Travaux Pratiques : N° 1 - Février 2012
Modélisation et vérification de processus communicants / LOTOS-CADP

1 Présentation de CADP

CADP : *Construction and Analysis of Distributed Processes* voir `www.inrialpes.fr/vasy/cadp/` est une boîte à outils pour la conception et l'analyse de protocoles de communications et plus généralement de systèmes communicants distribués.
CADP permet la compilation, la vérification et la validation de programmes LOTOS. Elle contient de nombreux outils sous forme de modules (indépendants) de compilation ou de vérification.
La vérification dans CADP est essentiellement basée sur le *model checking*. C'est une technique qui consiste à explorer entièrement un modèle fini (un système de transitions étiquettées) en le confrontant avec des propriétés données.

2 Principe général d'utilisation de CADP

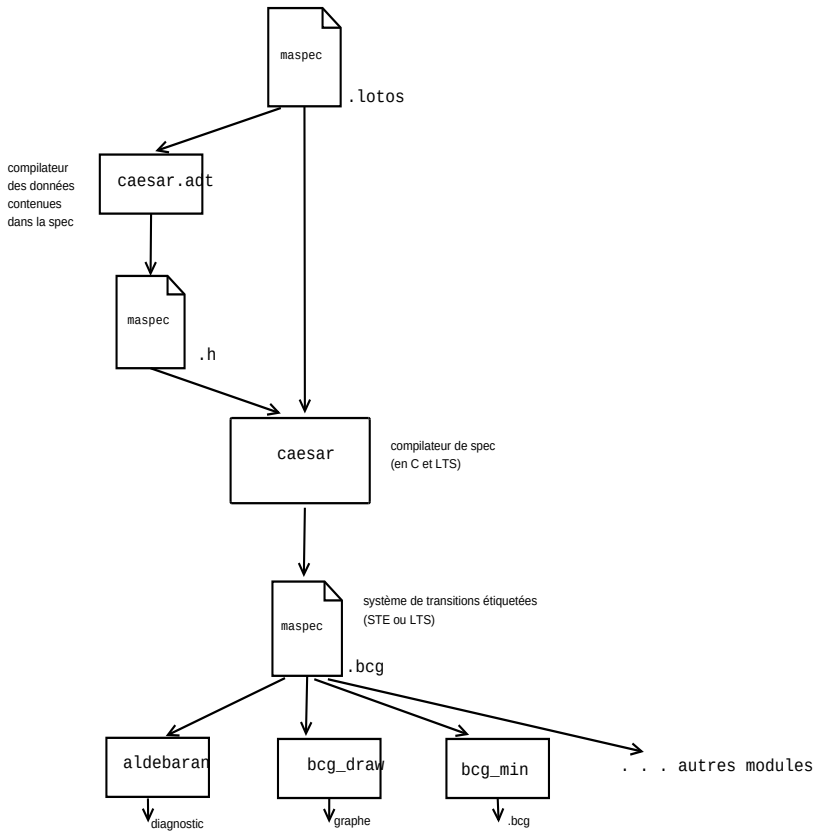

FIGURE 1 – Principe d'usage de CADP

Intégration de LOTOSNT
LOTOSNT est conçu et développé comme une couche au dessus de LOTOS. Par conséquent il y a un préprocesseur qui traduit les spécifications LOTOSNT en LOTOS et la chaîne de vérification reste inchangée.

Une spécification en LOTOSNT doit être contenue dans un fichier avec l'extension `.lnt`

Il y a un préprocesseur lnt2lotos (détails à `http://cadp.inria.fr/man/lnt2lotos.html`) qui traduit les spécifications LNT en LOTOS.

La couche LNT est gérée simplement par le module lnt.open ; les informations détaillées se trouvent en ligne `http://cadp.inria.fr/man/lnt.open.html`.

3 Aperçu de quelques modules de CADP

3.1 Modules de base

- **caesar.adt** : compile la partie donnée, lorsqu'elle existe dans la specification (library ... endlib), en un fichier .h qui est utilisé ensuite par le compilateur **caesar**
- **caesar** : compile toute la spécification LOTOS dont on donne le nom en paramètre.
  Ce module (et commande) admet plusieurs options (telles que `-aldebaran, -bcg, -open`, etc)

3.2 Modules spécifiques

- **ALDEBARAN** : principal module de vérification ; il compare deux graphes, celui du modèle et celui de la propriété ou d'un programme, selon des relations indiquées (bissimulation forte, équivalence observationnelle, etc) ;
- **EXECUTOR** : permet une exécution aléatoire ;
- **GENERATOR, REDUCTOR** : permettent de construire le graphe des états accessibles ;
- **TERMINATOR** : recherche les états de blocage ;
- **BCG_MIN** : minimise le graphe par les techniques de bissimulation ;
- ...

3.3 Modules utilitaires

Tous ces modules prenent en entrée le fichier .bcg
- **BCG_DRAW** : produit une représentation graphique (2D) en postscript du graphe construit après la compilation d'une spécification ;
- **BCG_EDIT** : un éditeur interactif du graphe issu de la compilation d'une spécification ;
- **BCG_INFO** : produit des informations sur le graphe ;
- **BCG_IO** : transforme le .bcg en d'autres formats spécifiés comme options (...)
- **BCG_LABELS** : permet de modifier les étiquettes des transitions du graphe
- **BCG_LIB** : générateur de librairies dynamiques pour bcg
- **BCG_OPEN** : permet d'établir une passerelle entre bcg et l'environnement open/caesar.

4 Vérification de propriétés

On vérifie les propriétés d'un système (ou on analyse formellement un système)
- soit en utilisant ALDEBARAN,
- soit en utilisant le langage et le module XTL (*eXecutable Temporal Language*).

Absence de *deadlock* : L'absence de blocage (*deadlock*) est simplement vérifiée en utilisant le module/commande ALDEBARAN de la façon suivante :

```
aldebaran -dead maspec.bcg
```

ou bien en utilisant le module XTL de la façon suivante :

```
xtl mapropriete.xtl maspec.bcg
```

où `mapropriete.xtl` est un fichier contenant l'expression de la propriété. On utilise les options `-french, -verbose, ...`

En effet pour des propriétés spécifiques, ALDEBARAN peut paraître limité, on exprime alors les propriétés en logique temporelle, puis on les fournit comme paramètre à la procédure de vérification.

4.1 Expression de propriétés en XTL

Absence de deadlock

[true] <true> true

qui exprime que tout état à aumoins un successeur !.

On ne peut faire l'action bb, sans avoir fait l'action aa :

[(not "aa.*") . "bb.*"] false

On arrivera à effectuer l'action "mmm" :

<true*. "mmm"> true

5 Introduction à Evaluator

Parmi les outils de vérification de CADP, il y a le module EVALUATOR, que nous introduisons ici. Nous allons donc réaliser l'étude formelle de propriétés en utilisant le module EVALUATOR.

Le principe de travail est quasiment le même que pour l'utilisation du module XTL, mais l'expression des propriétés est plus simple avec EVALUATOR.

Modules spécifiques de CADP

EVALUATOR : permet d'évaluer à la volée, une formule logique (exprimant une propriété) sur le graphe du processus à étudier. Les formules données en paramètre à ce module sont exprimées dans une logique temporelle appelée μ-calcul régulier. Cette logique est constituée des opérateurs booléens (`true, false, not, and, ...`), des opérateurs modaux (necessaire `[]`, possible `<>`, inevitable, ...) contenant des expressions régulières sur les séquences des actions utilisées dans l'alphabet des processus.

Un manuel complet est en ligne `http://cadp.inria.fr/man/evaluator.html`

6 Vérification de propriétés avec EVALUATOR

6.1 Commande de vérification de propriétés

On exprime les propriétés désirées et on les enregistre dans un fichier avec l'extension .mc (μ-calcul).

La spécification du système à tester est elle, au préalable compilée en un .bcg

On vérifie les propriétés du système en faisant système)

- `bcg_open [bcg_opt] maspec[.bcg] [cc_opt] evaluator [evaluator_opt] prop[.mcl]`
  ou
- `caesar.open [caesar_opt] maspec[.lotos] [cc_opt] evaluator [evaluator_opt] prop[.mcl]`
- ...

Le résultat de cette vérification est TRUE ou FALSE et est affiché à l'écran éventuellement avec des commentaires selon les options utilisées pour la commande.

Vous pouvez voir les autres possibilités et les options avec un `man` comme d'habitude, et aussi la page de documentation de EVALUATOR sur le site web de CADP.

6.2 Expression de propriétés en .mcl pour EVALUATOR

Généralement on effectue l'analyse formelle des systèmes en considérant
– des propriétés de sûreté (*safety*) : "rien de mal n'arrivera" et
– des propriétés de vivacité (*liveness*) : "quelque chose de bien arrivera".

Sûreté : le schéma de telles propriétés est

`[R]false`

où R est une expression régulière sur les actions des processus.
Cette propriété exprime que toutes les séquences d'actions satisfaisant R, mènent nécessairement à des états satisfaisants la propriété `false`. Comme de tels états n'existent pas, les séquences correspondantes à la propriété ne devraient pas exister.

Par exemple

Propriété	expression
la suite d'actions `aa bb` *ne doit pas être possible* dans un système	["aa" . "bb"] false
On ne peut faire l'action bb, sans avoir fait l'action aa	[(not "aa"*) . "bb"*] false

Vivacité : le schéma de telles propriétés est

`[R]inev(A,B,P)`

où R est une expression régulière sur les actions des processus, A et B des prédicats sur les actions, P une propriété sur les états.

Cette propriété exprime que toute séquence d'actions issue de l'état courant et satisfaisant l'expression R, mène nécessairement à des états à partir desquels toutes les séquences contenant 0 ou plusieurs actions satisfaisant A, suivies d'une action satisfaisant B, et menant à un état satisfaisant P.
Par exemple dans un système de contrôle d'accès à une pièce, si on veut s'assurer que lorsqu'un usager a donné un code correct (lco), il finit par entrer dans la pièce (danspiece)...avant de sortir (bouton) on écrira :

[true* . "lco"]inev(not"bouton" , "danspiece", true)

Absence de deadlock

[true*] <true> true

qui exprime que tout état à aumoins un successeur ! après toute séquence d'action, on peut faire n'importe quelle action, et arriver à un état satisfaisant `true`.

Eventualité :
On arrivera à effectuer l'action "mmm" :

<true* . "mmm"> true