduction et génération de makefile

En remplacement de ComenC, un nouveau générateur de code, C4B, a été développé sur la base du compilateur B. S'il ne supporte pas encore le renommage et les paramètres de machine abstraite, le langage B0 supporté est plus large que celui de ComenC. Quelques limitations connues sont listées plus bas.

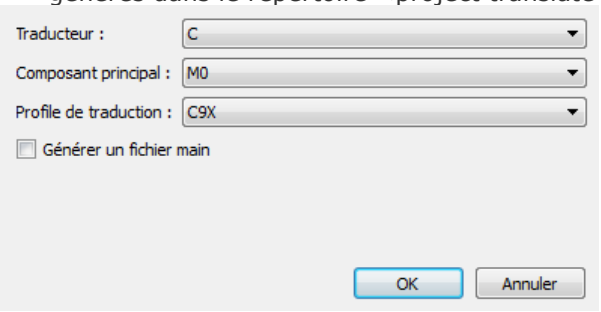
Le générateur de code peut être utilisé :

- en mode composant, pour traduire en C une implémentation. Il faut définir le profil de traduction parmi C9X, LIGHT, PROJECT. Les fichiers .c et .h de l'implémentation sélectionnée sont générés dans le répertoire <project translate directory>/c.

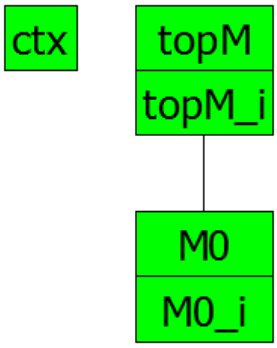


- en mode projet, pour traduire en C le projet complet. Il faut définir :
  - le nom du composant de plus haut niveau, à choisir dans la liste des implémentations du projet. Il doit s'agir d'un composant ne contenant qu'une seule opération sans paramètre.
  - le profil de traduction : C9X, LIGHT ou PROJECT.
  - Si un cmakefile doit être généré afin de faciliter la compilation du projet.

  Les fichiers .c et .h des implémentations du projet, ainsi que le fichier Cmakelist.txt[8] sont générés dans le répertoire <project translate directory>/c.



Exemple : pour le projet ci-dessous, on obtient le fichier Cmakelist.txt suivant :




```
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})
```

[8] L'outil cmake (http://www.cmake.org) permet d'automatiser le création d'un exécutable à partir d'un fichier de définition cmakelist.txt.

Le composant b_init.c réalise l'initialisation des différents composants du projet.
Le composant b_main est le composant chapeau : il déclenche l'initialisation des composants (b_init.c) puis les traitements définis dans l'unique opération du composant topM_i. La machine topM doit contenir une seule opération sans paramètre.

Les profils de traduction permettent de choisir la traduction de certains éléments :

Elément	Profil C9X	Profil Light	Profil 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

Le profil Project permet de choisir plus finement comment traduire chaque élément, par l'intermédiaire de ressources positionnées dans le fichier AtelierB.

⚠ La version 4.1.0 ne permet d'utiliser que le profil C9X.

Les limitations connues concernent la gestion des tableaux :
- Les constantes et les variables de type « tableau de T » doivent être définies avec le type : (0..MAX) --> T

où MAX est un littéral entier ou une constante concrète valuée avec un entier.

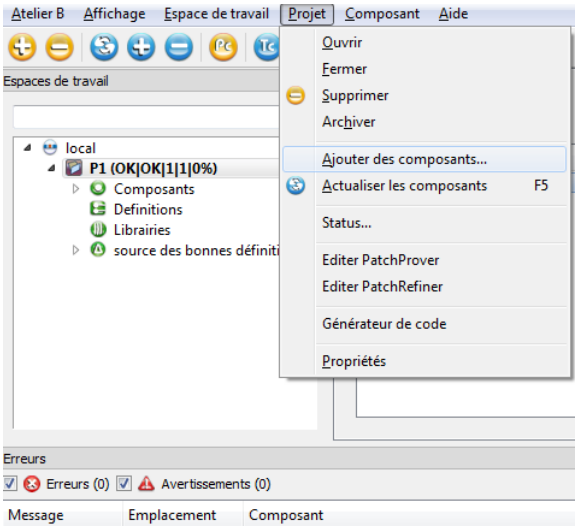
```
1- IMPLEMENTATION
2     M0_i
3- REFINES
4     M0
5- CONSTANTS
6     C0
7- PROPERTIES
8      C0: NAT
9- VALUES
10     C0 = 12
11- CONCRETE_VARIABLES
12     T1
13- INVARIANT
14     T1: (0..C0) --> INT
15- INITIALISATION
16     T1 := (0..C0) * {0xFF}
17+ OPERATIONS
20 END
21
```

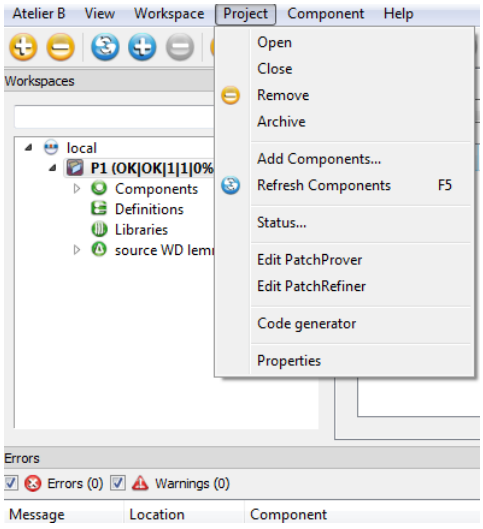
- Les tableaux de dimension 2 ou supérieure ne sont pas supportés.
- Les constantes concrètes définissant des types tableaux ne sont pas supportés (exemple : TT = 0..MM --> BOOL dans la clause PROPERTIES avec TT constante concrète).
- Les constantes tableau ne sont pas déclarées ou initialisées correctement lorsque les tableaux sont de type SS --> TT avec SS ensemble abstrait.

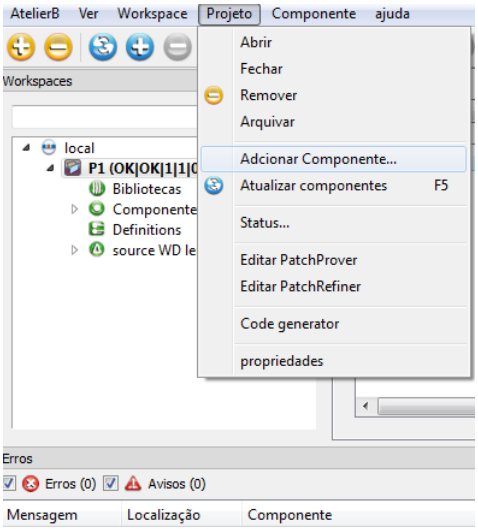
Localisation de l'interface graphique en Anglais, Français, Japonais[9] et Portugais[10]

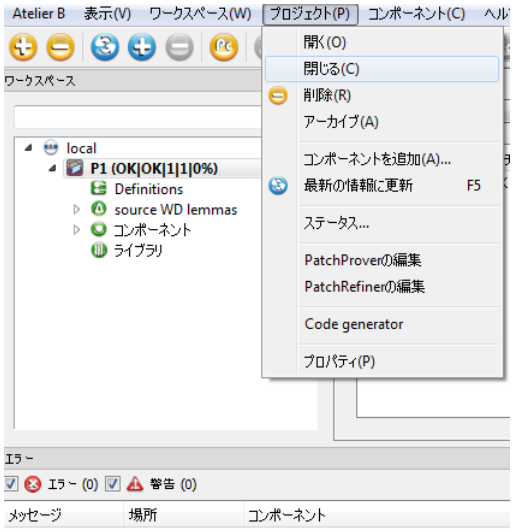
Le choix de la langue dépend de la localisation de l'ordinateur exécutant le logiciel Atelier B. Pour afficher une langue différente, il suffit de :

- modifier le choix dans le menu « Préférences/fenêtre principale/langage »
- modifier la variable d'environnement LANG (fr, en, br, ja) avant de lancer l'Atelier B.







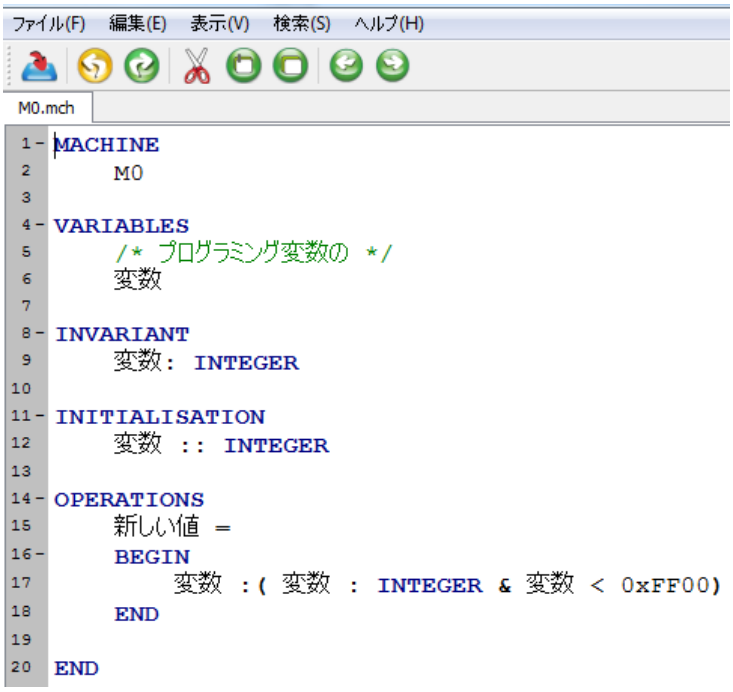


[9] La localisation en Japonais a été réalisée par Takuya Sawada (Hokkaido Electronics Corporation, Sapporo).

[10] La localisation en Portugais a été réalisée par Aryldo Russo Jr (Aes, Sao Paulo) et Haniel Herbosa (UFRN, Natal).

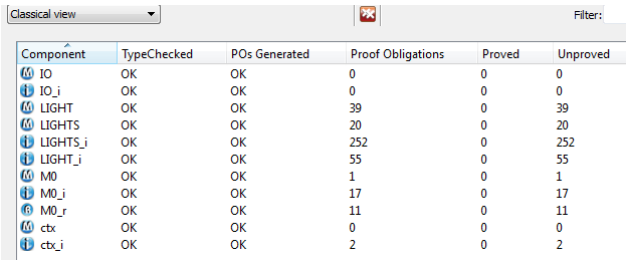
Support de l'Unicode.

Les identificateurs, chaines de caractères et commentaires peuvent contenir des caractères non ASCII.



Ajout d'une vue graphique pour le status des composants

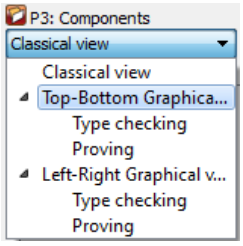
La vue composants de l'Atelier B liste les différents composants d'un projet, leur état (Typecheck, Génération d'Obligations de Preuve) ainsi que le nombre d'obligations de preuve (prouvées et à prouver).


Classical view

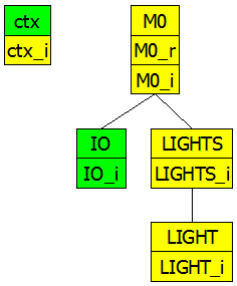
Filter:

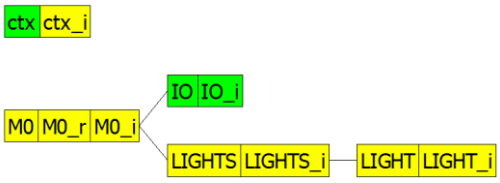
Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved
IO	OK	OK	0	0	0
IO_i	OK	OK	0	0	0
LIGHT	OK	OK	39	0	39
LIGHTS	OK	OK	20	0	20
LIGHTS_i	OK	OK	252	0	252
LIGHT_i	OK	OK	55	0	55
M0	OK	OK	1	0	1
M0_i	OK	OK	17	0	17
M0_r	OK	OK	11	0	11
ctx	OK	OK	0	0	0
ctx_i	OK	OK	2	0	2

Plusieurs vues graphiques ont été ajoutées afin de pouvoir apprécier certaines valeurs d'un seul coup d'œil. Le choix de la vue affichée est déterminé par le menu ci-dessous. On notera que la vue graphique peut être paramétrée pour un affichage du « haut vers le bas » ou de « la gauche vers la droite ».

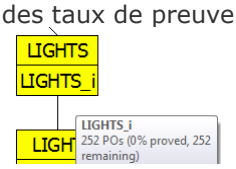


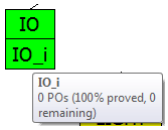
La vue graphique simple permet d'afficher l'état de preuve d'un projet. Les composants sont affichés selon leur ordre de dépendance.



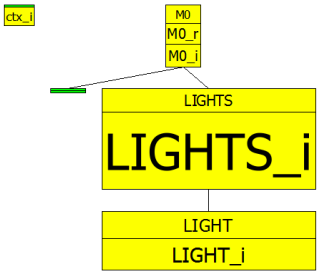


Leur couleur indique leur état de preuve : vert un taux de preuve de 100%, un dégradé du jaune au rouge pour des taux de preuve intermédiaires.





Les vues « typecheck » et « proving » permettent respectivement de dimensionner les boites des composants en fonction du nombre d'obligations de preuve.

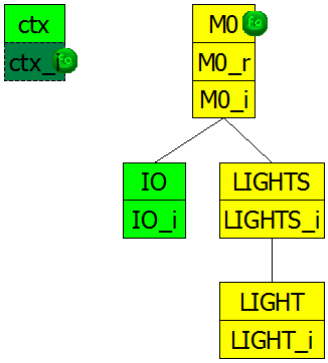


La vue graphique permet enfin de représenter la liste des tâches associées aux composants.
Si par exemple la liste des tâches contient ceci :

Tasks

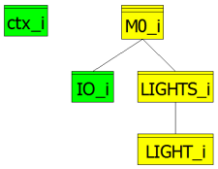
Project	Component	Action	Status	Messages	Server
P3	M0		Finished	End of Proof	localhost
P3	ctx_i		Finished	End of Proof	localhost

alors la vue graphique contient :



Une zone de saisie permet de filtrer les composants à afficher. Les composants dont le nom contient les caractères de la zone de saisie seront magnifiés.

Exemple :
La zone de saisie contient _i. Les implémentations sont mises en évidence.



Une zone de menu permet respectivement de :

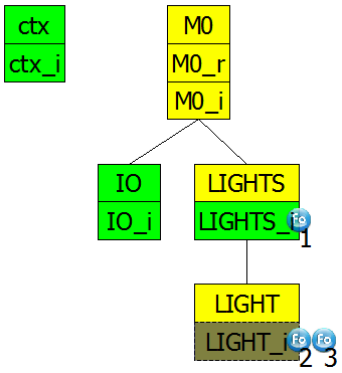
- supprimer de l'affichage toutes les tâches qui se sont terminées avec succès,
- zoomer sur le projet complet
- zoomer sur les composants filtrés
- zommer sur le composant sélectionné et ses descendants
- sélectionner un ensemble de composants
- imprimer la vue graphique
- exporter le graphe courant au format graphviz.

Exécution de tâches de preuve multiples en parallèle

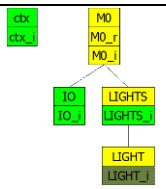
Il est désormais possible de lancer plusieurs tâches de preuve en parallèle sur un même composant. Il faut pour cela :

- avoir modifié la ressource « maximum running tasks » avec une valeur supérieure à 1 (menu *préférences* / *installation*) ;
- sélectionner un ou plusieurs composants et actionner plusieurs fois les boutons F0 ou F1.

La liste des tâches en cours s'affichera dans la liste des tâches. La vue graphique sera mise à jour en conséquence. Lorsqu'il y a plus de tâches demandées que de tâches exécutables, celles-là seront mises en attente et auront un numéro d'ordre qui apparaitra sur la vue graphique.



Ces numéros seront décrémentés au fur et à mesure de la terminaison des tâches en cours d'exécution.

	Si la case à cocher « Hide Finished Tasks » est cochée, seules les tâches en cours d'exécution sont affichées.	
	Sinon l'historique (le contenu de la fenêtre « tâches ») est affiché.