ARIMA épisciences

Langage et sémantique des expressions pour la synthèse de modèle Grafcet dans un environnement IDM

Gérard NZEBOP NDENOKA*[1], Maurice TCHUENTE[1], Emmanuel SIMEU[2]

[1]Département d'Informatique, Université de Yaoundé I, Cameroun
[2]Univ. Grenoble Alpes, CNRS, Grenoble Institute of Engineering, TIMA, 38000 Grenoble, France

*E-mail : ndenokag@yahoo.fr

DOI : 10.46298/arima.6452
Soumis le 6 mai 2020 - Publié le 27 octobre 2021
Volume : **33** - Année : **2020**
Numéro spécial : **Volume 33 - 2020 - Numéro spécial CRI 2019**
Éditeurs : *René Ndoundam, Eric Badouel, Maurice Tchuenté, Paulin Melatagia*

Résumé

Le GRAphe Fonctionnel de Commande Étapes Transitions (GRAFCET) est un puissant langage de modélisation graphique pour la spécification de contrôleurs dans des systèmes à événements discrets. Il fait usage des expressions pour exprimer les conditions de franchissement des transitions et des actions conditionnelles, ainsi que les expressions logiques et arithmétiques assignées aux actions stockées. Cependant, de nombreux travaux se sont penchés sur la transformation de spécifications Grafcet en code de contrôle pour systèmes embarqués. Pour faciliter l'édition de modèles Grafcet valides et la génération du code de contrôle, il est judicieux de proposer une formalisation du langage des expressions Grafcet, permettant de valider ses constructions et d'en pourvoir une sémantique appropriée. Pour cela, nous proposons une grammaire hors-contexte qui génère tout l'ensemble des expressions Grafcet. Nous proposons également un métamodèle et une sémantique associée des expressions Grafcet pour la mise en œuvre du langage Grafcet. Le parseur des expressions Grafcet (*G7Expr*) est obtenu grâce à l'outil ANTLR, alors que le métamodèle est mis en œuvre dans l'environnement d'Ingénierie Dirigée par les Modèles (IDM) Eclipse EMF. L'association des deux outils permet d'analyser et de construire automatiquement les expressions Grafcet lors de l'édition et la synthèse des modèles Grafcet.

Mots-Clés

expressions Grafcet ; grammaire hors-contexte ; analyseur syntaxique ; Ingénierie Dirigée par les Modèles ; métamodèle ; vérification de modèle ; sémantique des expressions

I INTRODUCTION

Le langage Grafcet (standard CEI 60848 [8]) est un puissant outil pour la spécification des systèmes de contrôle commande (SCCs) [1]. Il est à la base du SFC (*Sequential Function Chart*),

l'un des cinq langages du standard CEI 61131-3 de programmation des Automates Programmables Industriels, et intégré dans les environnements logiciels dédiés tels que *CoDeSys* de 3S-Smart Software, et plus récemment *EcoStruxure Control Expert* de Schneider Electric et *Step 7* de Siemens. Bien que le Grafcet soit un langage principalement graphique, il fait usage des expressions textuelles à différents niveaux pour exprimer les conditions et les calculs. Il s'agit essentiellement d'expressions arithmétiques et logiques auxquelles s'ajoutent des expressions temporelles et événementielles appelées front montant et front descendant [8] qui sont spécifiques au langage Grafcet. Les variables utilisées dans ces expressions sont reliées à l'état interne du modèle et des signaux d'entrée. La structure des expressions temporelles et événementielles doit respecter les règles de construction du langage pour être exploitable dans les étapes qui suivent la spécification fonctionnelle, au niveau du processus d'ingénierie des SCCs.

De nombreux travaux de recherche [4, 5, 10] se sont focalisés sur la transformation automatique de modèles de spécification Grafcet en code de contrôle. Il importe donc de disposer d'outils formels de spécification des expressions Grafcet pour permettre la vérification et la validation syntaxique, ainsi que la production d'une sémantique appropriée dans le code de contrôle qui découle du modèle Grafcet.

Les travaux réalisés dans [10] ont permis de proposer un framework pour la transformation du Grafcet en code PLC (CEI 61131–3). Ce framework garantit la validité du Grafcet au niveau des constructions structurelles (agencement étapes, transitions, actions, etc) sans s'intéresser aux expressions utilisées. F. Schumacher et al. ([9, 10]) ont étudié d'une part la transformation d'une spécification Grafcet en un réseau de Petri (RdP) [6] et d'autre part les préalables et obstacles à transformer une spécification Grafcet en un programme PLC (CEI 61131–3) [9]. Ces travaux se focalisent sur l'étude des trois types de conditions temporelles existantes et en proposent une sémantique formelle. Cependant, ils se limitent aux expressions temporelles construites autour d'expressions conditionnelles simples. En général, ces travaux ne considèrent pas la validité des expressions Grafcet. D'autres travaux [2, 4, 5, 12] ont concerné la transformation directe du Grafcet en code de contrôle (en langage C par exemple) [4]. Ici, aucun modèle de formalisation de l'ensemble des expressions Grafcet n'a été proposé afin de permettre leur validation et la production d'une sémantique sûre.

L'objectif dans cet article est de montrer l'intérêt que revêt les expressions Grafcet dans la mise en œuvre du langage Grafcet, puis de proposer des outils formels permettant de guider la construction d'expressions valides et leur intégration dans un environnement d'édition et de synthèse de modèles Grafcet.

Le reste du papier est organisé comme suit : la section II présente le langage de spécification Grafcet avec une analyse des expressions utilisées. La section III présente la grammaire algébrique proposée pour formaliser les expressions Grafcet, tandis que la section IV exploite cette formalisation pour définir les outils utiles pour la mise en œuvre dans un environnement d'ingénierie dirigée par les modèles (IDM), ainsi que la sémantique qui en découle. Enfin en section V, nous présentons une discussion relative à la contribution de ce travail, suivie d'une conclusion.

II LE LANGAGE DE SPÉCIFICATION GRAFCET

Le Grafcet (CEI 60848 Ed.3 [8]), est un langage graphique permettant de modéliser des systèmes séquentiels et les automatismes logiques. Il est utilisé pour la description détaillée du comportement de systèmes logiques séquentiels et s'inspire directement des réseaux de Petri

[1, 8]. Le Grafcet décrit les états d'un système et les actions associées permettant de prendre en compte les entrées nécessaires pour générer les sorties voulues. Il possède une syntaxe de description statique et une dynamique d'évolution.

2.1 Statique du Grafcet

La syntaxe de base du Grafcet est similaire à celle des RdP. Un modèle Grafcet (tel que présenté en Figure 1) est un graphe orienté avec deux types de nœuds : les étapes et les transitions.

Les étapes sont représentées par des carrés tandis que les transitions sont représentées par des traits horizontaux. Les étapes initiales définissent la situation initiale et sont représentées par des carrés ayant des lignes doubles. Les étapes et les transitions sont interconnectées par des arcs dirigés appelés connexions. Ces arcs relient nécessairement les étapes aux transitions et les transitions aux étapes. Les actions peuvent être associées à une étape pour agir sur la partie opérative du système via une variable de sortie. Une condition appelée réceptivité est associée à chaque transition. Il est aussi possible d'associer une condition à une action à niveau pour en faire une action conditionnelle.

2.2 Règles d'évolution du Grafcet

La situation du Grafcet évolue par franchissement successif des transitions selon cinq règles d'évolution [1, 8] visant à assurer un comportement déterministe, contrairement aux RdP qui évoluent en mode asynchrone, induisant ainsi des conflits et un comportement non-déterministe. Elles sont :

— **Règle 1 :** initialement, toutes les étapes initiales sont actives ; toutes les autres étapes sont inactives.
— **Règle 2 :** une transition est activée lorsque toutes les étapes qui précèdent immédiatement cette transition sont actives. Une transition est franchissable lorsqu'elle est activée et que la condition associée à cette transition est vraie. Une transition franchissable doit être immédiatement franchie.
— **Règle 3 :** le franchissement d'une transition provoque simultanément l'activation de toutes les étapes immédiatement suivantes et la désactivation de toutes les étapes immédiatement précédentes.
— **Règle 4 :** Lorsque plusieurs transitions sont franchissables simultanément, elles sont franchies simultanément.
— **Règle 5 :** lorsqu'une étape doit être à la fois activée et désactivée, en appliquant les règles d'évolution précédentes, elle est activée si elle était inactive ou reste active si elle était précédemment active.

Une étape définit un état partiel du système et peut être active ou inactive. Une variable booléenne X_i appelée variable d'activité d'étape est définie pour chaque étape i. La variable X_i est $vraie$ si l'étape i est active et $fausse$ sinon. L'état général d'un Grafcet, appelé sa situation, est caractérisé par le vecteur $X = (X_i)$. Les étapes initiales sont initialement activées. Sur l'occurrence d'événements externes, le changement de situation du grafcet caractérise l'évolution du système qu'il modélise. L'évolution de la situation se fait par franchissement des transitions. Si plusieurs transitions sont franchissables lors de la survenue d'un événement, elles sont toutes franchies simultanément. On passe alors d'un état (situation) à un autre, ce qui est à l'origine de l'activation de certaines actions et de la désactivation d'autres.

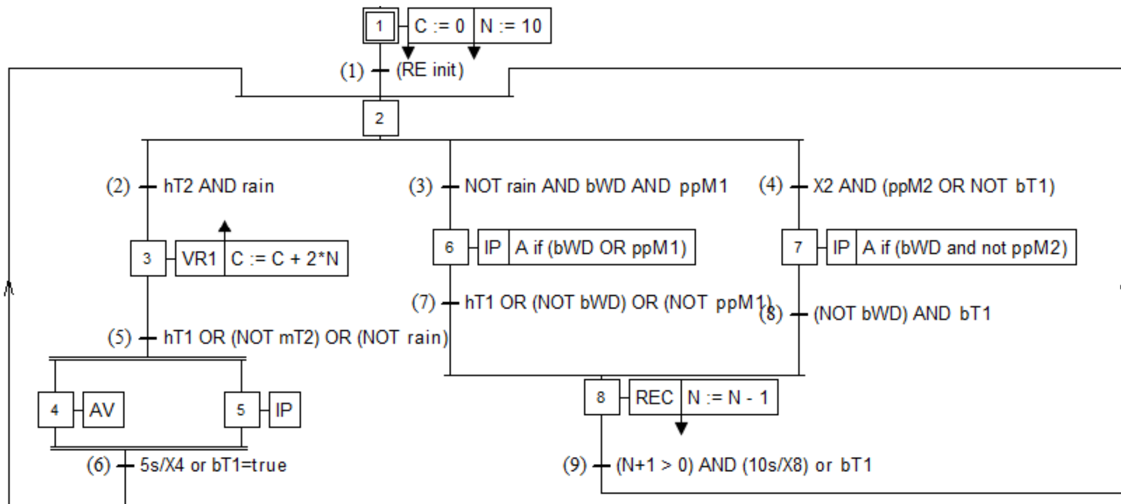


FIGURE 1 – Exemple de modèle Grafcet

Le modèle Grafcet présenté en Figure 1 est inspiré d'un Grafcet proposé par G. Nzebop N. et al. [12] pour modéliser un sous-système d'approvisionnement en eau d'un réservoir à partir d'une pompe immergée (IP). Ce modèle comporte huit étapes numérotées de 1 à 8 parmi lesquelles l'étape 1 est initiale, neuf transitions numérotées de (1) à (9) et plusieurs actions. $C := 0$ et $N := 10$ sont des actions stockées exécutées lors de la désactivation de l'étape 1 ; $VR1$, AV et REC sont des actions à niveau continues, associées respectivement aux étapes 3, 4 et 8. L'action *A if (bWD OR ppM1)* est une action à niveau conditionnelle. Elle est activée si l'étape 6 est active et la condition *bWD OR ppM1* a pour valeur logique *vrai*. La réceptivité de la transition (2) est *hT2 AND rain* ; elle exprime le fait que lorsque l'étape 2 est activée et que la valeur de *hT2 AND rain* est $vrai$, cette transition est franchissable et doit être immédiatement franchie. Lorsqu'elle est franchie, l'étape 2 est désactivée et l'étape 3 est activée. Ici, $HT2$ et $rain$ sont deux variables booléennes modélisation des signaux d'entrée numériques.

2.3 Les expressions Grafcet

Les réceptivités des transitions et les conditions associées aux actions à niveau conditionnelles, ainsi que les expressions des actions stockées constituent les expressions Grafcet. Ces expressions utilisent des opérateurs arithmétiques et logiques ordinaires, les comparaisons, et peuvent faire intervenir des événements et des contraintes/conditions temporelles.

Les événements : Un événement est soit un front montant ↑, soit front descendant ↓. Le comportement dynamique du Grafcet étant caractérisé par sa nature événementielle, un événement matérialise le changement de la valeur d'une variable booléenne ou d'une combinaison de variables booléennes (fonction booléenne). Comme l'illustre la figure 2.a, le changement de la valeur de a de $false$ à $true$ est noté $\uparrow a$ et est appelé front montant de a, tandis que le front descendant de a est noté $\downarrow a$ et symbolise son passage de $true$ à $false$. à la place de a, on peut avoir une expression logique quelconque.

Les contraintes/conditions temporelles : La forme générale d'une contrainte temporelles est $t1/*/t2$ (que nous appelons *Delayed 2* [9]), où l'astérisque * est un espace réservé pour la variable booléenne d'origine associée aux événements qui déclenchent la temporisation, tandis que $t1$ et $t2$ sont des durées ($t1, t2 \geq 0$). Comme l'illustre la figure 2.b, une contrainte de temps pourrait par exemple être spécifiée par $3s/a/5s$. Cela signifie que l'expression temporelle $[3s/a/5s]$ change sa valeur de $false$ à $true$ (ce qui équivaut à l'événement $\uparrow [3s/a/5s]$) trois

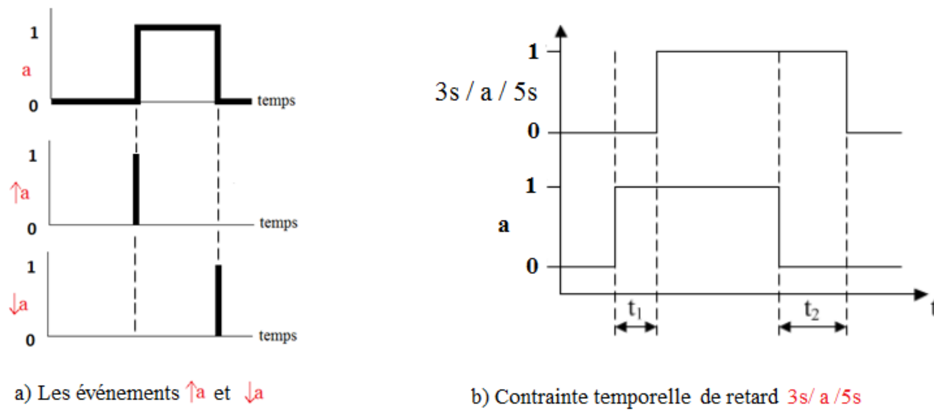


FIGURE 2 – Les événements et les conditions temporelles

secondes après que $\uparrow a$ se soit produit, et sa valeur passe de $true$ à $false$ cinq secondes après l'occurrence de l'événement $\downarrow a$.

Cet exemple illustre la relation entre les retards de temps spécifiques ($t1$ et $t2$) et les deux événements d'entrée ($\uparrow a$ et $\downarrow a$), définissant ainsi un intervalle de temps discret pendant lequel la variable a (de la contrainte temporelle $[3s/a/5s]$) a la valeur $true$:

$[t1..\Delta + t2] = ["3s$ après l'apparition de $\uparrow a"$ .. $"5s$ après l'apparition de $\downarrow a"]$, où Δ est la durée pendant laquelle la variable a est restée active, c'est-à-dire le temps qui sépare les instants d'occurrence des événements $\uparrow a$ et $\downarrow a$.

Il convient alors de déterminer l'état de la variable booléenne $[3s/a/5s]$ par un chronomètre qui observe l'intervalle de temps discret $[t1..\Delta + t2]$. Chaque fois qu'un événement se produit, tel que $\uparrow *"$ ou $\downarrow *$, le chronomètre est réinitialisé à zéro.

L'exemple de la figure 2.b est le cas le plus général d'une contrainte de temps dans le Grafcet. La norme CEI 60848 autorise également deux notations abrégées pouvant être considérées comme des cas particuliers de $t1/*/t2$:

— La notation $t1/*$. C'est le cas où $t2 = 0$. Elle spécifie un délai simple (que nous appelons *Delayed 1*).
— La notation $\neg(t1/*)$. Contrairement à un simple délai, il s'agit d'une limite de temps (que nous appelons *Limited*).
  La valeur de $[\neg(t1/*)]$ passe de $false$ à $true$ lorsque $\uparrow *$ survient et reste vraie pendant la durée $t1$. à l'apparition de l'événement $\uparrow [t1/*]$, la variable logique $[\neg(t1/*)]$ change sa valeur de $true$ à $false$.

Du point de vue des variables de temporisation, la contrainte temporelle peut être vue comme une sorte de chronomètre observant le changement d'état de la variable de temporisation. Il fournit des informations sur la durée écoulée depuis la dernière occurrence du front montant ou descendant de cette variable. Lorsque la durée écoulée est supérieure ou égale au temps de retard de la variable temporelle, la contrainte temporelle correspondante est $true$. En conséquence, il convient pour chaque variable Grafcet d'être caractérisée par son état (actif ou inactif) et sa durée de vie, la durée de vie étant le temps écoulé depuis son passage de $false$ à $true$.

III GRAMMAIRE DES EXPRESSIONS GRAFCET

Comme notre analyse repose sur la structure textuelle des expressions Grafcets on va s'appuyer sur les outils formels d'analyse des programmes, principalement les grammaires formelles telles que décrites dans le métalangage EBNF (*Extended Backus-Naur Form*). Ce métalangage définit une notation formelle permettant de décrire les règles syntaxiques des langages de programmation.

L'étude des expressions Grafcet nous permet de proposer la grammaire hors-contexte dont l'axiome est G7Expr, avec des règles décrites à la figure 3. Elle est obtenue en étendant les grammaires usuelles des expressions arithmétiques et logiques.

```
G7Expr →   G7Expr ('and'| 'AND'|'&&') G7Expr
           | G7Expr ('or'| 'OR'|'||') G7Expr
           | G7Expr ('==' |'=' | '!=' | '<>') G7Expr
           | G7Expr ('<=' |'>=' | '<' | '>') G7Expr
           | G7Expr ('+'| '-') G7Expr
           | G7Expr ('*'| '/') G7Expr
           | '(' G7Expr ')'
           | ('not'| '!') G7Expr
           | RE G7Expr
           | FE G7Expr
           | timeLogicG7Expr
           | Atomic
timeLogicG7Expr → '[' Number u '/' G7Expr ']'
                  | '[' ('not'| '!') Number u '/' G7Expr]
                  | '[' Number U '/' G7Expr '/' Number u ']'.
Atomic → Number | Id |BoolValue
```

FIGURE 3 – Grammaire des expressions Grafcet

Cette grammaire est récursive à gauche, avec des opérateurs ayant des niveaux de priorité différents. Elle est donc ambiguë en analyse $LL(1)$. Cependant, en analyse $LL(k)$, la gestion des conflits de choix des règles peut s'opérer en considérant l'ordre de priorité et l'associativité des opérateurs présents. Ces opérateurs sont donnés comme suit, classés du moins prioritaire au plus prioritaire [11] :

1. l'opérateur OU (OR ou bien $||$),
2. le ET (AND ou bien $\&\&$),
3. l'égalité ($=$) et la différence ($! =$ ou $\neq$),
4. les opérateurs de comparaison ($<=$, $>=$, $<$, $>$),
5. l'addition (+) et la soustraction ($-$),
6. la multiplication (*) et la division ($/$),
7. la négation NOT (! ou $\neg$), le front montant (*rising edge* : RE) et le front descendant (*falling edge* : FE).

Parmi les générateurs d'analyseurs syntaxiques sophistiqués de type $LL(k)$, il existe l'outil $ANTLR$ (*Another Tool for Language Recognition*) qui, depuis sa version 4 [3], offre la possibilité d'utiliser lors de la spécification de la grammaire l'ordre d'apparition des règles de production (en relation avec la priorité des opérateurs) pour gérer les conflits de choix de règles et désambigüiser les grammaires $LL(1)$. $ANTLR$ génère alors en sortie un analyseur syntaxique $LL(k)$ [11] dont le code peut être exprimé en langage java.

L'outil ANTLR est largement utilisé pour créer des langages, des outils et des frameworks [3]. Une spécification ANTLR de la grammaire des expressions Grafcet est donnée en figure 4.

```
33  //Grammar Production rules
34  myG7Expr: g7Expr ; // myG7Expr is the Axiom of the grammar (here is
        the first production)
35  g7Expr : op = (NOT|RE|FE) g7Expr _ #UnaryLogicOp
36        | left = g7Expr op = (LE|GE|LT|GT) right = g7Expr #LEcmp_G7Expr
          | left = g7Expr op = (EQUAL|DIFF) right = g7Expr #EqualDiff_G7Expr
37        | left = g7Expr op = (AND|OR) right = g7Expr  # AndOr_G7Expr
          | op = SUB g7Expr     #InfixMinus
38        | left = g7Expr op = (MULT|DIV) right = g7Expr  # MulDiv_G7Expr
          | left = g7Expr op = (ADD |SUB) right = g7Expr  # AddSub_G7Expr
44        | atomic        #primaryAtom
45        | timeLogicG7Expr     #primaryTiming
46        |'(' g7Expr ')'      #primaryParenthesis ;
47
48  timeLogicG7Expr :
49        '[' nb1 = Number unit1 = U '/' g7Expr '/' nb2 = Number unit2 U']
50        | '[' nb = Number unit = U '/' g7Expr']'    #timeLogicDelayed
51        | '[' op = NOT nb = Number unit = U '/' g7Expr']'  # timeLogicG7
53  atomic : Number  #AtomNumber
54        |Id   #AtomId
55        |BoolValue #AtomBool ;
```

FIGURE 4 – Implémentation ANTLR de la grammaire des expressions Grafcet

L'axiome de cette grammaire est *myG7Expr*. Les attributs (*op*, *left*, *right*, ...) sont ajoutées aux productions pour faciliter les processus de parcours d'arbre syntaxique. L'attribut *op* se réfère à la valeur de l'opérateur lors de l'application de certaines règles. Il en est de même de *left* et *right* qui référencent les expressions gauche et droite. Cette spécification ANTLR donne lieu à la génération du code java du parseur.

L'extension des variables de temporisation (tel que défini dans le standard CEI 60848) aux expressions/fonctions de temporisation dans les conditions temporelles augmente considérablement le pouvoir expressif du langage des expressions Grafcet. Ainsi, avec la notation $t1/*/t2$, * peut représenter non seulement une variable mais aussi une expression logique quelconque.

IV MÉTAMODÈLE ET SÉMANTIQUE DES EXPRESSIONS GRAFCET

Comme le Grafcet est un langage de modélisation, il est recommandé d'utiliser une approche d'Ingénierie Dirigée par les Modèles (IDM) pour sa mise en œuvre [9, 10]. L'IDM est en effet le domaine de l'ingénierie logicielle qui utilise des modèles et des transformations de modèles pour produire des artefacts logiciels. Dans ce paradigme, un système est représenté par un modèle, un modèle est conforme à son métamodèle, tandis que le métamodèle correspond à la syntaxe abstraite du langage de modélisation, spécifiée dans le formalisme défini par son méta-métamodèle (en langage MOF).

4.1 Métamodèle des expressions Grafcet

Nous proposons le métamodèle donné en figure 5. Il fait intervenir en son cœur le concept **`Expression`** qui utilise les relations et les attributs décrits comme suit : le type de l'expression peut être `Logical` ou `Arithmetic` (décrit par le type énuméré `ExprType`), une expression possède un nom, une valeur booléenne et une valeur arithmétique, ne pouvant être utilisées qu'exclusivement. L'attribut `boolValue` est utilisé en cas d'expression logique alors que `arithmValue` est utilisé pour des expressions arithmétiques.

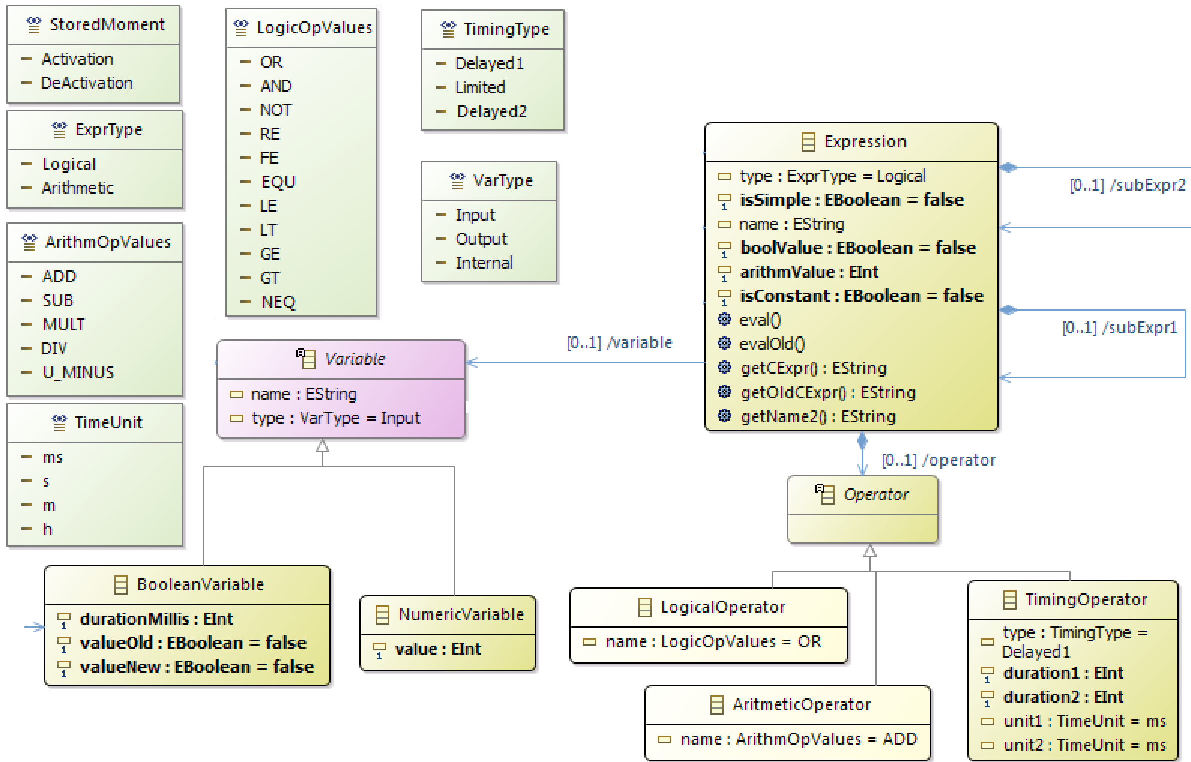


FIGURE 5 – Métamodèle des expressions Grafcet

La méthode `getCExpr()` est appelée pour obtenir la sémantique en langage C de l'expression, tandis que `getOldCExpr()` est appelée pour obtenir cette sémantique à l'instant précédent. La méthode `getName2()` est utilisée pour proposer un identificateur pour toute une expression, utile pour déclarer le compteur de temps associé. Une instance de **`Expression`** peut avoir deux sous expressions (`subExpr1` et `subExpr2`), auquel cas elles sont combinées par un opérateur (unaire ou binaire). Lorsque cette expression est simple (*isSimple* = *true*), alors il peut s'agir d'une variable (*isConstant=false*) ou bien d'une constante (*isConstant=true*) du Grafcet.

4.2 Sémantique C des expressions Grafcet

La sémantique C des expressions est implémentée dans les méthodes `getCExpr()` et `getOldCExpr()`. Pour les opérateurs ordinaires, elle reste inchangée. Par exemple, pour une expression composée par *AND*, la sémantique est `<anExpr.subExpr1.getCExpr()> && <anExpr.subExpr2.getCExpr()>`.

4.2.1 *Sémantique C des expressions Front montant et descendant*

Ces notions s'étendent de simples variables aux expressions. Pour chaque variable, on dispose de sa valeur actuelle et de sa valeur à l'instant précédent, ce qui permet de calculer la valeur d'une expression pendant ces deux instants. On a :

— `RE(anExpr)` est réalisée avec `!<anExpr.getOldCExpr()> && <anExpr.getCExpr()>`.

— `FE(anExpr)` est réalisée avec `<anExpr.getOldCExpr()> && ! <anExpr.getCExpr()>`.

4.2.2 *Sémantique C des expressions de conditions temporelles*

Initialement, `<anExpr.getName2()>` renvoie récursivement un identificateur approprié pour l'expression de temporisation. Pour chaque expression de temporisation, deux variables sont générées pour stocker la durée de l'expression à l'instant actuel et à l'instant précédent (`<anExpr.getName2()>_duration` et `<anExpr.getName2()>_duration_Old`). Ces durées doivent être mises à jour dans une section du code exécutée périodiquement (un *timer* par exemple).

À partir de l'analyse des variables de temporisation faite par F. Schumacher et al. [9] pour proposer un moyen d'évaluer les expressions temporelles basées sur une expression simple (une variable), la sémantique formelle proposée peut être étendue à toute condition temporelle basée sur n'importe quelle expression logique :

— **Condition de délai simple (*type Delayed1*)** : `[t/expression]` est vraie si le temps écoulé depuis l'occurrence de l'événement `RE(<expression.subExpr2>)` est supérieur à t. Alors `[t/expression]` a pour sémantique C : `<anExpr.getName2()_duration > t` .

— **Condition limitée dans le temps (*type Limited*)** : $\overline{t/expression}$ ou bien `[not(t/expression)]` est vraie si le temps écoulé depuis l'occurrence de l'événement `RE(<expression.subExpr2>)` est inférieur ou égal à t. Alors `[not(t/expression)]` a pour sémantique C : `<anExpr.getName2()>_duration <= t`
Pour les expressions temporelles de type $Delayed1$ ou $Limited$, la mise à jour de la durée d'activité est calculée par le listing 1.

Listing 1 – Mise à jour de la valeur de <expr.getName2()>_duration

```
1 if(FE(anExpr.subExpr2)) { <anExpr.getName2()>_duration = 0;}
2 else { <anExpr.getName2()>_duration ++;}
```

— **Condition de délai sous forme générale (*type Delayed2*)** : Sa syntaxe est `[t1/expression/t2]` alors que sa valeur est définie par :

  — `[t1/expression/t2]` passe de $false$ à $true$ après un temps écoulé de $t1$ depuis l'occurrence de l'événement `RE(<expression.subExpr2>)`.

  — `[t1/expression/t2]` passe de $true$ à $false$ après un temps écoulé de $t2$ depuis l'occurrence de l'événement `FE(<expression.subExpr2>)`. Alors, `[t1/expression/t2]` a pour sémantique C le listing 2, où la valeur de `<anExpr.getName2()>_duration` est calculée par le listing 3.

Listing 2 – Calcul de la condition t1/expression/t2

```
1 (<anExpr.getName2()>_duration > t1) && <anExpr.getCExpr()>)
2 ||
3 (<anExpr.getName2()>_duration > t2) && ! <anExpr.getCExpr()>)
```

Listing 3 – Mise à jour de <anExpr.getName2()>_duration

```
1    if( (! <anExpr.subExpr2.getOldCExpr()> && <anExpr.subExpr2.
        getCExpr()>)
2    || (<anExpr.subExpr2.getOldCExpr()> && ! <anExpr.subExpr2.
        getCExpr()>))
3    { <anExpr.getName2()>_duration = 0; } else { <anExpr.getName2()
        >_duration ++; }
```

Du listing 3, il ressort que la même variable <anExpr.getName2()>_duration est utilisée pour mesurer le temps écoulé depuis l'occurrence de l'un ou l'autre des événements, d'autant plus qu'ils s'alternent nécessairement : RE(expression) ou FE(expression).

4.3 Architecture du système d'analyse des expressions Grafcet

L'association du parseur avec le métamodèle des expressions Grafcet permet de réaliser un système qui dérive automatiquement les expressions des modèles Grafcet au sein d'éditeurs Grafcet. L'architecture d'une telle solution est présentée à la figure 6.

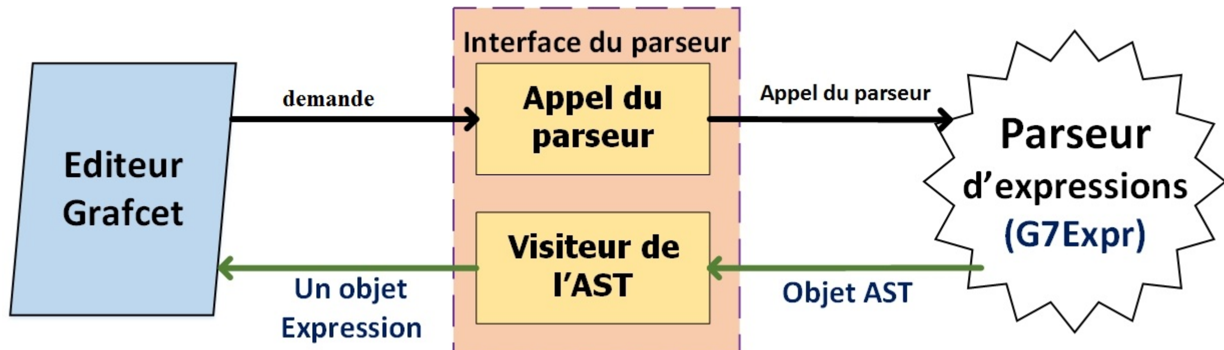


FIGURE 6 – Architecture du système parseur–métamodèle des expressions Grafcet

Une implémentation est faite au sein de la plateforme IDM Eclipse EMF, avec le code du métamodèle généré en java. Il en est de même du code de l'éditeur des expressions qui en découle, lequel est personnalisé pour permettre l'analyse des expressions par appel du parseur à travers l'interface réalisée.

4.4 Validation syntaxique et vérification sémantique

La validation syntaxique des expressions Grafcet est assurée par le parseur qui implémente la grammaire des expressions Grafcet. Lorsqu'une expression n'est pas valide, un message d'erreur est généré et affiché en console, avec des indications sur la zone concernée de l'expression. Comme l'implémentation du métamodèle est faite au sein d'Eclipse EMF, cette plateforme d'IDM offre des outils permettant la vérification sémantique des objets produits par le parseur des expressions Grafcet. Cette vérification a lieu au niveau de l'éditeur des expression Grafcet, à travers le métamodèle. Des règles sémantiques sont alors définies et implémentées en langage OCL (*Object Constraint Language* [7]), puis intégrées au métamodèle des expressions Grafcet. Toute expression produite par le parseur peut alors être vérifiée au regard des contraintes d'intégrité définies dans son contexte. Cette vérification sémantique garantit la bonne construction des objets par le parseur G7Expr et intervient avant la génération du code, qui constitue la sémantique C.

Comme illustration, « *tout opérateur binaire compose des opérandes du même type* » (*ValidExpressionWithBinaryOperation*) exprime une contrainte sémantique. Elle peut être formalisée en OCL par le listing 4 :

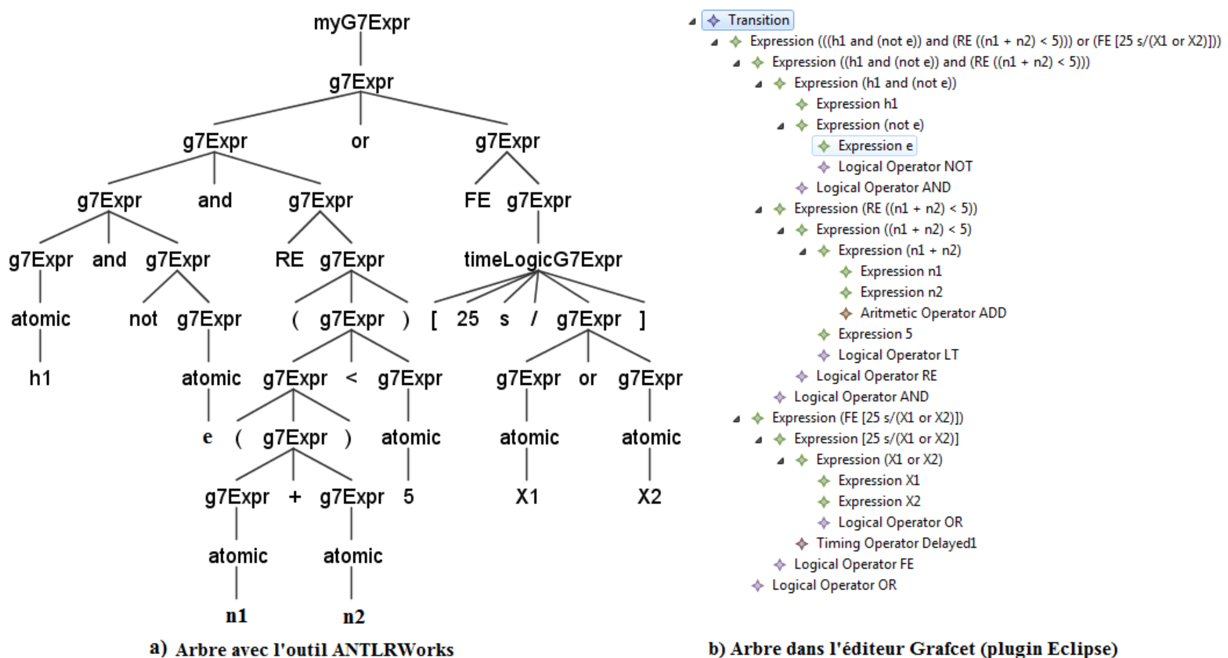
Listing 4 – Formalisation OCL de ValidExpressionWithBinaryOperation

```
1       (self.operator<>null and self.subExpr1<>null and self.subExpr2<>null)
            implies (self.subExpr1.type = self.subExpr2.type);
```

La validation sémantique est un problème NP-complet. Il faut donc énumérer les contraintes sémantiques, les formaliser et les intégrer au métamodèle des expressions pour permettre leur exécution lors de la validation sémantique des objets produits par le parseur *G7Expr*. En annexe A, nous présentons d'autres règles sémantiques énoncées et formalisées sur les expressions Grafcet.

4.5 Exemple

Considérons l'expression `(h1 and not e)and RE(n1+n2<5)or FE([25s/(X1 or X2)]))` représentant une condition dans un Grafcet. `[25s/(X1 or X2)]` est une condition temporelle tandis que `FE([25s/(X1 or X2)])` est une condition événementielle prenant en paramètre la condition temporelle. Son analyse par ce système produit le résultat présenté en figure 7, constituée d'un arbre de dérivation (figure 7.a) et de l'instance correspondante du métamodèle des expressions Grafcet (figure 7.b).



a) Arbre avec l'outil ANTLRWorks **b) Arbre dans l'éditeur Grafcet (plugin Eclipse)**

FIGURE 7 – Résultat d'analyse d'une expression Grafcet complexe

Cette expression contient deux sous-expressions composées par l'opérateur logique `OR`, et dont la première est composée par l'opérateur logique `AND`. La sémantique C qui en découle est donnée dans le listing **??**, où `d1` a pour valeur `25000/TIMER_PERIOD` (avec `TIMER_PERIOD` qui désigne la périodicité).

Listing 5 – Sémantique C de (h1 and not e) and RE(n1+n2<5) or FE([25s/(X1 or X2)

```
1 (h1 && (! e)) && ( !((n1_Old + n2_Old) < 5) && (n1 + n2) < 5)
2 || (X1_OR_X2_duration_Old >= d1) && !(X1_OR_X2_duration >= d1)
```

`X1_OR_X2_duration` est l'identificateur généré par appel de `getName2()` pour l'expression `X1 or X2` dont le rôle est de mesurer le temps depuis lequel cette expression est devenue *true*. La fonction exécutée périodiquement doit alors contenir le code de mise à jour de cet identificateur (listing 6).

Listing 6 – Mise à jour de X1_OR_X2_duration

```
1 if((X1_Old || X2_Old) && !(X1 || X2)){ X1_OR_X2_duration = 0; }
2 if((X1 || X2)){ X1_OR_X2_duration ++; }
```

V DISCUSSION ET CONCLUSION

5.1 Discussion

Les travaux réalisés par F. Schumacher [9, 10] visent la transformation du Grafcet en langage des PLCs. Ils se penchent sur l'étude de certaines expressions temporelles Grafcet, touchant des aspects ayant conduit à des ambiguïtés d'interprétation et contribuant à les lever. Cependant d'un point de vue synthèse, ils considèrent les expressions comme de simples chaines de caractères. Il en est de même de Bayó-Puxan et al [4].

Dans cet article, nous avons fait ressortir la particularité des expressions Grafcet par rapport aux expressions d'usage dans les langages de programmation, tout en proposant des modèles permettant de faciliter l'édition et la validation de modèles (programmes) Grafcet. On peut alors déceler des expressions non valides construites autour des éléments du modèle Grafcet.

L'implémentation de ce modèle permet la construction d'expressions événementielles basées sur des conditions composées, ainsi que la proposition d'une sémantique opérationnelle pour faciliter la mise en œuvre de ces expressions. Cette sémantique peut être exprimée aussi bien en langage C que dans tout autre langage de programmation des cibles embarquées.

5.2 Conclusion

Le langage Grafcet est très utilisé pour la spécification des systèmes de contrôle commande. On y retrouve des expressions (arithmétiques et logiques) faisant intervenir des événements et des notations temporelles, qui ne peuvent être évaluées ordinairement comme c'est le cas des expressions présentes dans les langages de programmation usuels. Dans cet article, nous avons proposé une grammaire hors-contexte simple et expressive, ainsi qu'un métamodèle équivalent dont l'édition des instances peut être associée à l'analyseur syntaxique obtenu, afin de faciliter l'analyse et la validation des expressions Grafcet. Une sémantique en langage C de ces expressions est aussi proposée. La grammaire et le métamodèle des expressions Grafcet proposés définissent un langage des expressions Grafcet plus expressif que celui présenté dans le standard CEI 60848. Le langage C est utilisé dans ce travail pour exprimer la sémantique des expressions Grafcet en raison de son importance dans le domaine des systèmes embarqués. Toutefois, il s'avère intéressant d'abstraire cette sémantique dans le but de diversifier les (langages) cibles envisageables.

RÉFÉRENCES

Publications

[1] R. DAVID. « Grafcet : A powerful tool for specification of logic controllers ». In : *IEEE Transactions on control systems technology* 3.3 (1995), pages 253-268.

[2] C. FERREIRA, S. MONTEIRO et J. MONTEIRO. « Automatic generation of C-code or PLD circuits under SFC graphical environment ». In : *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE International Symposium on*. Tome 1. IEEE. 1997, SS181-SS185.

[3] T. PARR. *The Definitive ANTLR Reference, Building Domain-Specific Languages, Pragmatic Bookshelf, Dallas Texas*. Rapport technique. ISBN 0-9787392-5-6, 2007.

[4] O. BAYÓ-PUXAN, J. RAFECAS-SABATÉ, O. GOMIS-BELLMUNT et J. BERGAS-JANÉ. « A GRAFCET-compiler methodology for C-programmed microcontrollers ». In : *Assembly Automation* 28.1 (2008), pages 55-60.

[5] J. MACHADO, E. SEABRA, J. C. CAMPOS, F. SOARES et C. P. LEÃO. « Safe controllers design for industrial automation systems ». In : *Computers & Industrial Engineering* 60.4 (2011), pages 635-653.

[6] F. SCHUMACHER et A. FAY. « Requirements and obstacles for the transformation of GRAFCET specifications into IEC 61131–3 PLC programs ». In : *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE. 2011, pages 1-4.

[7] J. CABOT et M. GOGOLLA. « Object constraint language (OCL) : a definitive guide ». In : *Formal methods for model-driven engineering*. Springer, 2012, pages 58-90.

[8] I. E. COMMISSION et al. *IEC 60848 : GRAFCET specification language for sequential function charts (3rd ed.)* Rapport technique. Tech. rep. International Electrotechnical Commission, 2012.

[9] F. SCHUMACHER et A. FAY. « Transforming time constraints of a GRAFCET graph into a suitable Petri net formalism ». In : *Industrial Technology (ICIT), 2013 IEEE International Conference on*. IEEE. 2013, pages 210-218.

[10] F. SCHUMACHER, S. SCHRÖCK et A. FAY. « Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code ». In : *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE. 2013, pages 1-4.

[11] L. BETTINI. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[12] G. N. NDENOKA, E. SIMEU et R. ALHAKIM. « Efficient controller synthesis of multi-energy systems for autonomous domestic water supply ». In : *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 24 (2017).

A ANNEXE

Pour faciliter la tâche de débogage lors de la validation des modèles construits dans l'éditeur, des contraintes spécifiques sont énoncées et formalisées en OCL selon le type d'opérateur en présence.

Expression avec opérateur unaire valide

ValidExprWithTimeOp : Une expression unaire modélise une variable temporelle, un opérateur logique ou la négation unaire. Ainsi, le membre gauche est nul, tandis que le membre droit est non nul. On a :

Listing 7 – ValidExprWithTimeOp

```
context Expression invariant ValidExprWithTimeOp:
(self.operator.oclIsTypeOf(TimingOperator) or (
self.operator.oclIsTypeOf(LogicalOperator)
and ( ((self.operator.oclAsType(LogicalOperator)).name = LogicOpValues::NOT
    )
or ((self.operator.oclAsType(LogicalOperator)).name = LogicOpValues::RE)
or ((self.operator.oclAsType(LogicalOperator)).name = LogicOpValues::FE)
))
or (self.operator.oclIsTypeOf(AritmeticOperator) and
(self.operator.oclAsType(AritmeticOperator)).name = ArithmOpValues::U_MINUS
    )
) implies (self.subExpr1=null and self.subExpr2<>null);
```

Expression constante et temporelle valides

AValidConstantExpression : une expression représentant une constante est valide si elle est simple et ne référence aucune variable. **SimpleNonConstExprIsOfVariableType** : une expression simple et non constante est du type de sa variable, soit logique, soit arithmétique. Leur formalisation OCL est données par :

Listing 8 – AValidConstantExpression et SimpleNonConstExprIsOfVariableType

```
context Expression invariant AValidConstantExpression:
self.isConstant implies (self.isSimple and self.variable = null);

context Expression invariant SimpleNonConstExprIsOfVariableType:
(self.isSimple and not self.isConstant and self.variable<>null
) implies(
(self.variable.oclIsTypeOf(BooleanVariable) implies self.type = ExprType::
    Logical)
or
(self.variable.oclIsTypeOf(NumericVariable) implies self.type =ExprType::
    Arithmetic) );
```

ValidExprWithTimeOp : Quand une expression concerne un opérateur temporel, elle est forcément logique. La formalisation OCL est :

Listing 9 – ValidExprWithTimeOp

```
context Expression invariant AValidConstantExpression:
self.operator.oclIsTypeOf(TimingOperator) implies self.subExpr1.oclIsTypeOf
    (LogicalOperator) and self.subExpr2.oclIsTypeOf(LogicalOperator);
```

Chaque fois que l'erreur `ValidExpressionWithBinaryOperation` se produit, elle est suivie de l'une des erreurs suivantes pour faciliter le débogage. Ces règles sémantiqies OCL concernent principalement la vérification de type.

Cas des opérateurs de comparaison

ValidComparisonOperator GT : Lorsqu'une expression concerne l'opérateur de comparaison GT (supérieur strict) , toutes les opérandes doivent être arithmétiques. Il en est de même de LT (inférieur strict), GE (supérieur ou égal) et LE (inférieur ou égal).

Listing 10 – ValidComparisonOperator_GT (supérieur strict)

```
1 invariant ValidComparisonOperator_GT:
2 (self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.oclAsType(
     LogicalOperator)).name = LogicOpValues::GT)
3        implies (self.subExpr1.type = ExprType::Arithmetic and self.subExpr2.
            type = ExprType::Arithmetic);
```

Cas des opérateurs logiques binaires (ET, OU)

ValidLogicOperator AND : Lorsqu'une expression concerne l'opérateur logique AND (ET logique), toutes les opérandes doivent être logiques.

Listing 11 – invariant ValidLogicOperator_AND (ET)

```
1 invariant ValidLogicOperator_AND:
2 (self.operator.oclIsTypeOf(LogicalOperator) and (self.operator.oclAsType(
     LogicalOperator)).name = LogicOpValues::AND)
3        implies (self.subExpr1.type = ExprType::Logical and self.subExpr2.
            type = ExprType::Logical);
```

Il en est de même de l'opérateur logique OR (OU logique).

Cas des opérateurs arithmétiques

ValidArithmeticOperator ADD : Lorsqu'une expression concerne l'opérateur arithmétique ADD (addition), toutes les opérandes doivent être arithmétiques.

Listing 12 – ValidArithmeticOperator_ADD(+)

```
1 invariant ValidArithmeticOperator_ADD:
2 (self.operator.oclIsTypeOf(AritmeticOperator) and (self.operator.oclAsType(
     AritmeticOperator)).name = ArithmOpValues::ADD)
3        implies (self.subExpr1.type = ExprType::Arithmetic and self.subExpr2.
            type = ExprType::Arithmetic);
```

Il en est de même de la soustraction, de la multiplication et de la division.