Présentation de la boîte à outils CADP: Application au protocole POTS

Benoît Fraikin et Jacob Tchoumtchoua

Groupe de Recherche en Génie Logiciel
Département de Mathématiques et d'Informatique
Université de Sherbrooke
Sherbrooke (Quebec) Canada J1K 2R1

23 avril 2001

1 Vérification et Validation

Spécifier, puis Valider, enfin Implémenter : s'il est possible, à la rigueur, de se dispenser de cette démarche pour de petits programmes, elle s'avère indispensable pour les systèmes temps réel comme par exemple dans le domaine des protocoles de télécommunications, à cause de la complexité de ses applications et les exigences de fiabilités auxquelles elles sont soumises.

Model Checking : Le model checking est une technique qui consiste à engendrer un modèle fini (ici un Système de Transitions Étiquetées) à partir du programme, puis à comparer ce modèle aux spécifications grâce à une procédure de décision.

LOTOS est un langage permettant de spécifier l'architecture et le fonctionnement de systèmes distribués. Il permet de définir et de manipuler les structures de données. Mais à la différence des langages classiques, LOTOS utilise le formalisme des types abstraits algébriques. Les spécifications algébriques constituent un modèle mathématique exprimant les propriétés que doit vérifier toute réalisation, sans imposer de contraintes d'implémentation superflues.

La vérification : elle consiste à comparer un système que l'on suppose donné par un programme exprimé dans un langage possédant une sémantique opérationnelle bien définie (la description des services qu'il doit rendre aux utilisateurs). Les outils de la boîte CADP utilisent deux catégories de spécifications :

1. Spécifications comportementales : elles décrivent le comportement du système exprimé sous forme d'états et une relation de transition entre

ces états. Elle peut être formalisée à l'aide des algèbres de processus ou des automates,

2. Spécifications logiques : elles caractérisent des propriétés globales du système, telles que l'absence de blocage, l'exclusion mutuelle. Les logiques temporelles sont utilisées pour leur formalisme car le système est décrit dans le temps. Dans ce cas, les programmes sont liés aux formules logiques par une relation de satisfaction.

2 Présentation de la boîte à outils

CADP est une boîte à outils pour la compilation, la vérification et la validation de programmes LOTOS. Elle a été développée par l'équipe VASY[1] de l'INRIA Rhônes–Alpes. Elle intègre un ensemble cohérent d'outils permettant de traiter des spécifications comportementales et logiques, en utilisant les méthodes basées sur les modèles. L'environnement comprend un ensemble de bibliothèques avec leurs interfaces de programmation, ainsi que divers outils parmi lesquels :

- EVALUATOR, qui évalue à la volée des formules de μ-calcul régulier sans alternance,
- EXECUTOR, qui permet l'exécution aléatoire,
- EXHIBITOR, qui recherche des séquences d'exécution caractérisées par une expression régulière,
- GENERATOR et REDUCTOR, qui construisent le graphe des états accessibles,
- SIMULATOR et XSIMULATOR, qui permettent la simulation interactive, et
- TERMINATOR, qui recherche les états de blocage.

Il existe deux catégories de composants dans la boîte à outils CADP :

1. les compilateurs : utilisés pour traduire le programme à vérifier vers un modèle (graphe d'états, Système de transitions étiquetées, automate d'états). La boîte à outils contient deux compilateurs :
   - CAESAR.ADT pour le traitement des données,
   - CAESAR pour le contrôle des programmes LOTOS.
2. les vérificateurs : ils effectuent des vérifications sur les graphes engendrés par les compilateurs. La boîte à outils comprend deux vérificateurs :
   - ALDÉBARAN : c'est un vérificateur de spécifications comportementales, qui compare deux graphes modulo diverses relations.

[1] http://www.inrialpes.fr/vasy/cadp

- XTL : (eXecutable Temporal Language) est un méta-langage adapté à l'expression des algorithmes d'évaluation et de diagnostic pour les formules de logiques temporelles telles que CTL, HML, ACTL, LTAC.

En sortie ces outils sont capables, lorsque le programme à vérifier est incorrect, de fournir à l'utilisateur des diagnostics en termes de séquences dans le graphe.

Le cas pratique à vérifier - appelé protocole - est décrit en LOTOS, puis traduit vers un graphe. Le service attendu - appelé service - est spécifié, soit par un programme LOTOS lui-même traduit en graphe, soit par des formules logiques LTAC.

2.1 Les graphes au format BCG

BCG (Binary Coded Graphs) est un format qui utilise des techniques efficaces de compression permettant de stocker des graphes (représentés sous forme explicite) sur disque de manière très compacte. Ce format joue un rôle central dans la boîte à outils CADP. Il est indépendant du langage source et des outils de vérification. En outre, il contient suffisamment d'informations pour que les outils qui l'exploitent puissent fournir à l'utilisateur des diagnostics précis dans les termes du programme source. Pour exploiter le format BCG, est développé un environnement logiciel qui se compose de bibliothèques avec leurs interfaces de programmation et de plusieurs outils, notamment :

- BCG_DRAW, qui permet d'afficher en PostScript une représentation 2D d'un graphe,
- BCG_EDIT, qui permet de modifier interactivement la représentation graphique produite par BCG_DRAW,
- BCG_IO, qui effectue des conversions entre BCG et d'autres formats d'automates,
- BCG_LABELS, qui permet de masquer et/ou de renommer par des expressions régulières les étiquettes d'un graphe BCG,
- BCG_MIN, qui permet de minimiser des graphes BCG selon la bisimulation forte ou la bisimulation de branchement (éventuellement étendue au cas des systèmes probabilistes ou stochastiques), et
- BCG_OPEN, qui permet d'appliquer à tout graphe BCG les outils disponibles dans l'environnement.

2.2 CAESAR.ADT

2.2.1 Présentation de l'outil

CAESAR.ADT est un compilateur qui traduit les définitions de types abstraits figurant dans un programme à vérifier en un programme C. Il permet donc, à partir d'une spécification formelle, d'obtenir automatiquement une implémentation prototype correspondante. Nous supposerons maintenant que le programme à vérifier est écrit en LOTOS.

2.2.2 Principes

Généralité :

CAESAR.ADT prend en entrée une spécification formelle LOTOS dont il ne traite que les définitions de types abstraits (les définitions de processus étant ignorées). Il produit en sortie la une bibliothèque en langage C contenant, pour chaque sorte (resp. opération) définie dans le programme à verifier un type (resp. une fonction) C qui l'implémente. Cette bibliothèque permet de manier les types de données algébriques contenus dans la spécification. Ainsi CAESAR.ADT permet aussi par extension l'utilisation des fonctions appliquées aux variables contenues dans cette dernière. Cependant dans le cas de LOTOS on ne peut accéder aux variables contenues dans la spécification (le programme à vérifier) puisque celles-ci sont attachées à une porte et non à un état comme le nécessiterait l'utilisation d'un STE (en l'occurence le format BCG).

Restrictions imposées par CAESAR.ADT :

Le programme LOTOS doit comporter certains commentaires spéciaux qui permettent d'établir une correspondance entre le nom des objets LOTOSet le nom des objets C qui les implémentent.

CAESAR.ADT impose certaines restrictions sur le sous–ensemble de LOTOStraité. L'utilisateur doit respecter la discipline de programmation par constructeurs, c'est–à–dire :

- Diviser l'ensemble des opérations LOTOSen constructeurs (opérations primitives) et non–constructeurs (opérations dérivées, exprimées en fonction des constructeurs). Les opérateurs constructeurs doivent être indiqués à l'aide de commentaires spéciaux.

- Orienter les équations pour qu'elles se comportent comme des règles de réécriture, le membre droit spécifiant comment le membre gauche doit être réecrit. La stratégie de réécriture choisie est l'appel par valeur, complété par une règle de priorité décroissante entre équations (si plusieurs équations peuvent simultanément s'appliquer, alors la première d'entre elles a précédence sur les suivantes).

- Veiller à ce que toute partie gauche d'équation ait la forme $f(v_1,\ldots,v_n)$, avec $n \geq 0$, où f est un non–constructeur et $v1,\ldots,v_n$ des sous–termes formés uniquement de constructeurs et de variables quantifiées universellement (les non–constructeurs étant proscrits).

2.3 CAESAR

CAESAR est un outil de compilation et de vérification pour les programmes LOTOS. CAESAR a pour objectif d'appliquer à LOTOS les méthodes basées sur les modèles, en traduisant le programme source à vérifier en un graphe qui modélise toutes les évolutions possibles (états et séquences d'exécution) du programme.

2.3.1 Fonctionnalités

En entrée, CAESAR prend le programme LOTOS à vérifier, ainsi qu'une implémentation en C pour les types abstraits qu'il contient (soit écrite à la main, soit engendrée automatiquement par CAESAR.ADT). En sortie CAESAR produit un système de transitions étiquetées (STE). Les informations contenues dans ce STE peuvent être exploitées par divers outils (réducteurs d'automates, évaluateurs de logiques temporelles ou de calculs, outils de diagnostics).

- ALDÉBARAN
- EVALUATOR
- TERMINATOR
- EXECUTOR

CAESAR est capable d'engendrer ce graphe sous des formats multiples afin d'interfacer de nombreux outils existants ; en premier lieu ALDÉBARAN. Le tableau 1 donne un rapide récapitulatif des extensions utilisés par CAESAR.

Extension	Type de fichier
NomFichier`.lotos`	spécification LOTOS (en entrée)
NomFichier`.aut`	automate pour ALDÉBARAN (en sortie)
NomFichier`.bcg`	STE au format BCG (en entrée ou en sortie)
NomFichier`.exp`	réseau de communication de STEs (en entrée)
NomFichier`.rename`	liste des étiquettes à renommer (en entrée)
NomFichier`.hide`	liste des étiquettes à cacher (en entrée)
NomFichier`.net`	réseau de Pétri (en sortie)
NomFichier`.gph`	automate pour le traçage d'erreur (en sortie)
NomFichier`.err`	détails des messages d'erreurs (en sortie)

Table 1: récapitulatif des extensions des fichiers gérés par CADP

2.3.2 Principes de fonctionnement

La traduction du programme LOTOS en graphe se fait en quatre étapes successives :

1. La première phase traduit le programme LOTOS en un programme SubLOTOS équivalent (SubLOTOS étant un sous-ensemble simplifié de LOTOS).

2. La phase de génération traduit le programme SubLOTOS en une forme intermédiaire, appelée réseau, qui comprend :

   - une partie contrôle, basée sur les réseaux de Pétri, qui consiste en un ensemble d'états et de transitions sur lesquels une structure en unités permet de conserver la décomposition en processus communicants, qui existe dans le programme source LOTOS,
   - une partie donnée, formée d'un ensemble de variables globales typées, dont les valeurs peuvent être consultées ou modifiées par des actions attachées aux transitions du réseau.

3. la phase d'optimisation a pour objectif de réduire la complexité du réseau (c'est-à-dire de diminuer le nombre de places, de transitions, d'unités et de variables) en appliquant une liste de transformations qui préservent la sémantique pour la relation de bisimulation forte. Ces optimisations portent aussi bien sur la partie contrôle du réseau que sur la partie données.

4. La phase de simulation construit le graphe correspondant au réseau ainsi optimisé. Elle effectue une exploration exhaustive du graphe ; tous les états rencontrés sont conservés en mémoire principale, tandis que les arcs du graphe sont écrits au fur et à mesure dans un fichier.

2.4 ALDEBARAN

ALDÉBARAN est un outil permettant la réduction et la comparaison de graphes par rapport à différentes relations d'équivalence et de pré–ordre. Parmi les relations utilisées pour la vérification des systèmes parallèles nous avons en premier :

- la bisimulation forte
- l'équivalence observationnelle
- l'équivalence par modèle d'acceptation, ainsi que le pré-ordre
- l'équivalence de sûreté qui préserve les propriétés de sûreté des systèmes.

2.4.1 Fonctionnalités

ALDÉBARAN offre deux fonctionnalités correspondant à deux besoins pratiques distincts :

- On peut comparer le graphe d'un programme avec celui de ses spécifications comportementales modulo une relation d'équivalence ou de pré–ordre. Dans ce cas, les deux graphes et la relation doivent être fournis en entrée, et le résultat de la comparaison (vrai ou faux) est obtenu en sortie. En outre, lorsque les deux graphes ne sont pas équivalents, ALDÉBARAN fournit un diagnostic constitué d'un ensemble de séquences d'exécution menant, à

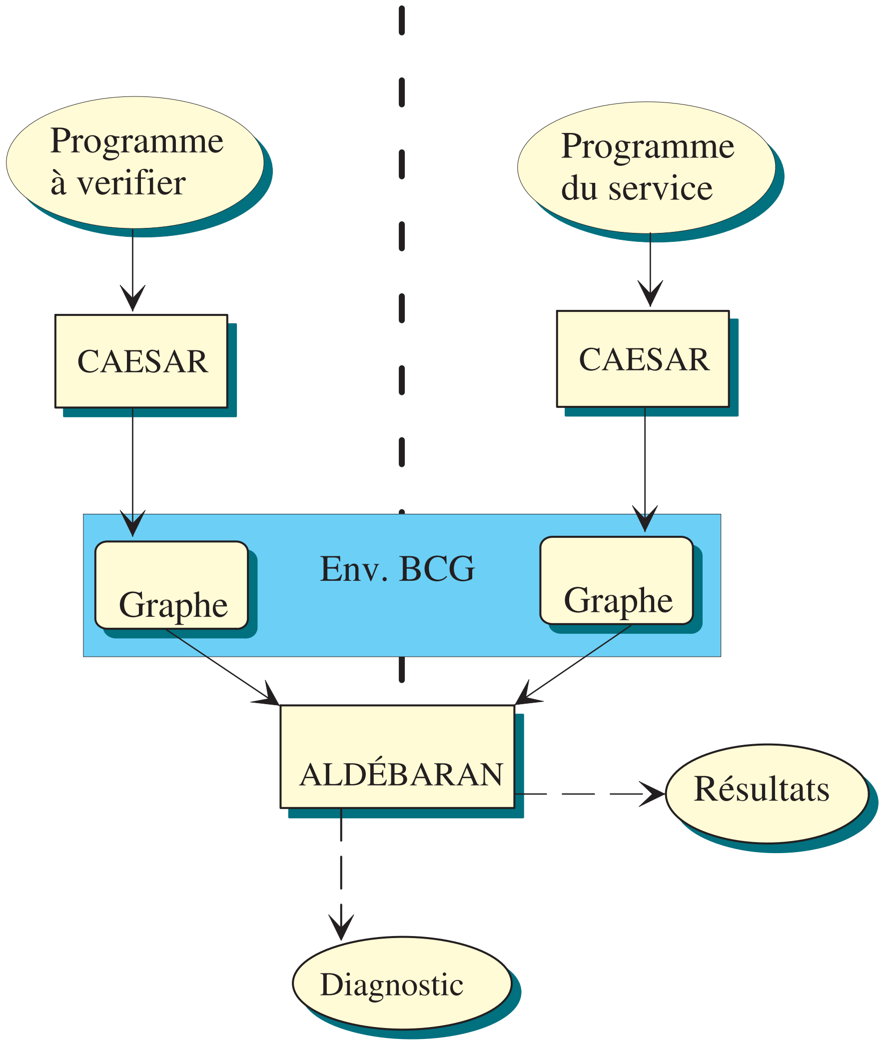


Figure 1: Schéma du fonctionnement général avec Aldébaran

partir des états initiaux des deux graphes et en suivant les mêmes actions, à deux états où la non-équivalence apparaît clairement (il existe une action ne pouvant être effectuée que dans un seul des deux états).

- On peut réduire un graphe, c'est-à-dire calculer son quotient par rapport à une relation d'équivalence donnée (c'est-à-dire le plus petit graphe équivalent pour cette relation au graphe original). Dans ce cas, le graphe et la relation d'équivalence sont fournis en entrée, et le graphe quotient est obtenu en sortie.

2.4.2 Principes de fonctionnement

Le fonctionnement général d'ALDÉBARAN est donné par la syntaxe suivante :

```
<method> -<relation><action>
```

où `<method>` est une chaîne de trois caractères, désignant l'algorithme utilisé pour la réduction du graphe à savoir `std`, `fly` ou `bdd` :

- `std` : utilise l'algorithme de Paige & Tarjan (excepté les branches équivalentes) ou l'algorithme de Groote & Vaandrager (pour les branches équivalentes uniquement)
- `fly` : utilise l'algorithme Fernandez & Mounier
- `bdd` : utilise l'algorithme de la Génération du Model Minimal

où `<relation>` est une chaîne d'un caractère, à savoir `b`, `d`, `i`, `o`, `p` ou `s` :

- `b` : utilise l'équivalence par bisimulation forte ou la relation de préordre correspondant
- `d` : utilise la bisimulation temporelle
- `i` : utilise la bisimulation tau*.a ou la relation de préordre correspondant
- `o` : utilise la relation d'équivalence observationnelle
- `p` : utilise la bisimulation par branche
- `s` : utilise l'équivalence de sûreté ou la relation de préordre correspondant

et où `<action>` est une chaîne de caractères, ayant une des valeurs suivantes `min`, `cla`, `equ` ou `ord` :

- `min` : minimise le système de transitions étiquetées contenu dans *Nomfichier*`.aut`, ou le réseau de Petri pour la communication du système de transition représenté par *Nomfichier*`.exp`, d'après la `<relation>`.
- `cla` : Idem, à la différence qu'au lieu de réduire le système de transitions étiquetées, sont affichées les classes équivalentes sur la sortie standard.

- `equ` : compare les deux systèmes de transitions étiquetées contenus respectivement dans *Nomfichier1*`.aut` et *Nomfichier2*`.aut` ou le réseau de Petri pour la communication du système de transition *Nomfichier*`.exp` avec le systèmes de transitions étiquetées *Nomfichier2*`.aut`, ou le réseau de Petri pour la communication du Système de transition *Nomficher2*`.exp` avec le système de transition étiquetées *Nomfichier1*`.aut`, selon `<relation>`, l'algorithme choisi, et l'affichage des résultats en format de séquence. Ce résultat peut être vrai ou faux; en dernier ressort, ALDÉBARAN est capable lorsque le programme à vérifier est incorrect, de fournir à l'utilisateur des diagnostics en termes de séquences du graphe.

- `ord` : même chose que `equ`, mais utilise une relation de préordre au lieu de la relation d'équivalence.

Deux approches principales existent pour décider si deux graphes se bisimulent fortement. Elles peuvent toutes deux être étendues au cas des bisimulations faibles :

- La première approche, basée sur un calcul de point fixe, procède par raffinements successifs d'une partition initiale des états du graphe. Après stabilisation, la partition obtenue fournit les classes d'équivalence du graphe pour la bisimulation forte (c'est-à-dire les états du graphe quotient). Par la suite, comparer deux graphes pour la bisimulation forte revient à comparer leurs quotients. Un algorithme efficace de raffinement de partitions, proposé par Paige et Tarjan, a été implémenté dans ALDÉBARAN.

- La seconde approche permet la comparaison "à la volée" de deux graphes en les parcourant en profondeur simultanément, ce qui s'assimile au calcul d'un produit synchrone d'automates ; cet algorithme ne permet pas d'effectuer des réductions de graphe.

2.4.3 Algorithmes de bisimulation

'STD' algorithme de Paige & Tarjan

- Basé sur un calcul de point fixe
- Procède par raffinement successifs d'une partition initiale des états du graphes

Avantage :

- Réduction du graphe qui prend peu de place sur le disque,
- Intéressant lorsque le graphe servira pour plusieurs vérifications.

Limites :

- Coût très élevé en mémoire,
- Diagnostics dans les graphes quotients et non dans les graphes originaux, ce qui rend plus difficile la recherche d'erreurs.

'FLY' Fernandez et Mounier

- Comparaison “à la volée” de deux graphes,
- Parcours en profondeur simultanément.

Avantage :

- Traite les bisimulations forte et faible,
- Traite les cas de grande taille en peu de temps,
- En cas d'erreurs le graphe n'est pas forçement entièrement parcouru,
- Diagnostics sur les graphes originaux.

Limites :

- Le graphe n'est pas optimisé ce qui rend coûteux sa réutilisation pour plusieurs bisimulations.

2.4.4 Conclusion

ALDÉBARAN est un outil capable en quelques minutes de traiter des graphes de plusieurs milliers d'états modulo la bisimulation forte ou l'équivalence observationnelle, ou de comparer des graphes d'une centaine de milliers d'états modulo l'équivalence de sûreté. Le plus souvent, ALDÉBARAN est utilisé avec CAESAR de manière interactive, par exemple pour comparer le graphe d'un protocole au graphe du service attendu. ALDÉBARAN peut aussi être utilisé de façon “interne” par d'autres logiciels appelés à minimiser des graphes.

Inconvénient :

- Le problème de l'explosion du nombre d'états limite l'utilisation actuelle d'ALDÉBARAN à la comparaison de graphes de taille moyenne.
- La principale limitation de cette méthode est son coût en mémoire elle nécessite en effet de mémoriser les états et les transitions des deux graphes. Cependant la possibilité de procéder quelques fois par la méthode de génération “à la volée” du modèle permet de limiter ce problème lors de la phase de recherche d'erreurs.

Avantage : Il présente l'avantage de fournir des diagnostics directement exprimés en terme de séquences d'exécution sur les graphes originaux.

2.5 XTL

Xtl - pour *eXecutable Temporal Language* - est un langage de programmation fonctionnelle qui permet d'implémenter des formules de logiques temporelles vérifiables sur un modèle de STE composé par un graphe BCG dans notre cas.

2.5.1 Principes

Xtl permet donc une description compacte de nombreuses formules de logique temporelle et permet aussi leur évaluation sur un STE. Les opérateurs de la logique temporelle sont définis à l'aide d'expressions itératives calculant des ensembles d'états ou d'étiquettes de transitions.
Plusieurs logiques temporelles très utilisées ont été implémentées en Xtl telles que Actl, Hml ou encore le μ-calcul. Il a été utilisé pour des études de cas pour le protocol Brp de Philips et le protocol Cfs de Bull et l'Inria entre autres.

2.5.2 Description du langage

Les types de données

On peut distinguer deux types de données importantes dans Xtl :

- les types de données externes que composent les types de données abstraits provenant du programme à vérifier et
- les types de données internes que composent les types prédéfinis - comme `integer` et `boolean` - et les *meta–types* `stateset`, `state`, `edgeset`, `edge`, `labelset` et `label` qui correspondent aux (ensembles de) états, transitions et étiquettes du STE. De plus ces types sont munis de *meta–opérateurs*, donnés dans la table 2 permettant d'accéder à l'état initial et d'explorer la relation de transition.

Opérateur	Sémantique
`init : -> state`	état initial
`succ : state -> stateset`	successeurs d'un état
`pred : state -> stateset`	prédécesseurs d'un état
`in : state -> edgeset`	transitions menant à un état
`out : state -> edgeset`	transitions provenant d'un état
`source : edge -> state`	états d'origine d'une transition
`target : edge -> state`	états de destination d'une transition

Table 2: Meta–opérateurs basiques de Xtl

Les opérateurs du langage

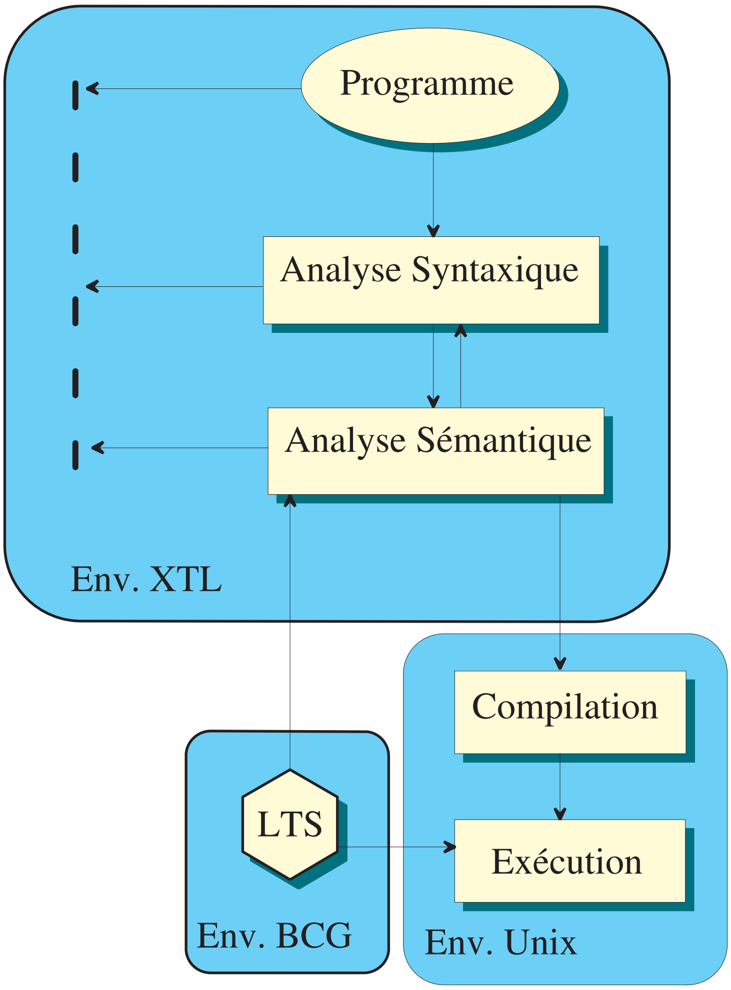


Figure 2: Schéma du fonctionnement général avec XTL

Le moyen le plus simple d'implémenter des opérateurs temporels en XTL exprimant des propriétés sur des étiquettes de transitions ou des états est de calculer la sémantique dénotationnelle de ces propriétés, *i.e.* l'ensemble des étiquettes ou des états les vérifiant. Pour cela on dispose dans XTL de deux outils principaux que sont la correspondance d'étiquettes (*label matching*) et le calcul d'un ensemble par compréhension. La table 3 indique la syntaxe de ces expressions.

SYNTAXE	SÉMANTIQUE
E `->` `[` G `?`$x : T$ `!`E_1 [`where` E_2] `]`	correspondance d'étiquettes
`{` $x : T$ `where` E `}`	calcul par compréhension

Table 3: Opérateurs de calcul d'ensembles

XTL permettant l'évaluation de formules, on retrouve bien évidemment les deux quantificateurs de la logique classique (table 4).

SYNTAXE	SÉMANTIQUE
`exists` $x : T$ `in` E `end_exists`	quantificateur existentiel
`forall` $x : T$ `in` E `end_forall`	quantificateur universel

Table 4: Les quantificateurs

Enfin XTL reste un langage de programmation et offre donc un grand éventail d'opérateurs. On retrouve ainsi les appels de fonctions, définition de variables, de branchement conditionnel et de branchement itératif. La table 5 donne un aperçu de la syntaxe des plus utilisés.

SYNTAXE	SÉMANTIQUE

`let` $x : T$`:=`E `in`
    E_1
`end_let`
	définition de variable

`if` E `then` E_1 `else` E_2 `end_if`	branchement conditionnel

`for` [$x_0 : T_0$] [`in` $x_1 : T_1$] [`while` E_1]
  `apply` F
  `from` E_2
  `to` E_3
`end_for`
	branchement itératif

Table 5: Résumé des expressions classiques de XTL

Définitions et macros

Enfin, il est possible de créer des macros et des définitions de fonctions réutilisables. XTL offre de nombreuses librairies d'implémentation des logiques temporelles les plus classiques ainsi que de macros utiles. Cependant nous n'en parlerons pas dans ce rapport car cela sort du cadre du cours.

2.5.3 Exemples de modalités

La modalité $\langle\alpha\rangle\phi$ de la logique HML

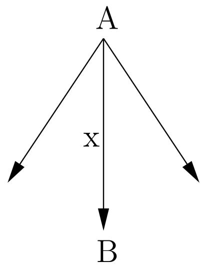
- cette modalité calcule l'ensemble des états qui vérifient la propriété : "on peut exécuter une transition dont l'étiquette vérifie α et arrive dans un état qui vérifie la propriété ϕ". La figure 3 illustre un état vérifiant cela.
-
```
def DIA(A:labelset,F:stateset) : stateset =
  { s:stateset where
    exists t:edge among out(s) in
       label(t) in (A) and target(t) in (F)
    end_exists
  }
end_def
```

La modalité $\texttt{EF}_\alpha(\phi)$ de la logique ACTL

- cette modalité calcule l'ensemble des états qui vérifient la propriété : "il existe un chemin dont toutes les étiquettes de ces transitions vérifient α et dont l'arrivée est un état qui vérifie la propriété ϕ". La figure 4 illustre un état vérifiant cela.
-
```
def EF_A (A:labelset,F:stateset) : stateset =
    lfp(X, G or DIA ( A or TAU, X ))  end_def
```
  où `lfp(X,H)` est le calcul du plus petit point fixe `X` vérifiant `H` et où `TAU` est l'ensemble composé de l'étiquette τ.

2.5.4 Conclusion

XTL offre un moyen facile d'implémenter de nombreuses formules de logiques temporelles ainsi qu'un évaluateur efficace de telle formules. Les librairies de modalités implémentées permettent à l'utilisateur habitué à une logique existante de vite s'y retrouver, et aux nouveaux de pouvoir disposer des modalités les plus classiques déjà définies qui leur permettront d'implémenter leurs formules.
Cependant lorsque les cadres des formules ne sont pas directement prévus par ces librairies, il peut être difficile d'implémenter directement les formules.



$x \models \alpha \,\wedge\, B \models \phi$

Figure 3: Modalité $\langle\alpha\rangle\phi$

$x \models \alpha \,\wedge\, y \models \alpha \wedge$
$z \models \alpha \,\wedge\, B \models \phi$

Figure 4: Modalité $\texttt{EF}_\alpha(\phi)$

2.6 XSIMULATOR

Enfin il est important de noter XSIMULATOR un simulateur interactif qui nous a permis durant la partie de réécriture de la spécification de trouver une erreur dans les types algébriques définis.
Cette outil est très efficace pour trouver les erreurs lorsque la propriété à vérifier est fausse alors qu'elle aurait dû être vraie. Ceci permet de s'affranchir du fait par exemple qu'avec XTL on ne peut récupérer les variables de la spécification LOTOS ce qui sert normalement pour la découverte d'erreurs.

3 Étude de cas sur la spécification du POTS

Notre étude portera sur la spécification du POTS vu en cours ift734. La spécification écrite en LOTOS a été réécrite par rapport à la spécification originelle pour correspondre aux restrictions imposées par CAESAR.ADT (*c.f.* la section 2.2.2).

3.1 La spécification LOTOS

La spécification du POTS est une spécification d'un standard téléphonique. Les propriétés que nous allons tenter d'établir sont de deux sortes :

- propriétés comportementales, qui portent sur le comportement du système pour un utilisateur ; ce seront des propriétés telles que la succession des états du combiné (décroché ou raccroché), la succession des évènements auxquels peut s'attendre un utilisateur avec son téléphone,

- propriétés logiques, qui portent sur le fonctionnement du système et qui nous permettront de vérifier que certaines séquences peuvent apparaître ou que le système obéit à certaines contraintes ; ce seront par exemple le fait qu'un utilisateur ne peut joindre un autre que s'il est le premier à l'appeler.

Les types algébriques ont été définis complètement pour la spécification. Cependant ne pouvant pas accéder aux variables de celle–ci, nous n'en verrons pas d'utilisation directe. Ces définitions ne serviront qu'à illustrer les contraintes de CAESAR.ADT (*c.f.* section 2.2.2).

La version de la spécification est donnée en annexe.

3.2 L'étude avec ALDÉBARAN

La vérification par bisimulation, qu'utilise ALDÉBARAN, peut porter sur plusieurs types de comportements. Pour notre étude ce seront essentiellement les comportements observationnel et de sécurité auxquels nous nous intéresserons. Il y a une équivalence observationnelle entre deux programmes si les transitions sont équivalentes du point de vue de l'utilisateur, en tenant compte du fait que le système peut modifier son état interne. L'équivalence de sécurité ne tient pas compte des τ–transitions.

3.2.1 Préliminaires

Nous utiliserons ici une spécification sans transitions cachées (c'est–à–dire sans utiliser de `hide` dans la spécification). En effet ALDÉBARAN offre la possibilité d'effectuer des renommages et de cacher des étiquettes lors des bisimulations. Le fait de garder toutes les étiquettes visibles nous laissera plus de libertés a priori sur les propriétés à vérifier.

La première phase va consister à compiler la spécification LOTOS par le compilateur CAESAR.ADT. Cette phase peut être ignorée dans le cas où la spécification LOTOS ne contient pas de type abstrait algébrique.

```
#> caesar.adt specMF.lotos
```

Ceci crée un fichier d'en–tête pour la compilation suivante.

```
#> caesar specMF.lotos
```

Cette phase permet de créer un fichier *specMF*`.bcg` contenant le système de transitions étiquetées.
Nous pouvons procéder à la réduction du graphe précédent par équivalence observationelle :

```
#> aldebaran -omin specMF.bcg
```

Comme nous l'avons indiqué précédemment, nos propriétés ne seront bisimulées que par équivalence observationnelle ou de sécurité. La réduction par équivalence observationnelle du graphe BCG ne faussera donc pas les résultats.

À partir de là, vient le moment de nous interroger sur la méthode à suivre pour pouvoir écrire une propriété et la simuler sur le graphe BCG. Il faudra que nous nous posions pour chaque propriété les questions suivantes :

- *Quelle est la classe d'équivalence la plus restrictive qui va me permettre d'établir cette propriété ?*
- *Quelles sont les étiquettes de transitions mises en jeu ?*
- *Y a–t–il des étiquettes que je vais pouvoir regrouper en une seule catégorie pour faciliter l'écriture de ma propriété ?*
- *Quelles vont être les étiquettes cachées ?*

Ces questions permettent de créer les fichiers *service*`.rename` et *service*`.hide` nécessaires à la simulation et d'écrire le fichier *service*`.aut` ou *service*`.lotos` représentant le service à bisimuler.

3.2.2 Propriété 1

1. Nous voulons établir une absence d'interblocage.
2. Il n'est pas nécessaire de se poser les questions indiqués ci–dessus puisqu'ALDÉBARAN nous offre une commande pour vérifier cette propriété :
   `#> aldebaran -dead specMF.bcg`.
3. Résultat : il n'y a pas d'interblocage dans le protocole.

3.2.3 Propriété 2

1. On veut prouver qu'un utilisateur pourra toujours exécuter l'action décrocher puis raccrocher.
2. On utilisera pour cela l'équivalence observationnelle.
3. On ne s'intéresse qu'aux étiquettes `DEC !1` et `RAC !1`. Toutes les autres étiquettes seront dissimulées et aucune ne sera renommée. Seul l'utilisateur 1 est testé puisque tous les utilisateurs ont un rôle symétrique.
4. La figure 5 montre le graphe du service.
5. Résultat : la propriété est vérifiée pour l'utilisateur 1 et donc pour tout les autres.

3.2.4 Propriété 3

1. On veut maintenant montrer qu'un utilisateur ne peut pas raccrocher avant d'avoir décroché ; il s'agit ici de vérifier que l'opération inverse de la propriété 2 n'est pas possible.
2. On conserve la même équivalence.
3. Les réponses à nos questions sont identiques.
4. La figure 6 montre le graphe du service.

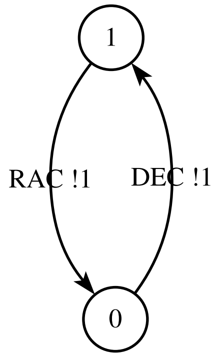


Figure 5: ALDÉBARAN- Propriété 2

5. Résultat : le message d'erreur suivant est affiché
   ```
   sequence 1:
   initial states
   Only service.aut can do a ``RAC !1''-transition from these states
   ```
   Ce qui prouve que le protocole ne peut effectuer `RAC!1` à l'état initial.

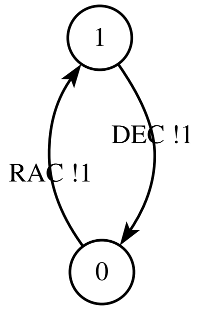


Figure 6: ALDÉBARAN- Propriété 3

3.2.5 Propriété 4

1. Cette propriété devra vérifier qu'un utilisateur peut décrocher, avoir une tonalité, puis raccrocher.

2. On utilise toujours l'équivalence observationnelle.

3. Les étiquettes intéressantes sont bien évidemment `DEC !1`, `RAC !1` et `TON !1`. Toutes les autres seront dissimulées. La remarque de la propriété 2 sur la symétrie du rôle des utilisateurs est toujours valable.

4. La figure 7 montre le graphe du service.

5. Résultat : la propriété est vraie.

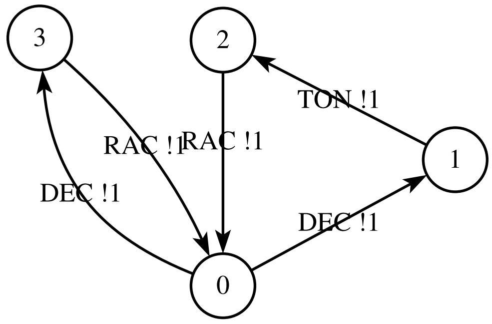


Figure 7: ALDÉBARAN- Propriété 4

3.2.6 Propriété 5

1. On va chercher à établir l'assertion qu'après avoir composé un numéro on ne peut avoir que deux possibilités : début sonnerie ou tonalité occupée.

2. L'équivalence en jeu va toujours être l'équivalence observationnelle.

3. Ici, les étiquettes qui nous intéressent sont toutes les étiquettes `COMP !1 !x` et `DEBSON !1 !x` ainsi que `TONOC !1`. La valeur exacte de `x` a peu d'importance en soi. Ce qui compte c'est que du point de vue de l'utilisateur lorsqu'il appelle quelqu'un, une de ces deux sonneries retentira dans son combiné. Nous allons donc renommer les étiquettes `COMP !1 !x` en une seule `COMPUN` et les étiquettes `DEBSON !1 !x` en une seule `DEBSONUN`. Par la suite seules les trois étiquettes `COMPUN`, `TONOC !1` et `DEBSONUN` ne seront pas dissimulées. Ce travail s'effectue facilement par l'utilisation d'expressions rationnelles (*regular expression*).

4. La figure 8 montre le graphe du service.

5. Résultat : la propriété est vraie.

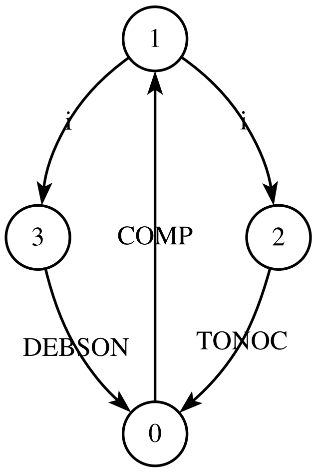


Figure 8: ALDÉBARAN- Propriété 5

3.2.7 Propriété 6

1. Dans cette dernière propriété nous allons représenter le comportement global pour un utilisateur. C'est–à–dire que nous allons modéliser l'ensemble des actions possibles qui peuvent interagir avec lui. On pourra ainsi remarquer une erreur de notre spécification.

2. L'équivalence choisie ici sera l'équivalence de sécurité. Elle correspond mieux à un comportement confronté à un humain. De plus elle suffit pour établir l'erreur suscitée.

3. La figure 9 montre le graphe du service. Comme le montre cette figure, il est possible à un utilisateur de téléphoner à un autre utilisateur, d'obtenir la sonnerie du téléphone et, juste après avoir raccroché, de pouvoir être rejoint par une tierce personne l'appelant avant que la première sonnerie ait fini. Le chemin est le suivant :

   - état initial 0
   - transition `DECUN`, état 2,
   - transition `TONUN`, état 3,
   - transition `COMPUN`, état 5,
   - transition `DEBSONUN`, état 13,
   - transition `RACUN`, état 7 (jusqu'ici l'utilisateur–témoin a appelé quelqu'un et a raccroché avant que l'autre ne réponde. La sonnerie d'appel n'a pas encore terminé),

- transition `DEBSONAPUN`, état 10 (le téléphone sonne car quelqu'un appelle l'utilisateur–témoin),
- transition `DECUN`, état 12,
- transition `FINSONAPUN`, état 9,
- transition `CONNUN`, état 4,
- transition `RACUN`, état 7, (l'utilisateur–témoin a décroché et repondu à l'appel)
- transition `FINSONUN`, état 0, (ce n'est que maintenant que l'appel de la première sonnerie s'arrête ; elle a retenti durant toute la conversation !).

4. Résultat : la propriété est verifiée par la simulation. L'erreur est donc bien présente. Ceci est dû à deux choses :

   (a) la libération de l'appelant s'effectue avant la fin de sonnerie (`rac !1;lib !1;finSon !1 !x`) et

   (b) il y a possibilité d'exécuter la transition `dec !1` dans le processus de chaque utilisateur.

   Nous n'avons pas trouvé de solution à ce problème pour l'instant.

3.2.8 Bilan

La vérification par bisimulation permet d'effectuer des vérifications sur des services simplement. Ainsi la dernière propriété nous a permis de mettre à jour une erreur de la spécification du POTS de manière assez simple en utilisant la réduction de graphe afin d'analyser le service offert du point de vue d'un seul utilisateur. Le défaut principal de procéder par bisimulation est la mémoire utilisée. Cependant les algorithmes de Paige & Tarjan et "à la volée" (*c.f.* section 2.4.3) permettent de traiter avec efficacité la majorité des cas : le premier étant plus adapté pour la vérification de plusieurs propriétés et le second pour la recherche des premières erreurs. Pour des propriétés plus détaillées cependant, le procédé peut sembler trop faible (bien que des études de cas aient montré qu'on peut vérifier parfois de nombreuses propriétés facilement). Le langage XTL peut prendre la relève afin d'essayer d'établir des propriétés logiques du système.

3.3 L'étude avec XTL

Comme il a été expliqué dans la description du langage, de nombreuses formules de logiques temporelles sont implémentables en XTL. Du fait de l'existence de librairies bien fournies il vaut mieux chercher à décrire d'abord les propriétés à l'aide de ces bibliothèques qui optimisent souvent l'efficacité de l'évaluateur. Nous détaillerons cependant ici les fonctions déjà existantes pour aider à la compréhension des modalités et illustrer le fonctionnement du langage.

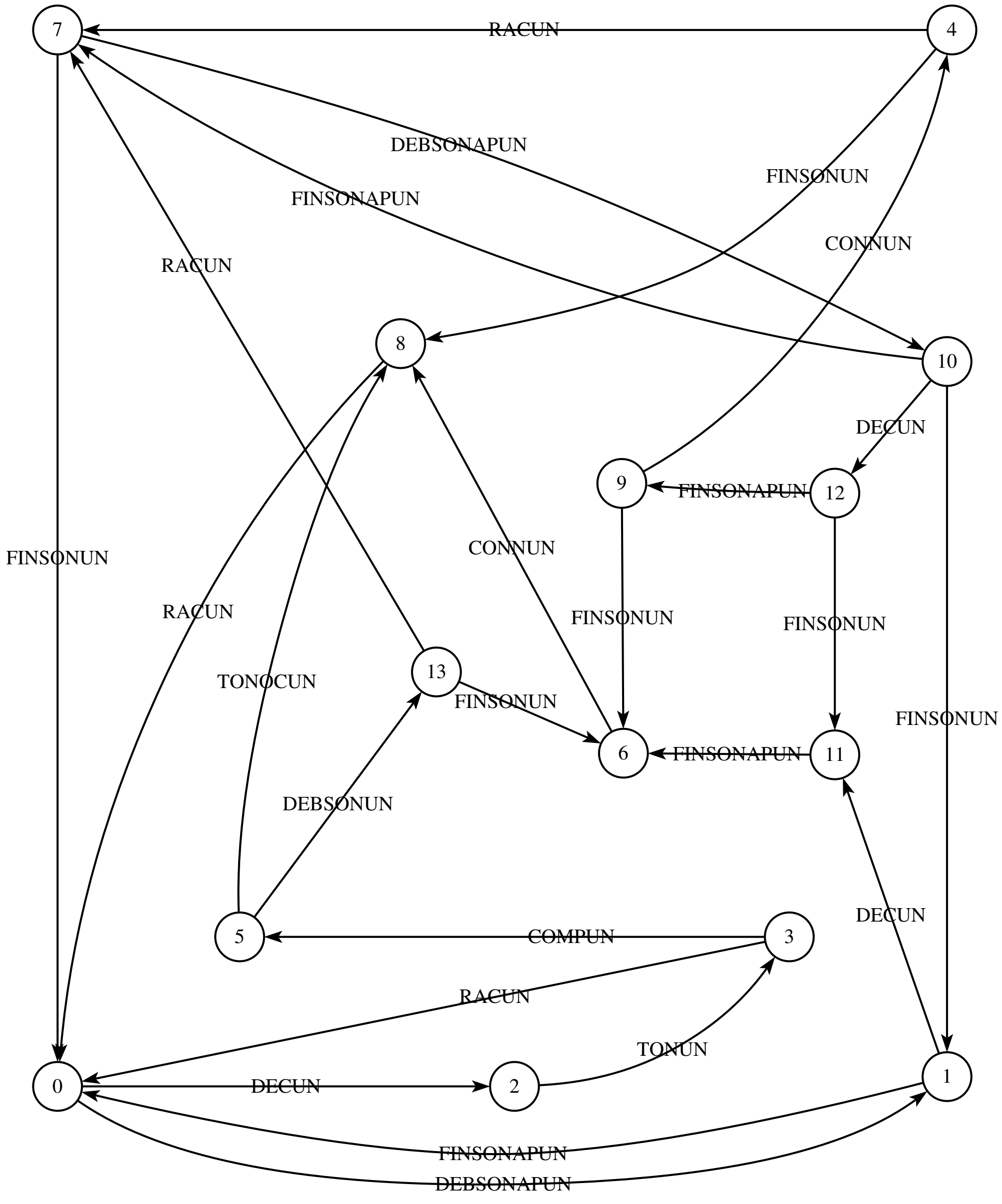


Figure 9: Aldébaran- Propriété 6

3.3.1 Préliminaire

La propriété est consignée dans un fichier *prop*.`xtl`. Ce fichier est tout simplement passé en paramètre à XTL ainsi que le STE au format BCG *spec*.`bcg` de la façon suivante :

```
#> xtl -verbose -french prop.xtl spec.bcg
```

La sortie d'erreur n'est souvent pas assez explicite mais suffit avec un peu de connaissance du langage.

On peut donner un cheminement à suivre pour aider à l'implémentation d'une propriété. On commence bien entendu par formuler la propriété de manière informelle. L'utilisateur d'XTL se posera les questions suivantes :

- *La formulation de ma question est–elle correcte ?* On ne se demande pas ici si je formule ce que je souhaite vérifier, mais si la formulation est la plus adaptée pour la transcrire en XTL. Si les questions qui suivent ne peuvent déterminer la ou les modalités qui nous permettrait d'écrire notre formule, c'est en effet la question à se poser. Il est idéal de formuler la question en terme de séquence d'étiquettes ou de contraintes sur des suites états par exemple. De plus comme nous le verrons dans la propriété 4, le fait de transformer une quantification universelle en négation d'une quantification existentielle peut simplifier considérablement les choses.

- *Ma propriété est–elle une propriété universelle ou existentielle ?* Il est clair que cette question permettra de cibler les modalités qui nous intéressent. Cette question est en vérité double car il s'agit de savoir si une des modalités `AG`, `EF`, `EG` ou `AF` de la logique ACTL est applicable. Il y a donc la question de la quantification sur les chemins et celles sur les états.

- *Quelles sont les étiquettes de transitions mise en jeu ?*

- *Y–a–t'il des étiquettes que je vais pouvoir regrouper en une seule catégorie pour faciliter l'écriture de ma propriété ?* Celles–ci seront regroupées dans des ensembles d'étiquettes à l'aide de la correspondance d'étiquettes (*Label Matching*). La macro `EVAL_A()` nous servira à cela.

- *Dois je tenir compte des τ–transitions ?*

La question des étiquettes cachées ne se pose pas. Nous utiliserons en effet une specification normale (utilisant le `hide` de LOTOS) lorsque nous ne travaillons que sur des étiquettes apparentes à un utilisateur et une spécification transparente (sans étiquette dissimulée par un `hide`) lorsque certaines le sont. Bien sûr ces questions ne suffisent pas à déterminer exactement la façon d'implémenter les formules mais donne un cadre afin de trouver plus facilement les modalités à employer. Il faudra parfois procéder en découpant la formule en formules plus élémentaires et se reposer la série de questions.

3.3.2 Propriété 7

1. "On ne peut pas décrocher deux fois d'affilée *i.e.* sans avoir raccroché entre temps."
2. Nous allons reformuler notre énoncé ainsi : "Tout chemin qui ne possède pas d'étiquette de transition `DEC !x` ne mène pas à un état pouvant exécuter `RAC !x` "
3. La propriété est clairement un énoncé universel. Elle porte sur tous les états de tous les chemins vérifiant une propriété. On utilisera la modalité `AG_A` de la logique ACTL.
4. Les étiquettes nous concernant son `DEC !1` et `RAC !1`. Comme pour l'étude avec ALDÉBARAN, on utilisera la symétrie entre les trois utilisateurs pour ne traiter qu'un cas.
5. Implémentation :

```
let DEC : labelset = EVAL_A(DEC !1),
    RAC : labelset = EVAL_A(RAC !1) in
        AG_A ( not(DEC), [RAC]false )
end_let
```

6. On calcule ici l'ensemble des états qui vérifie que tous les chemins n'ayant pour étiquettes que des éléments de `non(DEC)` (le complémentaire de `DEC`) ne peut mener qu'à un état qui vérifie `[RAC]false` *i.e.* que `RAC` n'est pas exécutable.
7. Résultat : La propriété est vérifiée.

3.3.3 Propriété 8

1. "Un téléphone ne peut sonner que s'il est raccroché."
2. Nous allons reformuler notre énoncé ainsi : "À partir d'un certain état duquel on peut effectuer une transition étiquetée par `DEBSON !x !y` et en considérant les transitions qui ont précédées celle–ci, on trouve toujours une transition étiquetée par `RAC !y` avant une étiquettée par `DEC !y`. "
3. Encore mieux en considérant plutôt l'arbre en descendant : "On ne peut pas trouver une séquence de transitions commençant par l'étiquette `DEC !y` et terminant par `DEBSON !x !y` sans qu'il y ait de transition d'étiquette `RAC !y` exécutée dans cette séquence."
4. Notre propriété est constitutée de plusieurs énoncés universels :
   - Tous les états vérifient que s'ils exécutent `DEC !y`

- l'état cible vérifiera que tout chemin qui n'exécute pas `RAC !y`
- mène à un état qui ne peut pas exécuter `DEBSON !x !y`.

5. Les étiquettes qui nous intéressent sont clairements apparentes.

6. La propriété sera implémentée par :

```
let DEC : EVAL_A (DEB !1),
    RAC : EVAL_A (RAC !1),
    DSO : EVAL_A (DEBSON !1 ?x:integer where x among {1 ... 3} in
        BOX( DEC, AG_A( not(RAC), BOX(DSO, false) )
end_let
```

7. `BOX(A,F)` correspond à la modalité qui calcule l'ensemble des états à partir desquels toutes les exécutions de `A` mènent à un état vérifiant `F`.

8. Résultat : La propriété est vérifiée.

9. Remarquons qu'il existe une macro qui permet d'implémenter exactement cela, c'est `NOT_TO_UNLESS(A,B,C)`.

3.3.4 Propriété 9

1. On veut considérer le fait que le premier qui appelle doit avoir la communication en priorité sur ceux qui appelent après.

2. On veut vérifier la propriété "Si on peut exécuter `ACQPAR !y !x` c'est qu'auparavant toute exécution de `COMP !z !y` est suivie (pas forcément immédiatement) de l'exécution de `ACQPAR !y !z` ou de `ESTOC !y !z`. "

3. On peut reformuler cette propriété par "Il n'y a pas de séquence de transitions commençant par l'étiquette `COMP !z !y` et terminant par `ACQPAR !y !x` sans qu'il y ait de transition d'étiquette `ACQPAR !y !z` ou `ESTOC !y !z` exécutées dans cette séquence."

4. On note que ce sont là des étiquettes dissimulées. On vérifiera la propriété sur la spécification transparente.

5. La macro définie pour la propriété 8 est parfaite ici :

```
forall x : integer among { 1 ... 3 },
       y : integer among remove({ 1 ... 3 },x),
       z : integer among remove(remove({ 1 ... 3 },x),y) in
     NOT_TO_UNLESS( EVAL_A(COMP !z !y), EVAL_A(ACQPAR !y !x,
               EVAL_A(ACQPAR !y !z) or EVAL_A(ESTOC !y !z) )
end_forall
```

6. Résultat : La propriété est fausse.
7. En effet celle–ci a été mal formulée. Nous ne devrions pas considérer les étiquettes `COMP !z !y` située après `COMP !x !y`.
8. On voit là un des problèmes du langage. Celui–ci ne donne pas de moyen simple de représenter sa sémantique. Les états ne sont indiqués que par leurs numéros et si on peut calculer les états violants la propriété on n'en est pas pour autant plus avançé.

3.3.5 Propriété 10

1. On veut toujours vérifier le fait que l'appel revient au premier qui appelle en priorité.
2. Il faut que notre propriété ne tienne compte que des personnes qui ont appelé avant que `x` ne compose le numéro de `y`. Nous allons donc en calculant les états qui violent notre condition, *i.e.* les états à partir desquelles il existe une séquence débutant par `COMP !3 !2` finissant par `ACQPAR !2 !3` contenant `COMP !1 !2` et ne contenant pas ni `ACQPAR !2 !3` ni `ESTOC !2 !3`. La figure 10 montre une telle séquence.
3. On va implémenter cette propriété avec l'opérateur `implies` sur les ensembles.

```
not(
  Dia( EVAL_A(COMP!3!2),
       EG_A( not( EVAL_A(ACQPAR!2!3) or EVAL_A(ESTOC!2!3)),
             Dia( EVAL_A(COMP!1!2),
                  EG_A( not( EVAL_A(ACQPAR!2!3) or EVAL_A(ESTOC!2!3)),
                        Dia( EVAL_A(ACQPAR!2!1),
                             true
))))))
```

4. Résultat : notre propriété est vraie. Donc aucun état ne permet d'exécuter une telle séquence.

$$\xrightarrow{\texttt{COMP!3!2}} \quad \overset{\texttt{X}}{\dashrightarrow} \quad \xrightarrow{\texttt{COMP!1!2}} \quad \overset{\texttt{Y}}{\dashrightarrow} \quad \xrightarrow{\texttt{ACQPAR!2!1}}$$

$$\texttt{x, y} \notin \{\texttt{ACQPAR!2!3, ESTOC!2!3}\}$$

Figure 10: Exemple de séquence vérifiant la propriété 10

3.3.6 Bilan

En somme, malgré parfois la difficulté de modéliser une propriété de façon simple et efficace l'outil XTL est globalement très efficace. Son exécution est relativement rapide à coté de nombreux outils dévoués aux mêmes tâches. Son utilisation permet normalement de formuler simplement des propriétés complexes à l'aide des nombreuses modalités déjà existantes. La dernière propriété en est une illustration.

3.4 Conclusions et remarques

Dans notre étude de cas, nous avons pu remarquer qu'ALDÉBARAN et XTL ont su s'allier pour démontrer de nombreux résultats. ALDÉBARAN a été particulierement efficace pour traiter les cas des interfaces globales en toute simplicité alors que l'utilisation de XTL, comme en témoigne la septième propriété, demande plus d'effort pour le même résultat. En contrepartie l'utilisation d'XTL semble s'imposer lorsque la propriété devient complexe à représenter avec un simple STE ; c'est le cas lorsque nous avons voulu prouver des propriétés portant sur l'interaction de l'ensemble des utilisateurs (la dernière propriété en est un exemple).

Dans les deux cas, une analyse préalable à pu permettre une simplification du problème afin d'éviter de vérifier des propriétés trop semblables (la symétrie de chaque utilisateur vis–à–vis des autres). Ceci montre que pour optimiser les performances et limiter le temps de vérification, une analyse mathématique même sommaire n'est pas négligeable.
On peut conclure finalement que la boîte à outils CADP est complète et efficace. Bien que n'étant pas limitée à un domaine précis (le langage LOTOS n'est pas obligatoire, seulement il permet d'utiliser les outils dédiés), elle sait conserver de bonnes performances.

Notons que la boîte possède de nombreux outils que nous n'avons pu tester et qui s'avèrent complémentaires à tous ceux dejà utilisés.

Annexe : specPots.lotos

```
(*
 * Specification d'un systeme telephonique en Lotos.
 * Marc Frappier
 * Universite de Sherbrooke
 *
 *
 *
 *
 * Si deux utilisateurs appellent le meme telephone,
 * le premier qui a compose le numero obtiens la communication.
 *
 * Par exemple, pour la sequence
 *     dec<n1>,dec<n2>,comp<n1,n3>,comp<n2,n3>
 * il y a une seule suite possible:
 *     debutSonnerie<n1,n3>,tonOccupe<n2>
 *)

specification systeme_telephonique[
        dec, rac,
        comp,
        ton, tonOc,
        debSon, finSon,
        acq, lib, estOc, acqPar,
        conn, decon
    ] : noexit

library
    Boolean
endlib

type Nat_number
   is Boolean

   sorts Nat

   opns 0 (*! constructor *),
        1 (*! constructor *),
        2 (*! constructor *),
        3 (*! constructor *),
        4 (*! constructor *) : -> Nat
        _eq_ , _ne_ : Nat,Nat -> Bool
   eqns
        forall x,y : Nat
        ofsort Bool
```

```
          0 eq 0 = true ;
          1 eq 1 = true ;
          2 eq 2 = true ;
          3 eq 3 = true ;
          4 eq 4 = true ;
          0 eq 1 = false ;
          0 eq 2 = false ;
          0 eq 3 = false ;
          0 eq 4 = false ;
          1 eq 2 = false ;
          1 eq 3 = false ;
          3 eq 4 = false ;
          2 eq 3 = false ;
          2 eq 4 = false ;
          3 eq 4 = false ;
          x eq y = y eq x ;
          x ne y = not(x eq y) ;

endtype

type seq_nat_number (* creation d'un nouveau type : sequence *)
   is Nat_Number (* definition abstraite *)

   sorts seq_nat

   opns  empty (*! constructor *) : -> seq_nat
         add (*! constructor *) : nat,seq_nat -> seq_nat
         tail : seq_nat -> seq_nat
         remove : nat,seq_nat -> seq_nat
         head : seq_nat -> nat

   eqns  forall x,y:nat, q:seq_nat
         ofsort nat
            head( empty ) = 0 ;
            head( add(x, empty) ) =  x ;
            head( add(x , add(y,q)) ) = head( add(y,q) ) ;
         ofsort seq_nat
            tail( add(x, q) ) = q ;
            remove(x , empty) = empty ;
            x ne y => remove( x, add( y, q ) ) = add( y, remove( x, q ) ) ;
            remove( x, add( x, q ) ) = remove( x, q ) ;

endtype (* fin du nouveau type*)

behaviour
```

```
    hide acq, acqPar, lib, estOc in
        tousAppels[dec, rac,comp,ton, tonOc,debSon, finSon,
                   acq, lib, estOc, acqPar,conn, decon]
    |[acq, lib, estOc, comp, acqPar]|
        controleTousAppels[comp, acq, lib, estOc, acqPar]

where

    process tousAppels[dec, rac,comp,ton, tonOc,debSon, finSon,
                       acq, lib, estOc, acqPar,conn, decon]:noexit :=

             appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                  acq, lib, estOc, acqPar,conn, decon](1)
        |||
             appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                  acq, lib, estOc, acqPar,conn, decon](2)
        |||
             appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                  acq, lib, estOc, acqPar,conn, decon](3)

    endproc

    process controleTousAppels[comp, acq, lib, estOc, acqPar]:noexit :=

            controleAppel[comp, acq, lib, estOc, acqPar](1, true, empty)
        |||
            controleAppel[comp, acq, lib, estOc, acqPar](2, true, empty)
        |||
            controleAppel[comp, acq, lib, estOc, acqPar](3, true, empty)

    endproc

    process controleAppel[comp, acq, lib, estOc, acqPar]
                              (n1 : Nat, libre : Bool, f : seq_nat): noexit :=

            [libre] ->
                acq!n1;
                controleAppel[comp, acq, lib, estOc, acqPar](n1, false, f)
        []
            [not(libre)] ->
                estOc!n1?n2:Nat;
                controleAppel[comp, acq, lib, estOc, acqPar](n1,libre, remove(n2,f))
        []
            [not(libre)] ->
                lib!n1;
                controleAppel[comp, acq, lib, estOc, acqPar](n1,true, f)
```

```
    []
        comp?n2:Nat!n1;
        controleAppel[comp, acq, lib, estOc, acqPar](n1,libre,add(n2,f))
    []
        [libre] ->
            acqPar!n1?n2:Nat[n2 = head(f)];
            controleAppel[comp, acq, lib, estOc, acqPar](n1,false,tail(f))

endproc

process appel[dec, rac,comp,ton, tonOc,debSon, finSon,
              acq, lib, estOc, acqPar,conn, decon](n1:Nat): noexit :=

    acq!n1;
    dec!n1;
    ton!n1;
    (
        rac!n1;
        lib!n1;
        appel[dec, rac,comp,ton, tonOc,debSon, finSon,
              acq, lib, estOc, acqPar,conn, decon](n1)
    []
        comp!n1?n2:Nat;
        (
            estOc!n2!n1;
            tonOc!n1;
            rac!n1;
            lib!n1;
            appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                  acq, lib, estOc, acqPar,conn, decon](n1)
        []
            acqPar!n2!n1;
            debSon!n1!n2;
            (
                rac!n1;
                lib!n1;
                finSon!n1!n2;
                lib!n2;
                appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                      acq, lib, estOc, acqPar,conn, decon](n1)
            []
                dec!n2;
                finSon!n1!n2;
                conn!n1!n2;
                (
                    rac!n1;
```

```
                        decon!n1!n2;
                        lib!n1;
                        rac!n2;
                        lib!n2;
                        appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                              acq, lib, estOc, acqPar,conn, decon](n1)
                    []
                        rac!n2;
                        decon!n1!n2;
                        lib!n2;
                        rac!n1;
                        lib!n1;
                        appel[dec, rac,comp,ton, tonOc,debSon, finSon,
                              acq, lib, estOc, acqPar,conn, decon](n1)
                    )
                )
            )
        )
    endproc
endspec
```