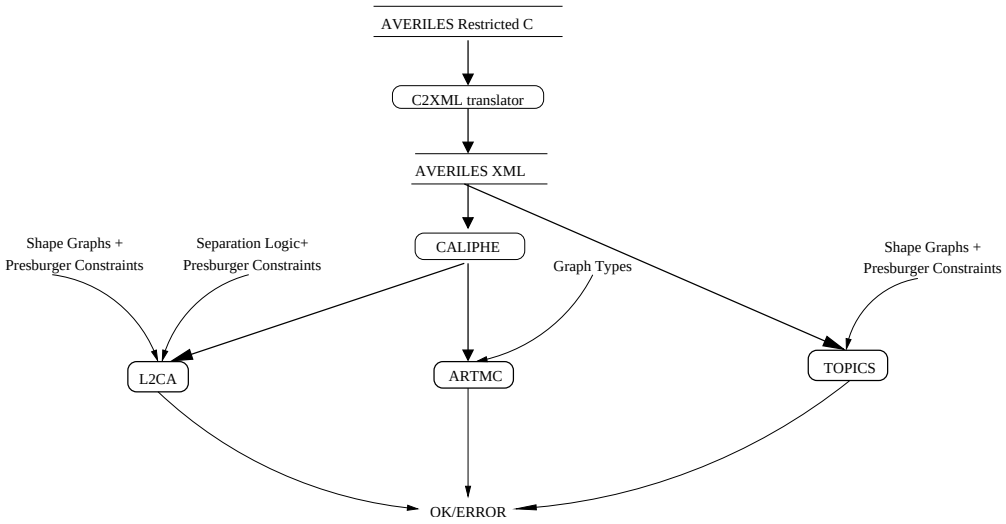
Projet RNTL AVERILES
Fourniture F1.2 : Extraction de Modèles

1 Organigramme de l´extraction de modèles

Le schéma ci-dessous explique la procédure de traduction d´un programme décrit dans le langage C restreint vers les trois outils d´analyse utilisées dans le cadre du projet AVERILES. L´entrée de chaque outil consiste dans une description du système à analyser, ainsi qu´une représentation symbolique de l´ensemble d´états initiaux. La plate-forme AVERILES permet l´unification des descriptions de systèmes, à travers son langage d´automates étendues. La spécification des états initiaux reste, à présent, spécifique a l´outil d´analyse choisi par l´utilisateur.



La plate-forme AVERILES prend en entrée un programme décrit en C restreint, ensuite le traduit sous forme d´un automate étendu. Cette traduction est automatique, et a été implémentée dans l´outil C2XML. Il est important de noter le fait que la traduction de C restreint vers automates étendus est sans perte d´information, l´automate résultant ayant exacte-

ment les mêmes comportements que le programme initial. Autrement dit, il existe une relation de bisimulation entre la sémantique du programme de départ et celle de l´automate.

La traduction à partir du langage des automates étendues vers les langages d´entrée des trois outils considérées a été implémentée dans la plateforme Caliphe, développée par CRIL Technologies. Notamment, on considère la traduction des automates étendues vers les langages des outils L2CA et ARTMC, car l´outil TOPICS prend directement en entrée un sous-ensemble du langage des automates étendues. Lorsqu´on considère des traduction vers des sous-ensembles strictes, on note le fait qu´une perte d´information dans la traduction est inévitable. Ceci est le cas, par exemple, lorsqu´on traduit un automate avec des opérations sur les tableaux vers un outil qui ne supporte pas les tableaux[1].

2 Traduction à partir du C restreint vers l´outil TOPICS

En ce qui concerne, l'outil TOPICS (qui est en cours de développement), les principales restrictions se font au niveau de la déclaration des types. L'outil ne prendra en compte que les programmes ne contenant qu'un seul type de structures de données de la forme **struct** *identifier* * qui pourra éventuellement être récursif. Les champs de cette structure seront soit des champs pointant vers des entiers, soit des champs pointant vers la structure elle-même. A noter qu'il ne pourra y avoir à la fois des champs pointant vers des entiers et des champ vers la structure. En ce qui concerne les déclarations de tableaux, il s'agira alors soit de tableaux d'entiers, soit de tableaux contenant des pointeurs vers la structure de donnée définie auparavant. Ces propriétés peuvent être décrites en modifiant la syntaxe du C restreint comme expliqué ci-dessous :

[1] A présent, aucun des outils considérés ne traite pas les opérations sur les tableaux. Le développement d´algorithmes et d´outils pour l´analyse de programmes manipulant des tableaux d´entiers est un sujet actuellement poursuivi par les membres du projet.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{type-declaration-list} \left\{ \textit{ declaration } \right\}^* \\
\textit{declaration} & ::= & \textit{var-declaration} \\
 & | & \textit{function-declaration} \\
 & | & \textit{function-definition} \\
\textit{type-declaration-list} & ::= & \left\{ \textit{ tab-declaration } \right\}^* \textit{ struct-declaration } \left\{ \textit{ tab-declaration } \right\}^* \\
\textit{struct-declaration} & ::= & \underline{\textbf{typedef}}\ \underline{\textbf{struct}}\ \textit{identifier}\ \underline{\textbf{\{}}\ \textit{struct-field-list}\ \underline{\textbf{\}}}\ \underline{\textbf{*}}\ \textit{identifier}\ \underline{\textbf{;}} \\
\textit{tab-declaration} & ::= & \underline{\textbf{typedef}}\ \textit{type-name}\ \underline{\textbf{*}}\ \textit{identifier}\ \underline{\textbf{;}} \\
\textit{struct-field-list} & ::= & \left\{ \textit{ struct-field } \right\}^* \\
 & | & \left\{ \textit{ int-field } \right\}^* \\
\textit{struct-field} & ::= & \underline{\textbf{struct}}\ \textit{identifier}\ \underline{\textbf{*}}\ \textit{identifier}\ \underline{\textbf{;}} \\
\textit{int-field} & ::= & \underline{\textbf{int}}\ \textit{identifier}\ \underline{\textbf{;}} \\
\textit{type-name} & ::= & \underline{\textbf{int}} \\
 & | & \textit{identifier} \\
\textit{var-declaration} & ::= & \textit{type-name identifier} \left\{ \underline{\textbf{,}}\ \textit{identifier} \right\}^* \underline{\textbf{;}} \\
\textit{function-declaration} & ::= & \textit{return-type identifier}\ \underline{\textbf{(}} \left[\textit{ fpar} \left\{ \underline{\textbf{,}}\ \textit{fpar} \right\}^* \right] \underline{\textbf{)}}\ \underline{\textbf{;}} \\
\textit{return-type} & ::= & \textit{type-name} \\
 & | & \underline{\textbf{void}} \\
\textit{fpar} & ::= & \textit{type-name identifier} \\
\textit{function-definition} & ::= & \textit{return-type identifier}\ \underline{\textbf{(}} \left[\textit{ fpar} \left\{ \underline{\textbf{,}}\ \textit{fpar} \right\}^* \right] \underline{\textbf{)}}\ \textit{block}
\end{array}
$$

Si un programme en entrée ne vérifie pas ces propriétés, l'outil TOPICS le signalera à l'utilisateur et arrêtera le traduction du programme.

3 Traduction à partir des automates étendues vers l´outil L2CA

Dans le cadre du projet RTNL AVERILES sont branchés sur la plate-forme Calife différents outils dont L2CA (“ List To Counter Automata ”) qui permet de transformer des programmes utilisant des listes simplement chaînées en automates de compteurs. Cet outil possède son propre langage d'entrée (cf. Annexes-9.2 BNF du langage de L2CA).

Pour utiliser l'outil L2CA, il est ainsi nécessaire de pouvoir exprimer l'automate, décrit pour la plate-forme Calife, dans le langage d'entrée de l'outil. Calife intègre donc un module permettant de transcrire l'automate dans le langage d'entrée de l'outil, de lancer l'outil et de récupérer la sortie résultante.

L'automate AVERILES est défini selon un flot de contrôle où les actions et les gardes sont décrites sur les transitions tandis que le langage d'entrée de L2CA est un langage impératif: les instructions sont exécutées les unes après

les autres. Pour pouvoir réaliser cette transformation, il est donc nécessaire de modéliser l'automate sous forme d'un graphe et de parcourir ce graphe pour transcrire l'automate vers le langage d'entrée de L2CA.

3.1 Mise en forme du graphe.

A partir de la représentation DOM (Document Object Model, W3C) de modèle de l'automate que Calife possède en mémoire, un graphe orienté représentant ce même automate est créé par le module de transformation en parcourant le fichier, en employant l'API java JAXP de gestion des fichiers XML.

Dans ce graphe sont représentés la liste des états, la liste des transitions ainsi que leurs paramètres respectifs. L'entrée pour L2CA sera déterminé en effectuant un parcours de ce graphe.

Le parcours du graphe a pour but d'identifier les différentes actions réalisées par l'automate et de les transcrire par des actions permises par le langage d'entrée L2CA :

Actions traitées par le langage L2CA (cf. Annexes 9.2):

- Affectations : lvalue = rvalue
- Branchements : if , while.

Une fois récupérée et parsée , chaque déclaration ou instruction sera insérée dans une liste. Pour obtenir le code, le contenu de la liste est écrit bout à bout dans un fichier texte.

3.2 Déclaration des variables

Les différentes variables sont décrites dans le fichier XML AVERILES au nœud : <Environment/Local/Var> où **@Name** représente le nom de la variable et **@Type** définit son type.

Le langage d'entrée L2CA nécessite une déclaration des variables. Il y aura donc un " matching " pour convertir les types d'AVERILES vers ceux du langage L2CA.

Ce " matching " est défini suivant le tableau suivant :

Type AVERILES @Type	**<Conversion L2CA>**
PointerVariables	pointer
IntegerVariables	int
ArrayVariables	XXX
IntegerSelectorVariables	XXX
PointerSelectorVariables	(Voir Chapitre 6)
ArraySelectorVariables	XXX

Les variables du type PointerSelectorVariables sont sauvegardées dans un tableau pour une utilisation ultérieure. En effet, elles correspondent à l'accès à une variable d'une structure. Le langage L2CA ne traitera que les variables du type PointerSelectorVariables qui représentent un pointeur sur une structure.

Dans le langage L2CA, chaque variable est définie de la façon suivante :
<Declaration " ::= <Type> <NomVariable>
<Type> ::= " int " | " pointer "
<NomVariable> ::= STRING

Le Code résultant sera donc, pour chaque variable n'étant pas de type " SelectorVariables ":

- <ConversionL2CA> @Name

Chaque déclaration de variable sera ajoutée à la liste contenant le code.

3.3 Transformation Graphe – Code

Pour définir le corps du programme L2CA à partir de l'automate AVERILES, le module de Calife effectue un parcours de graphe. On parcours le graphe état par état à partir de l'état initial et en effectuant le parcours et les opérations adéquates suivant le nombre d'états successeurs et prédécesseurs.

3.3.1 Initialisation

Variables globales :

List_Code : Liste contenant chaque déclaration du code L2CA

List_trans : Liste contenant l'ensemble des transitions restante à traiter

Pos_Code : Position à laquelle insérer la prochaine déclaration du code traité

Initialisation :

List_Code = ∅

Pos_code = 0

List_trans = ensemble de transition de l'automate

Execution :

- Recherche de l'état Initial.
- L2CAconversion (etat_initial)

3.3.2 Algorithme

Programme principal L2Caconversion(etat)

Si | Succ(etat)| = 0 **alors**

Retourner

Sinon

Si |pred(etat)| = 0 **alors**
Pour chaque succ(etat) **faire**
Une_transition(etat, succ(etat),false)
Fin Pour
Sinon
Pour chaque succ(etat) **faire**
Si Il existe chemin(succ(etat)→etat) **alors**
Une_transition(etat, succ(etat),true)
Fin Si
Fin Pour
Fin Si
Fin Si

Fonctions utilisées par le programme principal Une_transition(src, dst, isCycle)
Si garde ≠ true **alors**
Si isCycle = true **alors**
List_code.add(“ while ” + garde, Pos_code + “ do ”)
Sinon
List_code.add(“ if ”+garde, Pos_code + “ then ”)
Fin Si
Pos_Code ← Pos_Code + 1
Fin Si
List_Code.add(action, Pos_Code)
Pos_Code ← Pos_Code +1
List_Code.add(action, Pos_Code)
Pos_Code ← Pos_Code +1
List_Trans.remove(trans(src, dst))
L2CAConversion(dst)
Si garde ≠ true **alors**
Si isCycle = true **alors**
List_Code.add(“ od ”,pos_Code)
Sinon
List_Code.add(“ fi ”,pos_Code)
Fin Si
Pos_Code ← Pos_Code +1
Fin Si
Recherche_Chemin (Source, Destination)
Si Source == Destination
Return True
Sinon
Pour Chaque x <- succ(Source)
Recherche_Chemin(x,succ(Chemin))

Return false
FinPour
FinSi

3.3.3 Condition d'arrêt

Pour tout les états atteignables : nombre de successeurs = 0

3.4 Substitution des expressions

De la même manière que pour les déclarations, il est nécessaire d'adapter les mises à jour ou les gardes à la grammaire L2CA. Le module de transformation adaptera les différentes expressions des gardes et mises à jour par rapport au modèle suivant:

Langage AVERILES	**Langage L2CA**
p = malloc	p = new
p = p -> pointerSelectorVariables	p.next

Ces opérations sont réalisées grâce à la classe Matcher de Java. Lors de la création du graphe, à chaque ajout d'une nouvelle transition, les différentes déclarations de cette transition sont traitées par cette classe qui vérifie et établie une correspondance entre la description de l'automate AVERILES et le langage L2CA.

3.5 Gestion des exceptions

Le langage AVERILES ne peut être entièrement transformé vers le langage de L2CA, certaines expressions ne peuvent être traduites vers le langage L2CA donc certains automates AVERILES ne pourront pas être entièrement traduit. Le parseur enverra un avertissement à l'utilisateur lors de chaque problème rencontré et lui laissera la possibilité d'effectuer les modifications qu'il aura décidées.

Lors de la création du graphe, le module de transformation vérifie qu'il ne trouve pas de déclarations ou d'expressions non traduisibles selon le modèle ci-dessous

Déclarations non traduisibles :

- ArrayVariables.
- ArrayPointerSelectorVariables.
- IntegerSelectorVariables.

Expressions du langage AVERILES non traduisibles :

- free (<ArrayVariables>).
- free (<PointerVariables>).
- <ArrayVariables = Malloc(<IntegerVariables>).
- <ArrayVariables> [<iexpr>].
- <PointerVariable>→ IntegerSelectorVariables.
- <PointerVariable>→ ArraySelectorVariables .

3.6 Interfaçage à Calife

Le but de cette transformation est d'intégrer l'outil L2CA à la plate-forme Calife. Il faut donc définir dans Calife qu'une nouvelle transformation est implémentée, et à quel endroit se trouvent les fichiers de transformation de l'automate AVERILES vers L2CA. De plus, il est nécessaire de créer une interface intuitive pour l'utilisateur, lui permettant d'insérer les différents arguments de cette transformation.

3.6.1 Le model XML de l'automate AVERILES

Le modèle XML contient un nœud export permettant à Calife d'inclure et d'exécuter le bon programme. Dans ce nœud seront spécifiés le nom d'outil, son type d'action ainsi que la classe à utiliser pour effectuer la transformation.

<**Export** Label="L2CA" Type="Tools">
<**Property** Name="DestDir" Value="./Projet/${Projet}/L2CA"/>
<**JavaProjection**
Class="cril.calife.export.L2CA.L2CATranslation"
Jar="${CalifeDir}/Rules/l2ca.jar"
Output="${DestDir}/programL2CA"/>
< /**Export**>

Ainsi, la plate-forme Calife reconnaîtra automatiquement que l'automate AVERILES utilisé peut être lancé avec L2CA et place une aide visuelle pour lancer la transformation vers l'outil L2CA (cf. Chapitre 7.2).

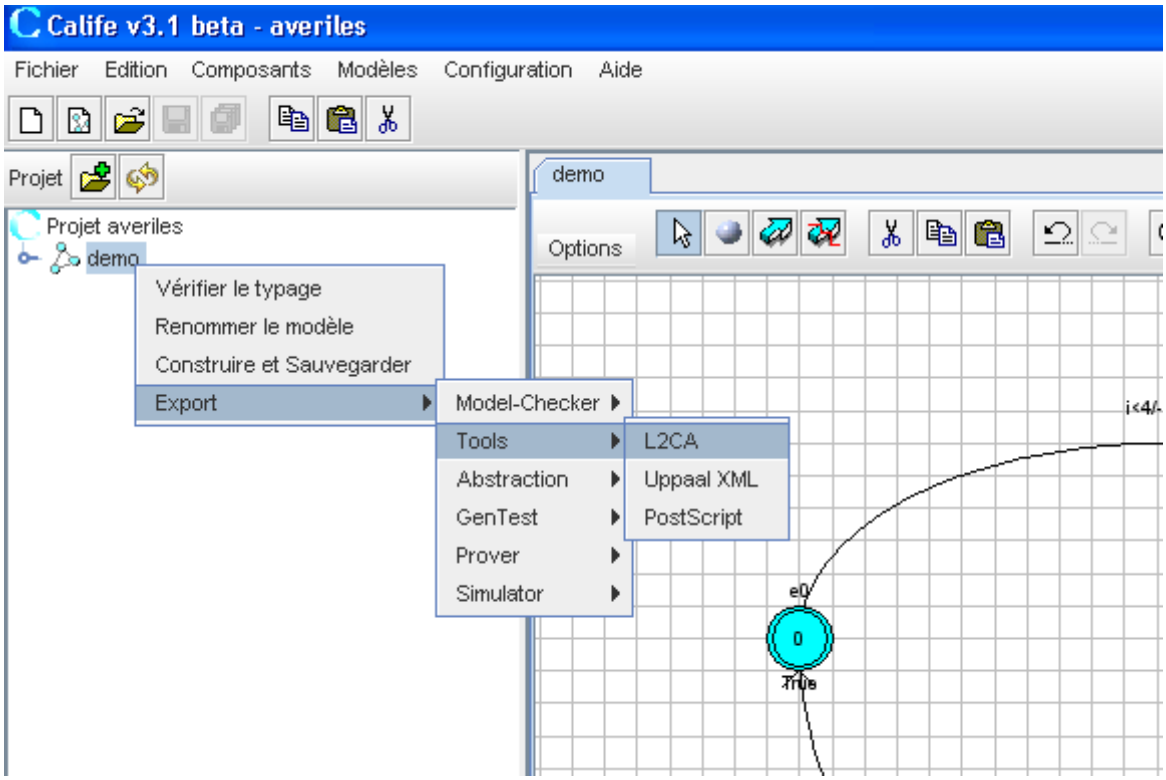
Dans Calife, les exports vers des outils implémentent l'interface " JavaProjection ", ce qui permet de lancer directement le module de transformation et de récupérer la représentation DOM de l'automate permettant la construction du graphe.

3.6.2 Exécution dans Calife

Pour lancer la transformation vers le langage d'entrée L2CA, il suffit d'effectuer les opérations suivantes:

- Clic droit sur le nom de l'automate dans la barre des projets
- Export
- Tools
- L2CA

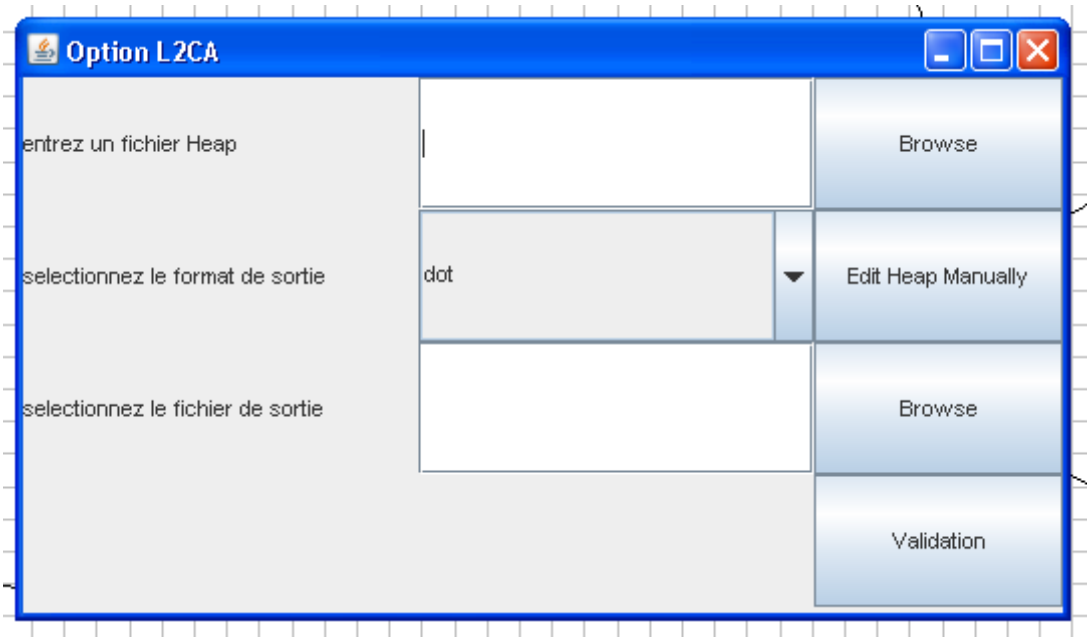
Calife crée alors un fichier représentant le programme d'entrée L2CA correspondant



Après avoir converti l'automate en langage L2CA, Calife ouvre une fenêtre permettant à l'utilisateur de donner les arguments nécessaires à l'utilisation de l'outil L2CA :

- Le fichier de tas (" heap ") : le fichier décrivant l'état initial de mémoire à partir duquel la vérification doit être effectuée.
- Le format de sortie (à choisir entre fast, dot et ARMC).
- Un fichier de sortie.

En appuyant sur le bouton " validation " l'utilisateur génère alors un fichier de sortie à l'emplacement souhaité.

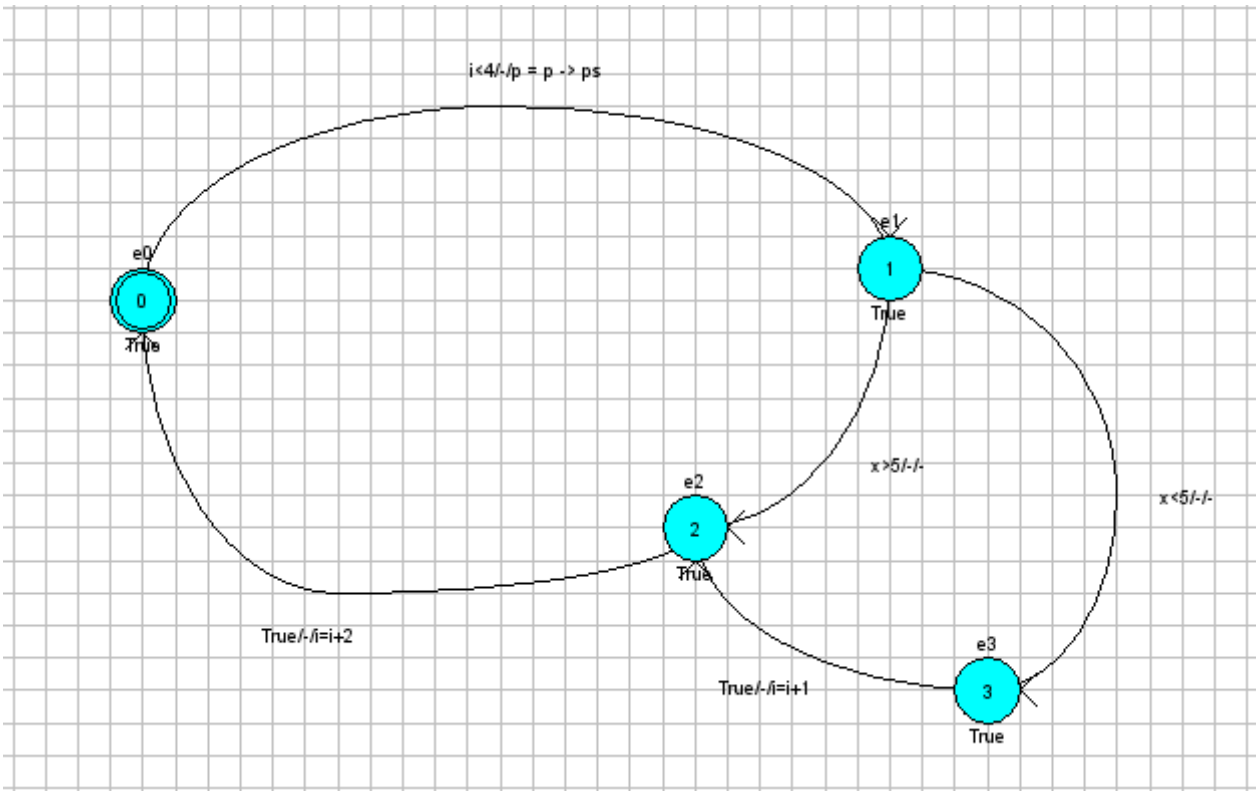


3.7 Exemple

3.7.1 Exemple d'automate

3.7.2 Code résultant

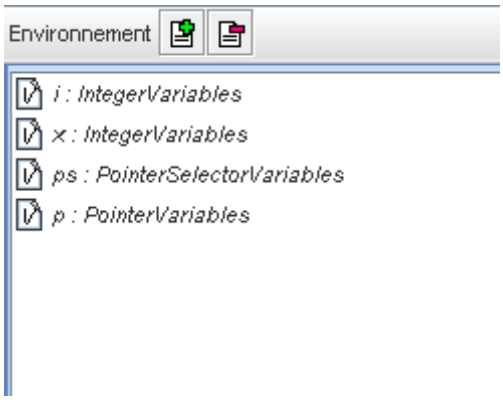
```
int i
    int x
    pointer p
    while i<4
    p = p -> ps;
    if x<5 then
    i=i+1;
    fi
    if x>5 then
    fi
    i=i+2 ;
    od
```



3.8 Annexes

3.8.1 Formalisme de l'automate AVERILES.

t,t',... ^{a} ArrayVariables
p,p'... ^{a} PointerVariables
i,i' ... ^{a} IntegerVariables
ps,ps'... ^{a} PointerSelectorVariables
is,is'... ^{a} IntegerSelectorVariables
as,as'... ^{a} ArraySelectorVariables
rule := guard ? action
guard := expr = expr | guard ν guard | guard Λ guard | +guard
action := lval := expr | t := malloc(iexpr) | p := malloc | free(p) | free(t)
lval := t | p | I | t[iexpr] | p→ps | p→is | p→as
iexpr := n ^{a} Z | i
expr := lval | n ^{a} Z | null | expr + expr | expr - expr



3.8.2 BNF du langage L2CA.

<Program> ::= <DeclarationPart> <StatementList>
<DeclarationPart> ::= <Declaration> | <Declaration> <DeclarationPart>
<Declaration> ::= <Type> <IdentifierList>
<Type> ::= ”int” | ”pointer”
<IdentifierList> ::= <Identifier> | <Identifier> ”,” <IdentifierList>
<StatementList> ::= <Statement> | <Statement> <StatementList>
<Statement> ::= <While> | <If> | <Assign>
<While> ::= ”while” <ConditionalExpression> ”do” <StatementList>
”od”
<If> ::= ”if” <ConditionalExpression> ”then” <StatementList> ”fi”
|”if” <ConditionalExpression> ”then” <StatementList> ”else” <StatmentList>
”fi”
<Assign> ::= <Variable> ”=” <Expression> ”;”
<ConditionalExpression> ::= ”!” <ConditionalExpression>
| <ConditionalExpression> ”&&” <ConditionalExpression>
| <ConditionalExpression> ”||” <ConditionalExpression>
| ”(” <ConditionalExpression> ”)”
| <RelationalExpression>
<RelationalExpression> ::= <Expression> ”<” <Expression>
| <Expression> ”>” <Expression>
| <Expression> ”<=” <Expression>
| <Expression> ”>=” <Expression>
| <Expression> ”==” <Expression>
| <Expression>
<Expression> ::= <Expression> ”+” <Expression>
| <Expression> ”-” <Expression>
| <Expression> ”*” <Expression>
| ”(” <Expression> ”)”
| <Term>
<Term> ::= <Variable>

| ”null”
| ”new”
| <Number>
<Variable> ::= <Identifier> | <Indentifier> ”.next”
<Identifier> ::= STRING
<Number> ::= POSITIVE INTEGER

4 Traduction à partir des automates étendues vers l´outil ARTMC

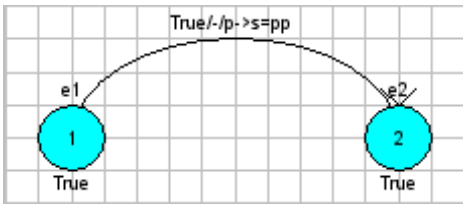
Les différents outil du projet RNTL/AVERILES (ARTMC et L2CA) sont intégrés à la plate-forme Calife.

Ce document décrit la transformation effectuée par la plate-forme Calife d'un automate “ AVERILES ” vers le langage d'entrée de l'outil ARTMC. Ce document ne décrit pas le langage d'entrée de l'outil. Il décrit la manière dont Calife génère une telle entrée à partir d'un automate “ AVERILES ”

Il s'agit de transformer un fichier XML représentant un automate “ AVERILES ” en un fichier. Ce fichier comprend, outre des affectations de variables, une liste de commandes codant le programme représenté par l'automate “ AVERILES ”. Un exemple est donné en annexe.

4.1 Parcours de l'automate

La transformation part de l'état initial de l'automate (la plate-forme Calife impose par ailleurs qu'il n'y en ait qu'un) et génère les lignes au fur et à mesure qu'elle franchit les transitions. Par exemple la transition suivante



où ‘p’ et ‘pp’ sont des pointeurs, et ‘s’ un sélecteur, est transformée en

(‘x.next=y’,‘num_line’,x,y,next,next_line)

Conformément au langage d'entrée de l'outil, ‘num_line’ est un identifiant binaire pour la ligne, ‘x’, ‘y’, ‘next’ des entiers codant les pointeurs et sélecteurs utilisés, et ‘next_line’ le numéro de la prochaine ligne à exécuter.

4.1.1 Liste des commandes acceptées par l'outil

Voici la liste des commandes du langage d'entrée de l'outil :

if x==null
if x==y
if x.data=="..."
if *
x:=new
x:=null
x:=y
x:=y.next
x.next:=new
x.next:=null
x.next:=y
x:=random_position
setdata
free
goto
exit

où 'x' et 'y' sont des variables de type pointeur, et 'next' et 'data' des sélecteurs.

Cette liste appelle le commentaire suivant. La syntaxe abstraite de l'automate AVERILES a un pouvoir d'expression plus grand. Aussi tout ce qui ne peut se traduire dans les commandes ci-dessus sera ignoré lors de la transformation, à savoir notamment :

- Les manipulations sur les entiers.
- Les manipulations sur les tableaux.
- Toute condition ne correspondant pas à l'une des trois conditions de la liste de commandes ci-dessus est transformée en 'if*'. C'est-à-dire que les gardes ne pouvant être traduites comme l'une de ces trois-là sont remplacées par de l'indéterminisme.

4.1.2 Traitement des sélecteurs

Dans un automate " AVERILES ", les variables utilisées sont déclarées. Les différents types de sélecteurs sont distingués lors de leur déclaration. Il est possible de déclarer des sélecteurs d'entiers, de pointeurs, et de tableaux. Il faut alors distinguer les transformations selon le type de sélecteur utilisé.

4.1.3 Transformations concernant les pointeurs et les sélecteurs

Trois cas de figure sont possibles :

1. l'instruction ne pose pas de problème,
2. l'instruction ne peut être transformée de manière précise, dans ce cas un message d'avertissement est lancé, et
3. l'instruction ne peut être traitée, auquel cas il faut arrêter la transformation.

Le tableau ci-dessous détaille les différents cas.

Instruction	Transformation	Instruction	Transformation
p = malloc	x=new		
p = null	x=null	p->ps = null	x.next=null
p = n (n in Z)	x=random	p->ps = n (n in Z)	x=random / x.next=y
p = ... + ...	x=random	p->ps = ... + ...	x=random
p = ... - ...	x=random	p->ps = ... - ...	x=random
p = t	erreur || x=null	p->ps = t	erreur || x=null
p = pp	x=y	p->ps = pp	x.next=y
p = i	x=random	p->ps = i	x=random / x.next=y
p = t[i]	x=random	p->ps = t[i]	x=random / x.next=y
p = p->ps	x=y.next	p->ps = p->ps	x=y.next / x.next=y
p = p->is	x=random	p->ps = p->is	x=random / x.next=y
p = p->as	erreur || x=null	p->ps = p->as	erreur || x=null
Instruction	**Transformation**	**Instruction**	**Transformation**
p->is = null	erreur	p->as = null	ignorer
p->is = n (n in Z)	ignorer	p->as = n (n in Z)	ignorer
p->is = ... + ...	ignorer	p->as = ... + ...	ignorer
p->is = ... - ...	ignorer	p->as = ... - ...	ignorer
p->is = t	erreur	p->as = t	ignorer
p->is = pp	erreur	p->as = pp	ignorer
p->is = i	ignorer	p->as = i	ignorer
p->is = t[i]	ignorer	p->as = t[i]	ignorer
p->is = p->ps	erreur	p->as = p->ps	ignorer
p->is = p->is	ignorer	p->as = p->is	ignorer
p->is = p->as	ignorer	p->as = p->as	ignorer

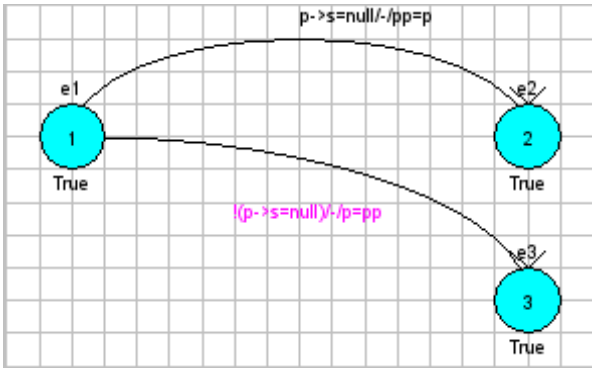
'p?' désigne un pointeur, 'i' un entier, 't' un tableau, 'ps' un sélecteur de pointeur, 'is' un sélecteur d'entier, et 'as' un sélecteur de tableau.

4.1.4 Ignorer une transition

" La transformation ignore la mise à jour de la transition " signifie que le programme ne fait rien et passe à la transition suivante. En l'occurrence, la transformation génère une ligne 'goto' et passe à la ligne suivante.

4.1.5 Gardes

La transformation suppose que différents branchements sont possibles si et seulement si ils possèdent des gardes différentes de 'True'. Toutes les transitions partant d'un état, si elles sont plusieurs, doivent posséder des gardes différentes de 'True'. La situation supposée impossible est qu'une transition partant d'un état X soit gardée par 'True' et qu'une autre transition parte du même état.



Exemple de gardes

En ce qui concerne la manière dont sont traitées les différentes gardes partant d'un état, la transformation génère une ligne pour chaque condition et termine par une ligne de commande 'exit'. Les blocs correspondant aux différents branchements sont générés à la suite.

Illustrons-le sur l'exemple de la figure ci-dessus. Celui-ci est transformé en séquentiel comme suit.

... ...
n **si** p->s = null **alors** goto n+3 **sinon** goto n+1
n+1 **si** * **alors** goto n+x **sinon** goto n+2
n+2 exit
n+3 ...
... ...
n+x ...
... ...

4.1.6 Etats finaux

Un état est considéré comme final si aucune flèche ne part de lui.

4.2 Traitement de l'automate " AVERILES "

4.2.1 Extension préalable de l'automate

Avant d'effectuer la transformation, deux choses sont effectuées.

La première est la réécriture de certaines lignes, afin d'augmenter ce que l'outil peut prendre en compte dans le programme modélisé.

La deuxième est la séparation des gardes des mises à jour dans les transitions où la garde est différente de 'True'. On peut voir un exemple de ce type de transition dans la figure ci-dessus. En effet, ces transitions comportent deux informations distinctes qui donneront lieu à deux lignes de commande différentes.

4.2.2 Réécriture

La transformation réécrit cependant certaines lignes, afin qu'elles ne soient pas ignorées. L'outil peut alors les prendre en compte.

```
p1->s1 = p2->s2;
```

se réécrit

```
tmp1 = p2->s2;
p1->s1 = tmp1;
if( p1->s1 == null )
```

se réécrit

```
tmp1 = p1->s1;
if( tmp1 == null )
if( p1->s1 == p2 )
```

et

```
if( p2 == p1->s1 )
```

se réécrivent

```
tmp1 = p1->s1;
if( tmp1 == p2 )
if(p1->s1 == p2->s2)
```

se réécrit

```
tmp1 = p1->s1;
tmp2 = p2->s2;
if( tmp1 == tmp2 )
```

où ‘p?’ et ‘tmp?’ sont des variables de type pointeur et ‘s?’ des sélecteurs de pointeurs.

Notons que les pointeurs sont nouveaux. La transformation les déclare donc parmi ceux utilisés par le programme. Cependant, le même pointeur suffit à chaque fois, puisqu'il ne sert qu'une fois. Il n'y a donc au maximum que deux nouveaux pointeurs.

4.2.3 Présupposés de l'outil

L'outils ARTMC suppose que les structures sont du type

struct xxx {
data = “ [01]* ”
struct xxx *0, *1, ...
}

où ‘data’ est un mot en binaire, et les autres champs (leur nombre n'est pas fixé) sont des next pointeurs sur le même type de structure.

4.2.4 Données

A l'heure actuelle, la transformation ne traite pas encore des données. Elle traduit alors les instructions de la forme ‘p->s=5’ en lignes de commandes ‘goto’ au lieu de ‘setdata’, et remplace les ‘ifdata’ par des ‘if*’.

4.2.5 Tableaux

Si des tableaux sont utilisés dans l'automate, Calife lance un avertissement disant que ceux-ci seront ignorés, mais effectue néanmoins la transformation.

En revanche, l'assignation d'un tableau à un pointeur peut poser problème. L'utilisateur peut choisir soit d'arrêter la transformation, soit d'assigner la valeur ‘null’ au pointeur considéré.

4.2.6 Descripteurs

En interne, l'outil utilise une notion de descripteur pour traiter des pointeurs next. L'utilisateur peut demander, avant que la transformation ne s'effectue, plusieurs descripteurs supplémentaires.

4.2.7 Ajout de commentaires

Dans l'entrée, les pointeurs et les sélecteurs de l'automate sont codés par des entiers. La transformation génère deux lignes de commentaires qui indiquent à quel entier correspond quel pointeur et quel sélecteur.

4.3 Annexe A – Exemple d'entrée de l'outils ARTMC

```
# pointer variables are : tmp1=3, pp=2, p=1
    # next pointers are : s=0
    def get_program():
    program=[
    ("x=null","00001",1,1,"NOABSTR"),
    ("x=y.next","00010",3,1,0,2,"NOABSTR"),
    ("ifx==null","00011",3,5,3,"NOABSTR"),
    ("if*","00100",7,4,"NOABSTR"),
    ("exit","00101","NOABSTR"),
    ("x=y","00110",2,1,6,"NOABSTR"),
    ("exit","00111","NOABSTR"),
    ("x=y","01000",1,2,8,"NOABSTR"),
    ("exit","01001","NOABSTR")]
    node_width=7
    pointer_num=4
    desc_num=1
    next_num=1
    err_line="11111"
    restrict_var=1
    env=(node_width,pointer_num,desc_num,next_num,err_line,restrict_var)
    return(program,env)
```

Cette entrée a été générée à partir de l'automate suivant.