Mémoire de magistère

Denis Oddoux

septembre 1998 -

septembre 2001

MMFAI : Magistère de Mathématiques Fondamentales et Appliquées et d'Informatique

Contenu

- Cursus Universitaire
- Rapport de stage de maîtrise
- Rapport de stage de D.E.A.
- Article (en anglais) publié au cours de la thèse

Partie 1

Cursus universitaire

Licence d'informatique
septembre 1998 - janvier 1999

Cours d'informatique

- Algorithmique et programmation (J. Stern)
- Langages de programmation et compilation (P. Cousot)
- Théorie de l'information (J. Vuillemin)
- Langages formels, calculabilité, complexité et analyse d'algorithmes (P. Gastin)

Cours de maths-informatiques

- Logique (A. Louveau)

Cours de mathématiques

- Intégration et probabilités (F. Comets)

Maîtrise d'informatique

février - septembre 1999

Cours d'informatique

- Sémantique (P. Cousot)
- Systèmes et réseaux (J. Beigbeder)
- Programmation orientée objet (P. Bertin, F. Bourdoncle)
- Réalisation de systèmes matériels (J. Vuillemein, M. Shand)

Stage de maîtrise

Conception et réalisation d'un système d'évaluation en ligne sur le web

de juin à septembre 1999

sous la direction de V. Bouthors et O. Dedieu

D.E.A. d'Algorithmique

septembre 1999 - juin 2000

Cours de tronc commun

- Algorithmique de la téléphonie mobile (D. Krob)
- Automates finis et applications (A. Petit)
- Calcul formel (D.Lazard)
- Complexité (A. Muscholl)

Cours de filières

- Algorithmique de la vérification (P. Gastin, F. Laroussinie)
- Complexité, codage et cryptographie (J. Stern, J.J. Quistater, P. Charpin, G. Hanrot, F. Morain, M. Santha)
- Automates et mots (C. Choffrut, C. Frougny, A. Muscholl)

Stage de D.E.A. (voir thèse)

d'avril à juin 2000
sous la direction de P.Gastin

Thèse d'informatique
septembre 2000 -

Preuves automatiques de systèmes par model-checking et logique temporelle linéaire

Sous la direction de P. Gastin,
au LIAFA (Laboratoire d'Informatique Algorithmique : Fondements et Applications),
université de Paris VII (Jussieu)

Le stage consistait à comprendre le fonctionnement d'un outil de vérification nommé Spin, produisant à partir d'une formule de logique temporelle linéaire un automate de Büchi reconnaissant le même langage. Après avoir étudié l'algorithme utilisé puis cherché à l'améliorer, le stage a consisté à trouver et implémenter un algorithme plus efficace faisant le même travail. Les résultats expérimentaux sont très satisfaisants.

Le stage de D.E.A. a débouché sur une thèse. Au cours de cette année, l'outil développé au cours du stage a été amélioré, permettant des gains de performance très importants. Le résultat des recherches a été exposé au cours de plusieurs séminaires et groupes de travails, puis soumis à une conférence. Ayant été accepté, un article a été publié en juillet 2001.

Partie 2

Maîtrise d'informatique

Rapport du stage de maîtrise
sous la direction
de V. Bouthors et O. Dedieu

juin à septembre 1999

Rapport de stage

Denis Oddoux

juin - septembre 1999

Table des matières

1 Introduction

Ce stage s'inscrit dans le cadre du projet *Pharos*[BD99], développé par Bull et l'INRIA.

Pharos offre une infrastructure de collaboration qui permet à des utilisateurs de cataloguer et d'évaluer des documents au sein d'un canal, associé à un sujet donné. Ces données, ou annotations, sont structurées. Elles sont synthétisées afin de produire des recommandations personnalisées.

Le but du stage est de proposer aux administrateurs d'un canal de personnaliser la structure de ses annotations, afin de l'adapter au thème du canal. Une annotation sera donc l'instance d'une classe qui aura été générée puis compilée dynamiquement. Chaque annotation sera enregistrée sur le disque sous le format ASCII XML[BPSM98]. Une interface graphique sera créée pour modifier aisément les propriétés d'un canal et la structure de ses annotations, la structure choisie étant aussi stockée au format XML.

2 Description du projet *Pharos*

Le Web est la plus grande source d'informations au monde, mais n'est pas une bibliothèque ; la pertinence et la qualité des documents qu'on y trouve est très variable. Les utilisateurs veulent savoir ce que les experts pensent d'un document, et pouvoir partager leur opinion avec d'autres utilisateurs.

Le projet *Pharos* a été développé pour permettre aux utilisateurs du Web de partager la connaissance qu'ils en ont. La base de données est gérée par des serveurs dispersés sur le réseau, avec un système de réplication. Une expérimentation est actuellement menée, par les responsables du projet d'une part, et par le CNET d'autre part, qui souhaite utiliser *Pharos* dans l'Éducation nationale.

2.1 Annotations et recommandations

Une *annotation* est une donnée structurée publiée par un utilisateur à propos d'un document Web donné (identifié par son URL). Cette donnée contient l'URL de la page, un titre, une note, éventuellement des mots-clés, et diverses informations qui caractérisent le contenu du document. Les mots-clés sont choisis dans un dictionnaire défini par les utilisateurs autorisés.

Une *recommandation* est une synthèse de toutes les annotations publiées par les différents utilisateurs à propos d'une même page. La note est la moyenne des autres notes, éventuellement pondérée par l'utilisateur suivant la confiance accordée aux autres utilisateurs du système. Elle permet ainsi de recevoir à partir d'annotations subjectives un avis proche du profil de l'utilisateur.

2.2 Canaux

Un *canal* matérialise une communauté d'utilisateurs partageant un intérêt commun, comme l'art, la musique, ou les jeux vidéo. Un canal est autonome, il gère seul sa base de données (utilisateurs, annotations, et mots-clés). Il peut donc tourner sur un serveur différent des autres canaux, ou être stoppé provisoirement pour une mise à jour. Il peut aussi être répliqué sur un autre serveur, avec des protocoles réguliers de synchronisation (Fig. 1). La structure des annotations publiées sur ce canal lui est propre, ce qui permet aux utilisateurs de donner des informations en rapport avec le thème du canal. Il définit aussi la liste des utilisateurs ainsi que leurs droits.

2.3 Assistant et accès Web

La manière privilégiée d'interagir avec les canaux est d'utiliser l'*assistant Pharos*. Fonctionnant en relation avec le navigateur de l'utilisateur, il liste les canaux disponibles, et permet de s'inscrire

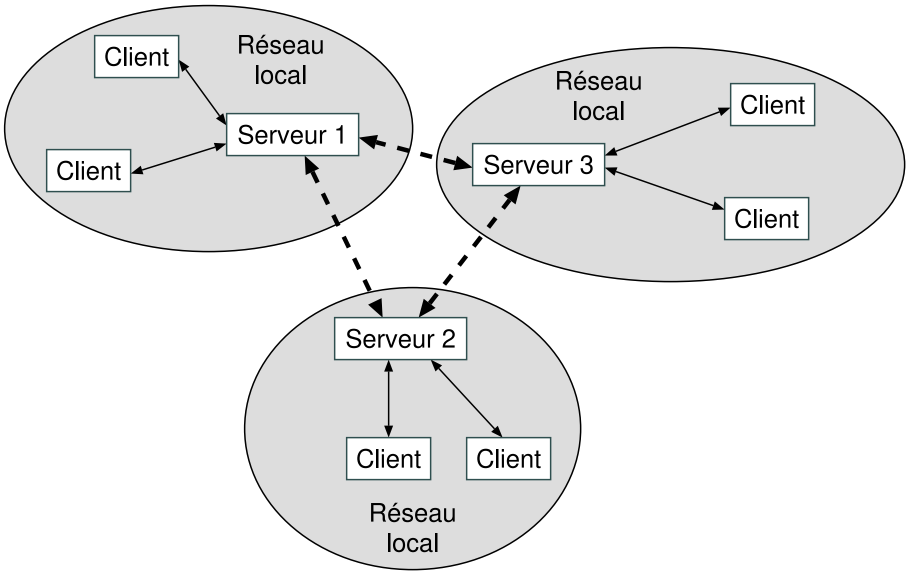

FIG. 1 – *Architecture de réplication dans Pharos*

à un canal, de paramétrer ses préférences, de publier une annotation, de consulter les annotations associées à une page donnée, et enfin de faire des requêtes dans la base de données.

L'assistant est utilisé comme proxy HTTP[Ded98] pour le navigateur Web (Fig. 2), étant ainsi compatible avec la plupart des navigateurs existants. Il observe les requêtes Web effectuées, et permet de consulter des annotations à propos de la page affichée sur le navigateur ou des pages précédemment visitées. Il offre aussi le moyen de poser une annotation ou de faire des requêtes.

D'autre part, un accès Web a été développé pour offrir une plus grande visibilité des canaux : on s'inscrit grâce à une adresse e-mail, et des fomulaires permettent de publier une annotation. On peut aussi effectuer une requête, simple ou avancée, et consulter ainsi les annotations publiées.

2.4 Requêtes

Les *requêtes* s'effectuent par l'accès Web ou avec l'assistant. On peut chercher les documents annotés avec certains mots dans les champs texte comme le titre et les commentaires, une note, des mots-clés, ou des utilisateurs particuliers. Tous ces critères permettent de sélectionner une liste d'annotations, regroupées en recommandations comme expliqué précédemment, ou triées selon le choix de l'utilisateur : par note, par mots-clés, par auteur, par date de publication, ou par titre. Le résultat est affiché dans le navigateur, avec de nombreux liens permettant d'effectuer d'autres requêtes simples ou de raffiner la recherche.

2.5 Utilisation

L'assistant *Pharos* est diffusé librement sur Internet, avec des modules additionnels profitant du rôle de proxy de cet assistant, comme un filtre à publicités ou un traceur du traffic[Ded98].

Pour utiliser leur propre serveur, les administrateurs peuvent choisir leur politique : employer un groupe de personnes qui annoteront pour les utilisateurs (c'est la solution retenue par le CNET), ou permettre aux utilisateurs de poser leurs propres annotations, afin de partager les points de vue.

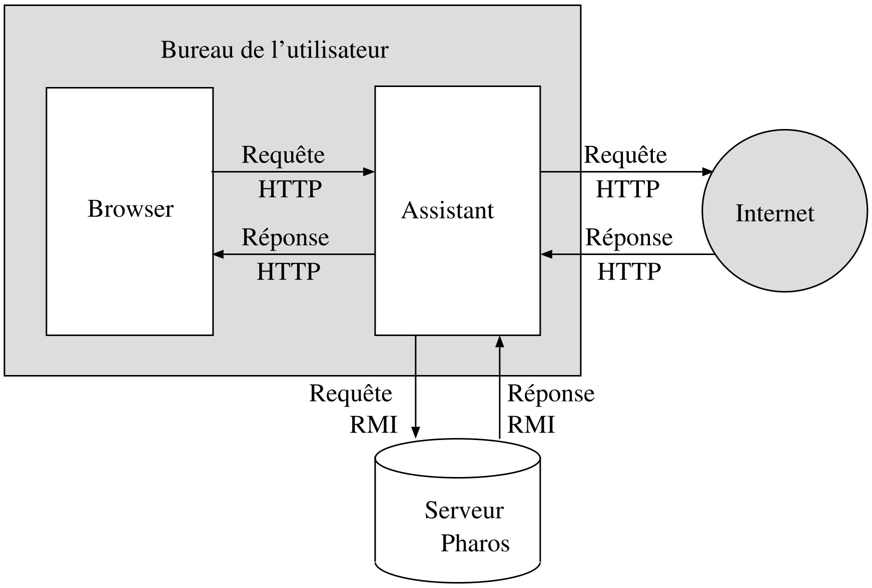

FIG. 2 – *Structure de l'outil Pharos*

2.6 Structure

L'assistant et le serveur *Pharos* sont programmés en Java. Ce langage permet une portabilité du code sur de nombreux systèmes. Les interfaces graphiques utilisent le package Swing[WC98] de Java.

Le serveur *Pharos* gère l'administration des canaux. Chaque canal démarré est un nouveau serveur qui communique avec ses clients.

L'assistant est le client du serveur *Pharos*. Il sert d'interface graphique pour administrer le serveur et créer les client des canaux qui possèdent leur propre interface.

La communication entre le serveur *Pharos* et un assistant, et celle entre un client et le serveur d'un canal se font par le protocole RMI[WRW96] (Fig. 3). Au niveau de *Pharos*, l'assistant s'informe des canaux disponibles, vérifie la version qu'il possède d'un canal et se met éventuellement à jour, crée, modifie ou détruit un canal. Au niveau d'un canal, le client propose la création, la modification ou la destruction d'annotations, de membres ou de mots-clés, ainsi que la liste des annotations disponibles sur une URL donnée.

2.7 Interface graphique du client d'un canal

Cette interface graphique est constituée de quatre principaux éléments :

- un visualiseur d'annotations, qui affiche toutes les annotations publiées par les différents utilisateurs à propos d'une URL donnée;
- un éditeur d'annotations, qui permet de publier sa propre annotation sur l'URL de son choix;
- une interface de requêtes permettant de choisir de nombreux critères de recherche;
- une section permettant de définir le profil de l'utilisateur, où il paramètre ses préférences et la confiance qu'il accorde à chaque membre du canal.

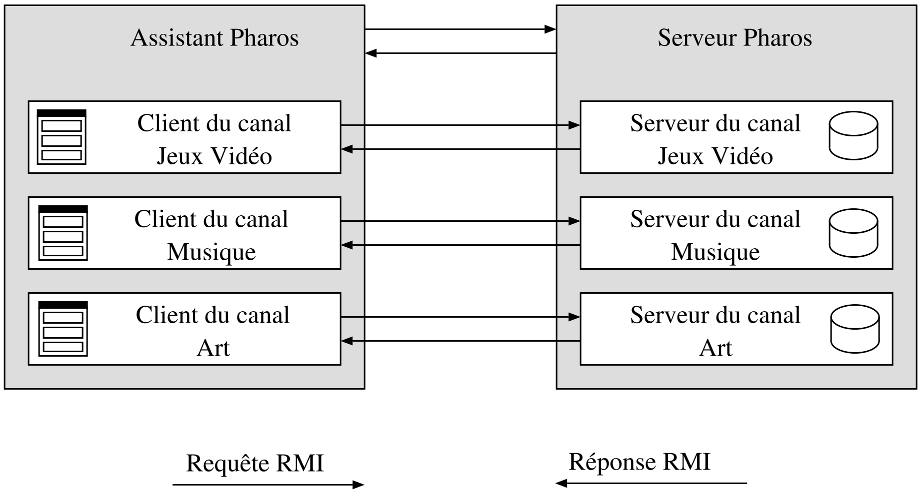

FIG. 3 – *Communications assistant/serveur*

3 But et déroulement du stage

Initialement, les annotations possédaient une structure fixe : une URL, un titre, une note, un commentaire et des mots-clés. Le but du stage était de proposer à la personne qui crée un canal de choisir les caractéristiques du canal et la structure des annotations.

Ainsi le stage s'est découpé en plusieurs étapes :

- finalisation des spécifications liées au stage
  1. choix des propriétés proposées pour un canal;
  2. choix des types des champs que peut regrouper une annotation;
  3. choix du format de sauvegarde de la structure d'un canal;
  4. choix du format de manipulation des annotations en mémoire;
  5. choix d'un protocole permettant la mise à jour du code des clients d'un canal.
- programmation effective du projet
  1. programmation de l'interface graphique de création et de modification d'un canal;
  2. programmation de la sauvegarde des propriétés choisies;
  3. programmation de la génération du code Java
  4. intégration à la version existante.

4 Précision des données à définir au début du stage

Ces données n'ayant pas été définies clairement avant le début du stage, une partie de celui-ci a été consacrée à finaliser les besoins avec les responsables du projet.

4.1 Propriétés d'un canal

La flexibilité des propriétés d'un canal avait déjà été prise en compte dans le code existant, mais les modifier nécessitait d'éditer le fichier de configuration des canaux en dehors de l'interface de l'assistant. Une liste de propriétés susceptibles d'être modifiées a donc été établie :

- nom du canal et descriptif;
- propriétés de l'accès Web :
  - existence ou non d'un acces Web;
  - nécessité d'un mot de passe pour consulter les annotations;
  - lien hypertexte vers l'accès Web du canal dans la liste proposée par le serveur *Pharos* sur l'accès Web;
- langue du canal;
- présence d'un dictionnaire de mots-clés dans les annotations.

4.2 Champs des annotations

Il a été convenu de laisser à l'administrateur d'un canal le choix de la structure graphique des interfaces de visualisation et d'édition d'annotations sur l'assistant. Ainsi c'est lui qui dispose les champs, en les agençant verticalement à la suite les uns des autres sur une ou plusieurs pages. Il sera donc fait en sorte que l'ordre et la répartition des champs soient aisément modifiables par la personne qui crée le canal.

Chaque champ possèdera un nom affiché, un nom interne pour la sauvegarde XML, un type, ainsi que des attributs spécifiques à chaque type, permettant de déterminer le format d'affichage, les contraintes de saisie ou le comportement lors des requêtes (Fig. 4).

4.3 Format de sauvegarde de la structure des annotations

La structure des annotations est sauvée dans un fichier ASCII. Elle peut ainsi être comprise aisément par des outils externes et même être modifiée manuellement si besoin est. Pour un souci de facilité de compréhension, les données sont écrites en XML, un format adapté à la sauvegarde de données structurées sous forme de texte (Fig. 5). D'autre part, utiliser un format connu permet de profiter de *parsers* disponibles et facilite la relecture pour la création de l'interface de modification du canal.

4.4 Format de manipulation en mémoire des annotations

Le problème est de choisir un format propice à la manipulation des annotations en mémoire, permettant une facilité de traitement et une manipulation rapide, tout en offrant une grande flexibilité dans la structure de l'annotation.

La première idée est de sauver le résultat dans une table associant au nom du champ sa valeur, des tests montrent que l'accès à une donnée d'une telle table est 70 fois plus lent que l'accès à la valeur d'un champ dans un objet. De même, rédiger les données au format XML est souple mais lent car il faut parser et interpréter le résultat. La solution trouvée consiste à générer du code Java adapté aux besoins de traitement des annotations.

texte :

- taille de la zone de saisie;
- taille de l'affichage dans une requête (les champs trop longs doivent être tronqués pour une meilleure lisibilité; un lien propose toujours l'annotation non tronquée avec tous ses champs);
- obligation de remplir ce champ dans toute annotation;
- participation de ce champ dans les recherches de texte.

date :

- nature de la date : future, passée ou quelconque;
- affichage ou non dans une requête;
- obligation de remplir ce champ dans toute annotation.

case à cocher :

- valeur par défaut;
- affichage ou non dans une requête.

liste de choix :

- valeurs possibles;
- valeur par défaut;
- affichage ou non dans une requête.

note :

- pondération pour la note moyenne;
- affichage ou non dans une requête.

FIG. 4 – *Propriétés des champs des annotations d'un canal*

```
<?xml version="1.0" ?>
<channel description="This channel is dedicated to the XML language"
         enabled="true" public="false" visible="true" language="fr" thesaurus="true">
  <tab name="Main">
    <field name="interest" type="score" display="true" weight="3" />
    <field name="performance" type="score" display="true" weight="1" />
    <field name="comments" type="text" display="240" required="true"
           searchable="true" input="4" />
    <field name="language" type="choice" display="true"
           values="English|French|Other" default="English" />
    <field name="examples in Java" type="check" display="true" default="false" />
    <field name="date of creation" type="date" display="false" range="past"
           required="false" />
  </tab>
  <tab name="Others">
    ...
  </tab>
</channel>
```

FIG. 5 – *Exemple de description de la structure des annotations d'un canal*

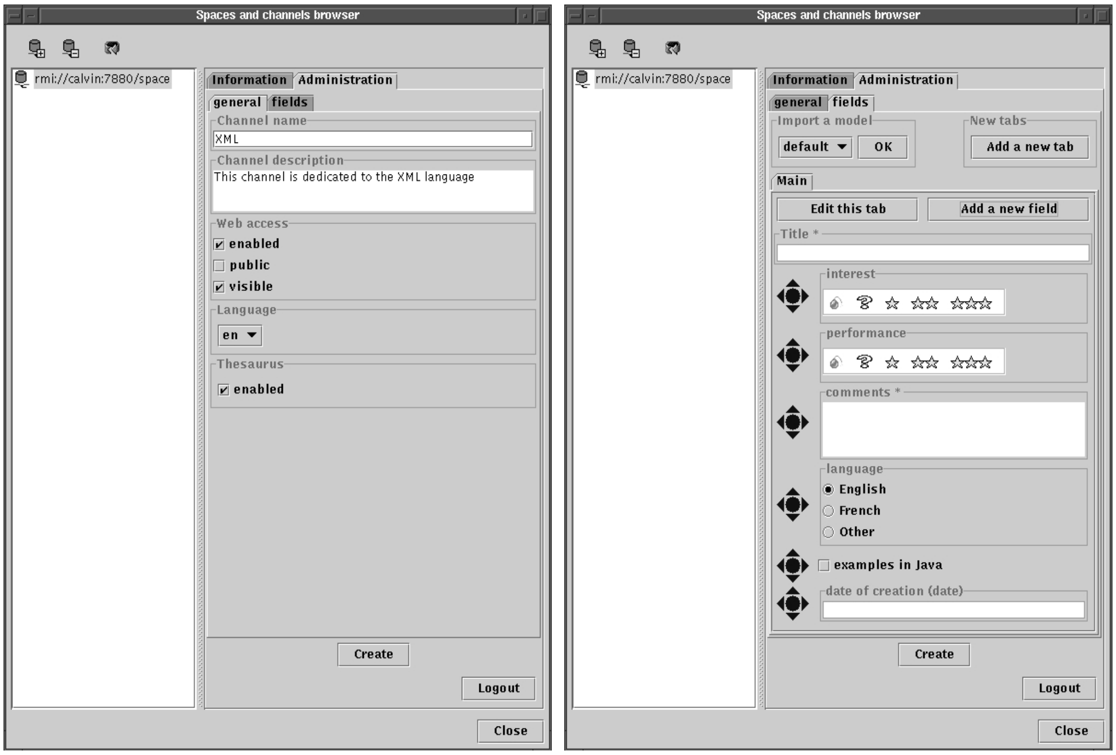

FIG. 6 – *Interface graphique de création d'un canal : à gauche, les propriétés ; à droite, les champs des annotations*

Ainsi, lors de la création ou de la modification d'un canal, le code permettant de représenter l'objet *annotation* est généré, avec des méthodes adaptées à sa manipulation, comme l'affichage au format HTML. Java permet ensuite de le compiler et de le charger dynamiquement en mémoire.

Cependant, il est impossible de préciser aux outils RMI où se trouve le code des annotations, et ce protocole ne permet donc pas de passer un tel objet de l'assistant au serveur. Pour la communication, les annotations sont donc transformées en tables comme décrit précédemment.

4.5 Protocole de mise à jour de l'interface de l'assistant

Chaque utilisateur possède le code de l'assistant et des clients des canaux. Ainsi, à chaque fois que la structure d'un canal est modifiée, il faut que les utilisateurs de ce canal reçoivent les modifications.

Or *Pharos* est un système distribué, l'accès ne se fait pas toujours par le réseau local. De plus, les utilisateurs peuvent être nombreux. C'est pourquoi il faut faire les mises à jour de façon paresseuse : lorsqu'un utilisateur se connecte sur un canal, il précise la version du code qu'il possède. Si celui-ci n'est pas à jour, il demande au serveur un fichier compressé contenant le nouveau code. L'assistant installe le nouveau code, le charge, l'initialise et se connecte sur le canal.

Par ailleurs, lorsqu'un canal est arrêté, on invalide l'objet qui permettait la communication RMI, en renvoyant une exception à chaque requête. Ainsi, à la prochaine requête de l'assistant, celui-ci sait que le canal a été arrêté, et quitte l'interface. Lorsque celle-ci est redémarrée, la nouvelle version est automatiquement chargée.

5 Programmation

5.1 Interface de création et de modification d'un canal

Les interfaces graphiques de création et de modification d'un canal sont sensiblement identiques. Celle de modification est pré-remplie avec les paramètres du canal, ceci après relecture du fichier XML de structure du canal. Pour simplifier la création d'un canal, des modèles standards d'annotations sont proposés.

Une première zone permet de saisir les propriétés du canal, la deuxième permet d'organiser les champs des annotations : ceux-ci sont répartis sur des onglets, et une icone permet de déplacer un par un les champs verticalement ou d'une intercalaire à la suivante, ainsi que de les modifier ou de les effacer (Fig. 6).

Après confirmation, le nouveau fichier XML de structure du canal est envoyé au serveur, qui génèrera le code adapté.

5.2 Sauvegarde de la structure choisie

Le serveur reçoit la structure du canal. Il analyse le fichier et génère à partir de son contenu :

- Le fichier d'information du canal, qui contient son nom, son numéro de version, et la localisation de ses propriétés et de ses classes;
- Le fichier de propriétés du canal, qui regroupe son descriptif, sa langue, les caractéristiques de son accès Web, et indique s'il doit contenir un dictionnaire de mots-clés;
- Le fichier XML est sauvé sur le disque afin de pouvoir être réutilisé ultérieurement;
- L'interface de publication d'annotations sur le Web est créée, avec des zones libres permettant d'intégrer les données variables comme les termes du dictionnaire ou le contenu des champs;
- Trois classes sont générées et compilées dynamiquement : la classe de l'annotation, celle de la recommandation qui lui est fortement liée, et une *annotation factory* qui permet de créer plus rapidement des annotations et des recommandations (5.3). Le nom et le package de l'*annotation factory* sont précisés dans le fichier de propriétés du canal, cité ci-dessus.
- Un fichier *zip* compresse toutes ces données et les archive. Ce sera le fichier qui sera communiqué à l'assistant lorsqu'il mettra à jour l'interface du canal.

Le serveur démarre alors le canal.

5.3 Génération et compilation du code Java

Une fois le code compilé, Java permet de decharger l'ancienne version du canal et de charger dynamiquement en mémoire la nouvelle version. Par sécurité, à chaque fois que le code change, le nom du package est modifié. Ceci permet que la machine virtuelle ne charge pas depuis son cache l'ancienne version du code.

De plus, pour que la création d'une annotation soit plus rapide, une instance de la classe *AnnotationFactory* est obtenue par introspection, et c'est celle-ci qui crée les nouvelles instances des annotations et recommandations, sans nécessiter de nouvelles introspections, procédé coûteux pour un canal possédant 100 000 annotations.

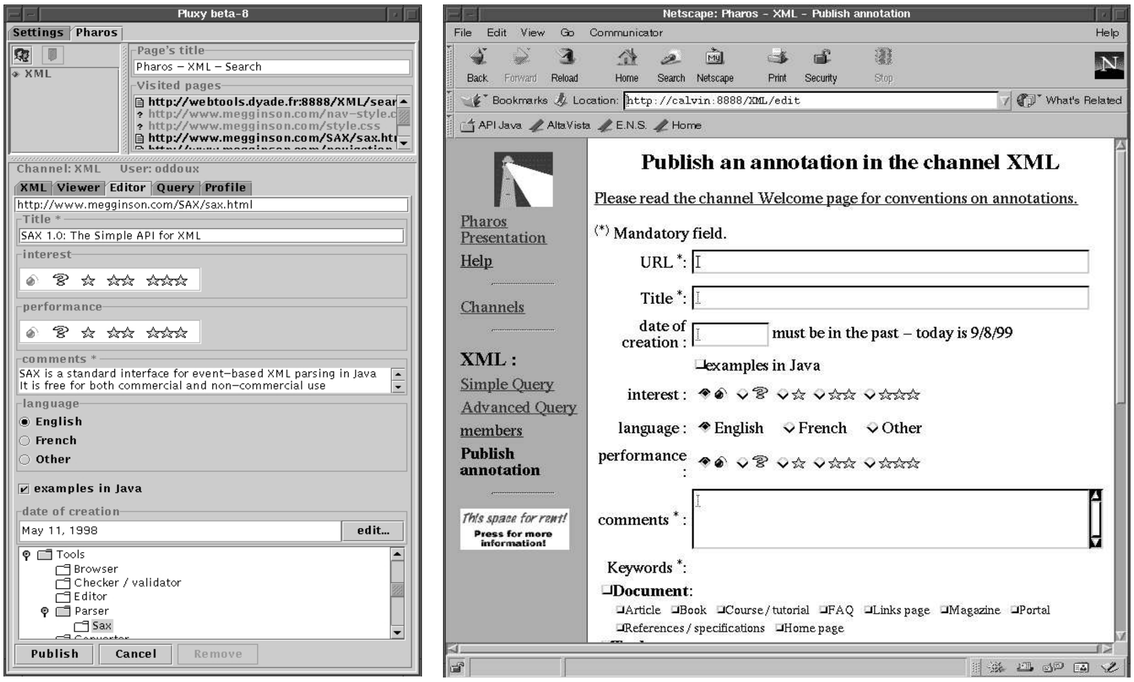

FIG. 7 – *Éditeurs d'annotations : celui de l'assistant à gauche, et celui de l'accès Web à droite*

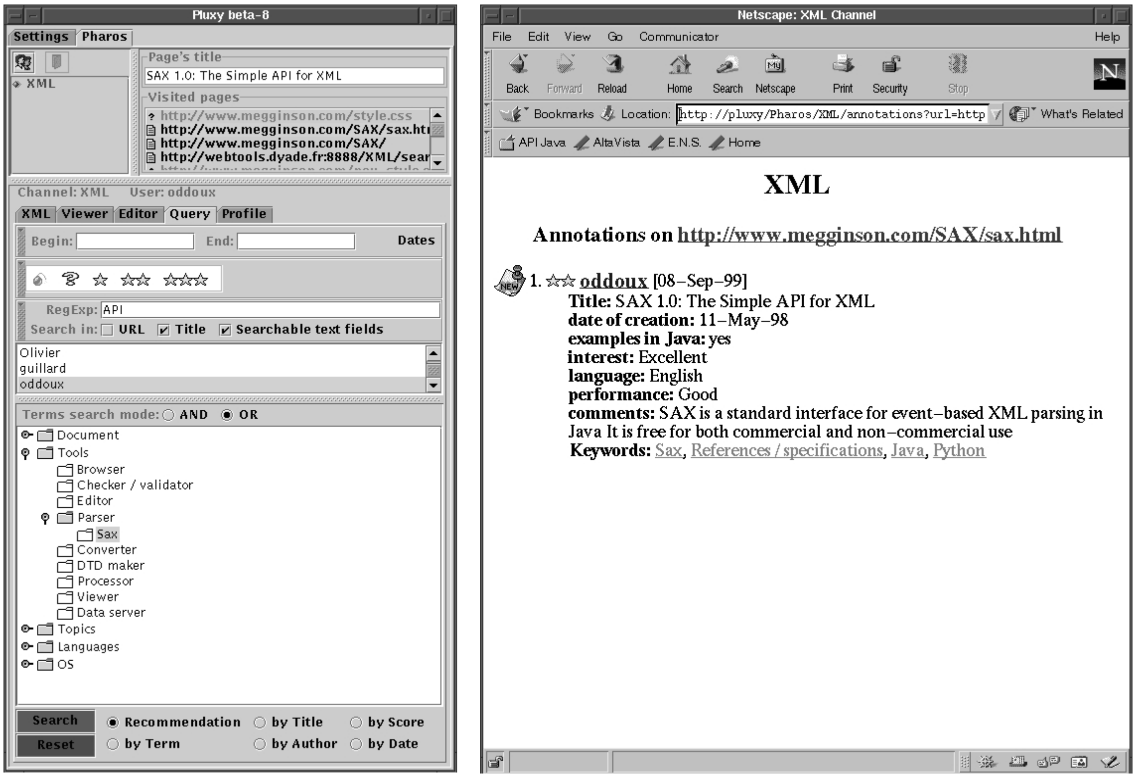

FIG. 8 – *Recherche avancée sous l'assistant, et affichage HTML complet d'une annotation*

5.4 Modification du code ayant un lien avec les annotations

5.4.1 Éditeur et visualiseur d'annotations

Ces deux interfaces étant très similaires, elles partagent le même code. Cette classe relit le fichier XML pour construire l'interface. Ensuite, les champs sont remplis par l'intermédiaire d'une table à chaque fois que l'utilisateur sélectionne une nouvelle URL (c'est en effet sous cette forme que les annotations sont transmises du serveur du canal à l'interface).

Dans l'éditeur, un bouton permet de valider l'annotation qui vient d'être rédigée. Si celle-ci est correcte (les champs obligatoires doivent être remplis, les dates doivent être passées ou futures s'il en a été décidé ainsi, et au moins un mot-clé doit avoir été sélectionné), elle est communiquée au serveur qui l'enregistre. Pour déterminer la forme des annotations *publiables*, l'administrateur aura fixé les attributs associés, comme l'aspect obligatoire d'un champ, sa valeur par défaut, la taille des zones de saisie, les dates autorisées, et les valeurs possibles d'une liste de choix (Fig. 7).

Le visualiseur, lui, possède les mêmes champs que l'éditeur mais leur contenu ne peut être changé. Il permet de consulter les annotations de tous les utilisateurs du canal sur une URL donnée.

5.4.2 Éditeur d'annotations sur le Web

L'interface d'édition d'annotations sur le Web est générée au moment de la fabrication du fichier *zip*. Elle est complétée au dernier moment par la valeur des champs, si l'utilisateur a souhaité se baser sur une annotation déjà présente, et par la liste des mots-clés du dictionnaire (Fig. 7).

5.4.3 Requêtes

Il existe deux types de requêtes :

- les requêtes simples : elles permettent de rechercher une expression régulière dans les champs texte, ou bien de demander les annotations associées à un mot-clé ou à un utilisateur. L'administrateur doit pouvoir décider quels champs sont utilisés dans la recherche de texte, c'est pourquoi cela fait partie des attributs du type *texte*.

- la recherche avancée : elle permet de préciser simultanément de nombreux détails : quand l'annotation a été publiée, la valeur de sa note moyenne, une expression régulière à chercher dans l'URL, dans le titre et/ou dans les autres champs de texte, et enfin un ou des mots-clés qui doivent être inclus.

L'interface de recherche avancée de l'assistant est similaire à celle du Web, avec la possibilité de préciser l'utilisateur qui a posé l'annotation (Fig. 8).

5.4.4 Affichage du résultat des requêtes

Qu'elles soient effectuées depuis l'assistant ou depuis le Web, le résultat des requêtes est affiché dans le navigateur de l'utilisateur. Un fichier HTML est donc généré, affichant les annotations suivant des critéres déterminés par la requête.

On dispose de deux formats d'affichage:

- l'affichage résumé affiche le résultat des requêtes sous la forme choisie par l'administrateur : seuls certains champs sont affichés, et les champs texte trop longs sont tronqués. Pour que l'administrateur puisse décider de l'aspect de l'affichage, des attributs de format d'affichage ont été ajoutés à tous les types de champs.

- l'affichage complet affiche la valeur de tous es champs remplis. Il est utilisé quand l'utilisateur demande à voir le détail d'une annotation tronquée (Fig. 8).

6 Extension possible

Afin de faciliter la création de nouveaux types de champs, un changement de structure pourrait être effectué. En effet, il serait judicieux de créer une interface implémentée par autant de classes qu'il y aurait de types de champs, et possédant toutes les méthodes nécessaires à la construction des interfaces graphiques, à la génération de code Java, et à l'écriture et la lecture XML.

Pour ajouter un type de champ, il suffirait alors de définir une nouvelle classe en se basant sur les modèles existants. Si cette idée n'a pas été intégrée, faute de temps, elle mérite d'être considérée. Un ajout possible serait des champs textuels typés, comme des entrées BibTeX ou des URL, ce qui permettrait de vérifier la validité de leur contenu au moment de la publication, et d'afficher par exemple une URL sous forme d'un lien dynamique...

7 Conclusion

Ce stage constitue l'apport d'une fonctionnalité importante pour le projet *Pharos*. Il augmente fortement la capacité d'adaptation d'un canal à son thème. Il sera intégré au projet dès la prochaine version.

Il m'a d'autre part permis de mieux me familiariser avec le langage Java, et de savoir manipuler et modifier un code déjà existant et dont il faut comprendre tout d'abord la structure et le fonctionnement.

Références

[BD99] Vincent Bouthors and Olivier Dedieu. Pharos, a collaborative infrastructure for web knowledge sharing. Rapport de Recherche RR-3679, INRIA, May 1999. `http://www-sor.inria.fr/publi/PCIWKS_rr3679.html`.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. W3C recommendation, W3C, feb 1998. `http://www.w3.org/TR/1998/REC-xml-19980210.html`.

[Ded98] Olivier Dedieu. Pluxy : un proxy Web dynamiquement extensible. In *Proceedings of the 1998 NoTeRe colloquium*, October 1998. `http://www-sor.inria.fr/publi/PPWDE_notere98.html`.

[WC98] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, Reading, MA, USA, 1998. `http://www2.awl.com/cseng/javaseries/swingtut.html`.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In *Proceeding of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)*, Toronto, June 1996. USENIX. `http://www.usenix.org/publications/library/proceedings/coots96/wollrath.html`.

Partie 3

D.E.A. d'Algorithmique

Rapport du stage de D.E.A.
sous la direction de P. Gastin

avril à juin 2000

Mémoire de DEA

Génération efficace d'automates de Büchi à partir de formules LTL

Denis Oddoux

Sous la direction de P. Gastin

avril à juin 2000

Résumé

Dans cet article nous proposons un nouvel algorithme pour générer un automate de Büchi reconnaissant exactement l'ensemble des mots qui satisfont une formule LTL. Cet algorithme, basé sur les automates alternants, constitue un apport important au model-checking LTL, car il est rapide, utilise peu de mémoire, et produits des automates de petite taille. Il est comparé aux travaux existants sur des formule LTL usuelles dans le cadre du model-checking, ainsi que sur des formules générées aléatoirement.

$$GFp \to G(q \to Fr)$$

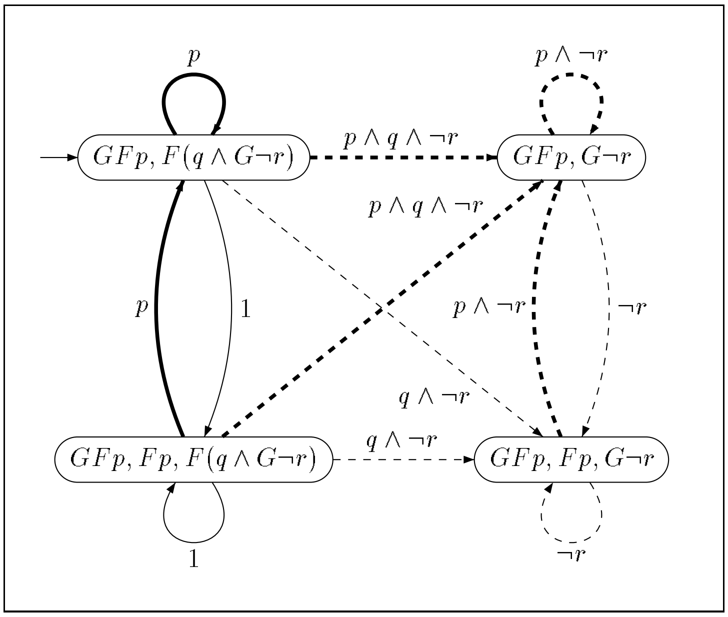

Table des matières

Introduction

Le model-checking LTL est un procédé permettant de prouver automatiquement la correction d'un programme. A partir d'une modélisation du programme sous la forme d'un automate et d'une formule de logique temporelle linéaire exprimant les contraintes qu'il doit satisfaire, un model-checker répond soit que le programme satisfait la contrainte, soit qu'il ne la satisfait pas et il donne dans ce cas une exécution du programme qui ne satisfait pas la formule.

Les formules LTL utilisées dans ce cadre ont souvent la structure suivante : sous réserve de plusieurs conditions d'équité (le programme passe infiniment souvent par un certain nombre d'états), une contrainte est vérifiée (c'est ici par exemple une condition de réponse : à chaque fois que le programme passe par un état donné, il passe ensuite nécessairement un peu plus tard dans un autre état donné). Ainsi une formule de forme usuelle pourrait être

$$(GFp_1 \wedge \ldots \wedge GFp_n) \to G(q \to Fr)$$

L'outil de model-checking fonctionne ainsi : il génère tout d'abord un automate de Büchi reconnaissant exactement les mots infinis qui satisfont la négation de la formule, puis il effectue le produit de cet automate avec l'automate qui représente le programme à tester. Il teste ensuite si le langage du produit est vide : s'il l'est effectivement, alors la formule est satisfaite. Sinon, un mot accepté par l'automate produit donne un contre-exemple.

L'outil le plus connu pour le model-checking LTL est Spin ([Hol97]). Toutefois, on remarque en l'utilisant que même dans la version la plus récente (version 3.3.10) le module qui génère un automate de Büchi pour la négation de la formule se trouve être extrèmement lent sur des formules usuelles, par exemple sur les formules :

$$\theta_n = \neg((GFp1 \wedge \ldots \wedge GFp_n) \to G(q \to Fr))$$

L'algorithme utilisé par Spin est exposé dans [GPVW95]. Il a ensuite été raffiné à plusieurs reprises (voir [DGV99] et [SB00] par exemple), mais sans changement du principe de base de l'algorithme. Notre article propose un procédé différent pour générer un automate de Büchi à partir d'une formule LTL, basé sur l'utilisation des automates alternants. Le programme qui l'implémente, « Alternating-LTL », obtient souvent des automates plus petits que ceux trouvés par Spin, mais surtout prend un temps de calcul et une place en mémoire dérisoires devant ceux que prend Spin sur des formules LTL usuelles (tableau 1).

	Alternating-LTL		Spin	
θ_1	< 0.01 sec	9 ko	0.3 sec	460 ko
θ_2	< 0.01 sec	11 ko	8 sec	4,2 Mo
θ_3	< 0.01 sec	19 ko	4 min	52 Mo
θ_4	0.08 sec	38 ko	5 h 30	970 Mo
θ_5	0.6 sec	48 ko		
θ_6	6.7 sec	88 ko		
θ_7	54 sec	176 ko		
θ_8	10 min	248 ko		
θ_9	1 h 30	490 ko		

TAB. 1 – *Comparaison en temps et en espace sur les formules θ_n pour $1 \leq n \leq 9$*

L'idée de générer un automate alternant à partir d'une formule LTL n'est pas nouvelle (voir [Var96]). En effet l'avantage de ce procédé est que l'automate alternant obtenu possède un nombre d'états n inférieur ou égal à la taille de la formule. Le problème est qu'il est nécessaire de le transformer ensuite en un automate de Büchi, et que l'automate ainsi créé par l'agorithme classique possède $O(2^{2n})$ états. Ce que propose notre article, c'est une méthode permettant de transformer un automate alternant très faible avec n états (l'automate alternant issu d'une formule LTL se trouve être très faible) en un automate de Büchi possédant au maximum $(n+1) \times 2^n$ états. Cette méthode est d'ailleurs parfaitement indépendante du cadre du model-checking LTL, et peut donc être utilisée dans d'autres contextes.

Ainsi cet article propose un algorithme permettant d'obtenir un automate alternant très faible reconnaissant les mots qui satisfont une formule LTL donnée, puis créant à partir de cet automate un automate de Büchi de même langage. Il passe par une étape intermédiaire, en construisant un automate de Büchi généralisé avec acceptation sur les transitions.

Afin d'être le plus efficace possible, Alternating-LTL remet en forme la formule LTL avant tout traitement, en utilisant pour cela le module de Spin 3.3.10 qui gère cette opération, lui-même inspiré par [SB00] et [EH00]. Ensuite, chaque automate est simplifié suivant des règles proposées dans ces deux mêmes articles.

Un autre point fort de notre implémentation est la simplification des automates à la volée : celle-ci s'opère au fur et à mesure de la construction, et permet de gagner très efficacement du temps et de la mémoire. En effet, les automates créés sont en général très gros, mais sont fortements réduits en taille après simplification. Simplifier à la volée permet d'occuper un espace mémoire de l'ordre de la taille de l'automate simplifié, au lieu d'un espace égal à la taille de l'automate généré avant la simplification, et de gagner beaucoup de temps car la simplification réduit sensiblement le nombre d'états accessibles donc par là-même le nombre de transitions à calculer.

Dans les sections 1 à 7, nous exposons, à la suite des définitions nécessaires à leur compréhension, les trois parties de notre algorithme ainsi que la preuve de leur correction. Lors de la définition de chacun des types d'automates, nous introduisons des méthodes de simplification pouvant être utilisées à la volée. Les algorithmes de simplification que nous employons sont exposés dans la section 8. Dans la section 9, nous proposons l'étude de notre algorithme sur un exemple, celui de la formule θ_1. Enfin, dans la section 10, nous comparons notre implémentation avec le logiciel Spin 3.3.10, en se basant aussi bien sur des formules LTL usuelles dans le cadre du model-checking LTL, que sur des formules générées aléatoirement.

1 Logique temporelle linéaire

1.1 Grammaire

Les formules de logique temporelle linéaire (formules LTL) sur l'ensemble fini de propriétés $Prop$ sont définies par la grammaire suivante :

$$\varphi ::= \mathrm{tt} \mid \mathrm{ff} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\ \varphi \mid \varphi\, \mathcal{U}\ \varphi \mid \varphi\ \mathcal{V}\ \varphi$$

où les p sont des éléments de $Prop$

On définit $F\varphi = \mathrm{tt}\, \mathcal{U}\ \varphi$; $G\varphi = \neg F \neg\varphi$; $\varphi \rightarrow \psi = \neg\varphi \vee \psi$; $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$

1.2 Sémantique

Considérons un mot $u = u_0 u_1 \dots u_n \dots$ sur l'alphabet Σ^ω, où $\Sigma = 2^{Prop}$. On définit par induction la relation $u \models \varphi$ (u satisfait φ) par :

- $\forall u \in \Sigma^\omega, u \models \mathrm{tt}$ et $u \not\models \mathrm{ff}$
- $u \models p$ si $p \in u_0$
- $u \models \neg\varphi_1$ si $u \not\models \varphi_1$
- $u \models \varphi_1 \vee \varphi_2$ si $u \models \varphi_1$ ou $u \models \varphi_2$
- $u \models \varphi_1 \wedge \varphi_2$ si $u \models \varphi_1$ et $u \models \varphi_2$
- $u \models \mathcal{X}\ \varphi_1$ si $u_1 \dots u_n \dots \models \varphi_1$
- $u \models \varphi_1\, \mathcal{U}\ \varphi_2$ si $\exists k \geq 0\ \forall 0 \leq i < k\ u_i \dots u_n \dots \models \varphi_1$ et $u_k \dots u_n \dots \models \varphi_2$
- $u \models \varphi_1\ \mathcal{V}\ \varphi_2$ si ($\exists k \geq 0\ \forall 0 \leq i \leq k\ u_i \dots u_n \dots \models \varphi_2$ et $u_k \dots u_n \dots \models \varphi_1$) ou ($\forall i \geq 0\ u_i \dots u_n \dots \models \varphi_2$)

1.3 Forme normale négative

Par la suite, nous supposerons que les formules LTL sont uniquement issues de la grammaire ci-dessous (formules en forme normale négative) :

$$\varphi ::= \mathrm{tt} \mid \mathrm{ff} \mid p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\ \varphi \mid \varphi\, \mathcal{U}\ \varphi \mid \varphi\ \mathcal{V}\ \varphi$$

Ainsi, les négations seront présentes uniquement au niveau des prédicats, ce qui simplifiera le traitement des formules. Pour mettre une formule en forme normale négative, il suffit d'appliquer récursivement les règles suivantes :

$$\neg\mathrm{tt} = \mathrm{ff};\ \ \neg\mathrm{ff} = \mathrm{tt};\ \ \neg(\neg\varphi_1) = \varphi_1;\ \ \neg(\varphi_1 \vee \varphi_2) = (\neg\varphi_1) \wedge (\neg\varphi_2);\ \ \neg(\varphi_1 \wedge \varphi_2) = (\neg\varphi_1) \vee (\neg\varphi_2)$$
$$\neg(\mathcal{X}\ \varphi_1) = \mathcal{X}\ (\neg\varphi_1);\ \ \neg(\varphi_1\, \mathcal{U}\ \varphi_2) = (\neg\varphi_1)\ \mathcal{V}\ (\neg\varphi_2);\ \ \neg(\varphi_1\ \mathcal{V}\ \varphi_2) = (\neg\varphi_1)\, \mathcal{U}\ (\neg\varphi_2);$$

Si l'on considère que les éléments de la forme '$\neg p$' sont des élément unitaires de la formule, alors cette transformation n'augmente pas la taille de la formule.

Nous appelerons formule *temporelle* toute formule LTL issue de cette grammaire qui n'est pas de la forme $\varphi_1 \vee \varphi_2$ ou $\varphi_1 \wedge \varphi_2$.

2 Automates alternants

2.1 Automate alternant classique

Un automate alternant est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ où

- Q est l'ensemble des états
- Σ est l'alphabet
- $\delta : Q \times 2^\Sigma \to 2^{2^Q}$ est la fonction de transition
- $I \in 2^{2^Q}$ indique les ensembles initiaux d'états
- $F \subseteq Q$ est l'ensemble des états finaux

Une exécution d'un automate alternant sur le mot $u_0 u_1 \ldots u_n \ldots$ est une forêt σ étiquetée dans Q vérifiant les propriétés suivantes :

- L'ensemble $i(\sigma)$ des étiquettes des racines est un élément de I.
  Si $I = \{\emptyset\}$, σ est la forêt vide. Par contre on ne peut pas avoir $I = \emptyset$ dans une forêt valide.
- Pour chaque nœud de σ d'étiquette q et de profondeur i, il existe $\alpha \in 2^\Sigma$ tel que $u_i \in \alpha$ et l'ensemble des étiquettes des fils de q est un élément de $\delta(q, \alpha)$.
  Si $\delta(q, \alpha) = \{\emptyset\}$, q est une feuille de σ. Dans une forêt valide $\delta(q, \alpha) \neq \emptyset$.

Une exécution σ est dite acceptante au sens de Büchi si toute branche infinie possède une infinité de nœuds étiquetés dans F. Elle est dite acceptante au sens de co-Büchi si toute branche infinie possède un nombre fini de nœuds étiquetés dans F. On supposera qu'une des deux conventions est choisie lors de la définition de l'automate, afin de pouvoir par la suite ommettre la précision Büchi ou co-Büchi.

Le mot $u_0 u_1 \ldots u_n \ldots$ est dit accepté par l'automate si il existe une exécution acceptante de cet automate sur ce mot. L'ensemble des mots acceptés par $\mathcal{A}$ est appelé langage de $\mathcal{A}$ et noté $\mathcal{L}(\mathcal{A})$. Par extension, on notera $\mathcal{L}(\mathcal{A}, J)$ ou plus simplement $\mathcal{L}(J)$ l'ensemble des mots acceptés par une exécution σ de $\mathcal{A}$ vérifiant $i(\sigma) \in J$.

2.2 Représentation en mémoire

Le lecteur a peut-être remarqué que la description d'un automate alternant est un peu différente de celle qu'il connait. Ces différences traduisent une volonté d'optimisation permettant de réduire l'espace occupé en mémoire.

Ainsi, les étiquettes des transitions ne sont plus des éléments mais des parties de Σ. Ceci permet de regrouper les transitions : au lieu d'avoir $\forall a \in \Sigma$ si $p \in a$ alors $\delta(p, a) = \{\emptyset\}$ ($2^{|\Sigma|-1}$ transitions), on a une unique transition $\delta(p, \Sigma_p) = \{\emptyset\}$. De plus, on vérifie que les ensembles α utilisés sont uniquement engendrés par Σ, Σ_p, $\overline{\Sigma_p}$ et par l'intersection de tels ensembles. On peut donc représenter α par une paire de $Prop \times Prop$, où (P_1, P_2) représente $\bigcap\limits_{p \in P_1} \Sigma_p \cap \bigcap\limits_{p \in P_2} \overline{\Sigma_p}$. Si on obtient lors d'un calcul un ensemble $\alpha = (P_1, P_2)$ tel que $P_1 \cap P_2 \neq \emptyset$ alors $\alpha = \emptyset$ et toute transition étiquetée par cet ensemble peut être supprimée. Les ensembles P_1 et P_2 sont représentés par des tableaux de booléens, et on a la propriété suivante : $(P_1, P_2) \cap (P_1', P_2') = (P_1 \cup P_1', P_2 \cup P_2')$.

D'autre part, en général les destinations des transitions d'un automate alternant sont exprimées comme une formule logique sur les états de l'automate, et ainsi toutes les relations de la forme $e \in \delta(\varphi, \alpha)$ sont remplacées par $e \models \delta(\varphi, \alpha)$. Avec cette repésentation, par exemple, de l'état $\{\mathcal{X}\ p\}$ de l'automate de Büchi généralisé partent $2^{|Q|-1}$ transitions, vers chacune des parties de Q qui contiennent p. Notre algorithme n'en crée qu'une seule, vers l'état $\{p\}$.

2.3 Automate alternant faible

Un automate alternant $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ est dit faible si il existe une partition de Q sous la forme $\bigcup_{\lambda \in \Lambda} Q_\lambda$ (si $q \in Q_\lambda$, on dit que λ est l'ordre de q : $\omega(q) = \lambda$) et un ordre partiel $\preccurlyeq$ sur Λ tels que :

- Tous les successeurs d'un état donné ont un ordre inférieur à l'ordre de cet état
  $\forall q \in Q \ \forall \alpha \in 2^\Sigma \ \forall e \in \delta(q, \alpha) \ \forall q' \in e \ \omega(q') \preccurlyeq \omega(q)$
- Deux états de même ordre sont tous deux finaux ou non finaux
  $\exists \Phi \subseteq \Lambda$ tel que $F = \bigcup_{\mu \in \Phi} Q_\mu$

Lemme 2.1 *Dans toute exécution d'un automate alternant faible, la suite des ordres des étiquettes d'une branche infinie est stationnaire.*

Démonstration Dans un automate alternant faible, tous les successeurs d'un état donné ont un ordre inférieur à l'ordre de cet état donc sur une branche infinie d'une exécution de cet automate, la suite des ordres des étiquettes est décroissante. Or Λ est fini donc cette suite est nécessairement stationnnaire (c'est-à-dire constante à partir d'un certain rang). □

On retrouve ainsi une propriété classique des automates alternants faibles :

Proposition 2.2 *En remplaçant F par $Q \backslash F$ on transforme un automate alternant faible de Büchi en un automate alternant faible co-Büchi de même langage et réciproquement.*

Démonstration Considérons une branche infinie d'une exécution acceptante d'un automate alternant faible de Büchi : elle contient une infinité de nœuds étiquetés dans F, et d'après le lemme 2.1, la suite des ordres de ses étiquettes est stationnaire. Ainsi elle ne contient qu'un nombre fini de nœuds étiquetés dans $Q \backslash F$ et est donc une exécution acceptante au sens de co-Büchi du même automate où F a été remplacé par $Q \backslash F$, sur le même mot. La réciproque est similaire. □

2.4 Automate alternant très faible

Un automate alternant $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ est dit très faible si il existe un ordre partiel $\preccurlyeq$ sur Q tel que $\forall q \in Q \ \forall \alpha \in 2^\Sigma \ \delta(q, \alpha) \in 2^{2^{\{q' \in Q \ | \ q' \preccurlyeq q\}}}$

C'est en fait un automate alternant faible avec $\forall \lambda \in \Lambda \ \mathrm{card}(Q_\lambda) = 1$

Lemme 2.3 *Dans toute exécution d'un automate alternant très faible, la suite des étiquettes d'une branche infinie est stationnaire.*

Démonstration C'est le lemme 2.1 appliqué au cas d'un automate alternant très faible. □

Proposition 2.4 *Soit σ une exécution d'un automate alternant très faible co-Büchi. Si $\forall f \in F$, σ ne possède pas de branche infinie dont les nœuds ont ultimement pour étiquette f, alors σ est acceptante.*

Démonstration D'après le lemme 2.3, la suite des étiquettes d'une branche infinie de σ est stationnaire, et sa valeur limite n'est pas dans F. Ainsi sa valeur limite est dans $Q \backslash F$ et donc cette branche posséde une infinité de nœuds étiquetés dans $Q \backslash F$. Il en résulte que σ est acceptante. □

2.5 Q-DAG

Nous appellerons run-dag sur Q ou Q-DAG un graphe acyclique dirigé $\sigma' = (V, E, \lambda)$, pour lequel il existe $n \in \mathbb{N} \cup \{\infty\}$ et une partition de V en $\bigcup_{i=0}^{n} V_i$ telle que $E \subseteq \bigcup_{i=1}^{n} V_{i-1} \times V_i$.
$\lambda : V \to Q$ est une fonction d'étiquetage. On note $\sigma'[j]$ la réduction de σ' à $\bigcup_{i=0}^{j} V_i$.

Proposition 2.5 *Un mot $u = u_0 u_1 \ldots u_n \ldots$ est accepté par un automate alternant $\mathcal{A}$ si et seulement si il existe un Q-DAG $\sigma' = (\bigcup_{i=0}^{n} V_i, E, \lambda)$ tel que :*

- $\{\lambda(v) \mid v \in V_0\} \in I$
- $\forall 0 \leq i \leq n \begin{cases} \lambda \textit{ est injective sur } V_i \\ \forall v \in V_i \; \exists \alpha \in 2^{\Sigma} \; u_i \in \alpha \textit{ et } \{\lambda(v') \mid (v, v') \in E\} \in \delta(\lambda(v), \alpha) \end{cases}$
- *toute branche infinie possède une infinité de nœuds étiquetés dans F si $\mathcal{A}$ est Büchi ;*
  toute branche infinie possède un nombre fini de nœuds étiquetés dans F si $\mathcal{A}$ est co-Büchi.

Démonstration La preuve du deuxième cas étant similaire à celle du premier, on supposera par la suite que $\mathcal{A}$ est Büchi.

Pour chaque nœud N de profondeur i d'une forêt σ, on définit la mesure du nœud, $\mu(N)$, comme le plus petit entier j vérifiant la propriété suivante : sur toute branche infinie de σ passant par N, il existe un descendant de N étiqueté dans F de profondeur inférieure à $i + j$. On pose $\mu(N) = \infty$ si il existe une branche infinie de σ passant par N telle qu'aucun descendant de N n'est étiqueté dans F. Ce cas ne peut pas se produire dans le cas d'une exécution acceptante. $\mu(N)$ est la plus petite profondeur à partir de laquelle on a vu un nœud étiqueté dans F sur toutes les branches infinies du sous-arbre issu de N.

Soit σ_0 une exécution acceptante de $\mathcal{A}$ sur u, et soit $i \geq 0$. Construisons une forêt σ_{i+1} à partir de σ_i de la façon suivante : $\forall q \in Q$, on choisit parmi les nœuds d'étiquette q et de profondeur i un nœud N_0 de mesure minimale, et on remplace le sous-arbre issu de chaque nœud d'étiquette q et de profondeur i par une copie du sous-arbre issu de N_0. On note μ_i la mesure sur σ_i.

On montre aisément par récurrence que $\forall i \geq 0$ σ_i est une exécution acceptante de $\mathcal{A}$ sur u, telle que $\sigma_i[i] = \sigma_{i+1}[i]$, les sous-arbres issus de deux nœuds de σ_i de profondeur inférieure à i et de même étiquette sont identiques. De plus, tout nœud N de σ_i de profondeur inférieure à i vérifie $\mu_{i+1}(N) \leq \mu_i(N)$.

Ainsi la suite $(\sigma_i)_{i \geq 0}$ converge vers une valeur limite $\bar{\sigma}$, exécution de $\mathcal{A}$ sur u telle que si deux nœuds de même profondeur ont même étiquette, alors les sous-arbres issus de ces nœuds sont identiques. De plus $\forall i \geq 0$ $\bar{\sigma}[i] = \sigma_i[i]$. Soit N un nœud de $\bar{\sigma}$, et soit $p = n + \mu_n(N)$. On a $\mu_p(N) \leq \mu_n(N)$ donc $n + \mu_p(N) \leq p$: dans σ_p, sur toute branche infinie passant par N il existe un descendant de N de profondeur inférieure à p étiqueté dans F. $\bar{\sigma}[p] = \sigma_p[p]$ donc cette propriété reste nécessairement vraie dans $\bar{\sigma}$. Ainsi pour tout nœud N de $\bar{\sigma}$, $\bar{\mu}(N) = \mu_{n+\mu_n(N)}(N) < \infty$, et donc $\bar{\sigma}$ est acceptante.

Il suffit maintenant de transformer $\bar{\sigma}$ en un Q-DAG σ', en fusionnant successivement les nœuds de même profondeur et de même étiquette. On vérifie aisément que σ' vérifie les propriétés annoncées.

Réciproquement si σ' vefifie les propriétés ci-dessus, alors on peut éclater σ' pour obtenir une forêt σ, en créant successivement, pour chaque nœud d'incidence k, k nœuds d'incidence 1 et de même sous-DAG. On montre aisément que σ est une exécution acceptante de $\mathcal{A}$ sur u : u est accepté par $\mathcal{A}$. □

2.6 Propriétés du langage d'un automate alternant

On remarquera que $\mathcal{L}(\emptyset) = \emptyset$ alors que $\mathcal{L}(\{\emptyset\}) = \Sigma^\omega$.

Proposition 2.6 $\mathcal{L}(J_1 \cup J_2) = \mathcal{L}(J_1) \cup \mathcal{L}(J_2)$

Démonstration Soit $u \in \mathcal{L}(J_1 \cup J_2)$: il existe une exécution acceptante σ de $\mathcal{A}$ sur u avec $i(\sigma) \in J_1 \cup J_2$. Par symétrie, on peut supposer que $i(\sigma) \in J_1$ et alors $u \in \mathcal{L}(J_1)$.

Réciproquement, si $u \in \mathcal{L}(J_1) \cup \mathcal{L}(J_2)$, par symétrie on peut supposer que $u \in \mathcal{L}(J_1)$: il existe une exécution acceptante σ de $\mathcal{A}$ sur u avec $i(\sigma) \in J_1$; $i(\sigma) \in J_1 \cup J_2$ donc on a $u \in \mathcal{L}(J_1 \cup J_2)$. $\square$

Proposition 2.7 *On note* $J_1 \otimes J_2 = \{e_1 \cup e_2 \mid e_1 \in J_1 \text{ et } e_2 \in J_2\}$. $\mathcal{L}(J_1 \otimes J_2) = \mathcal{L}(J_1) \cap \mathcal{L}(J_2)$

Démonstration Soit $u \in \mathcal{L}(J_1 \otimes J_2)$: il existe une exécution acceptante σ de $\mathcal{A}$ sur u avec $i(\sigma) \in J_1 \otimes J_2$. Ainsi $i(\sigma) = e_1 \cup e_2$ avec $e_1 \in J_1$ et $e_2 \in J_2$. Considérons la sous-forêt de σ obtenue en ne conservant que les arbres dont les racines sont étiquetées dans e_1. C'est une exécution acceptante σ_1 de $\mathcal{A}$ sur u avec $i(\sigma_1) = e_1 \in J_1$, et ainsi on a $u \in \mathcal{L}(J_1)$. De même, $u \in \mathcal{L}(J_2)$. Ainsi, $u \in \mathcal{L}(J_1) \cap \mathcal{L}(J_2)$

Réciproquement, si $u \in \mathcal{L}(J_1) \cap \mathcal{L}(J_2)$, alors il existe deux exécutions acceptantes σ_1, σ_2 de $\mathcal{A}$ sur u avec respectivement $i(\sigma_1) \in J_1$ et $i(\sigma_2) \in J_2$. La réunion de ces deux forêts est une exécution acceptante σ de $\mathcal{A}$ sur u, avec $i(\sigma) = i(\sigma_1) \cup i(\sigma_2) \in J_1 \otimes J_2$. Ainsi $u \in \mathcal{L}(J_1 \otimes J_2)$. $\square$

2.7 Simplification d'un automate alternant

Théorème 2.8 (transitions) *Si* $\exists q \in Q\ \exists \alpha_1, \alpha_2 \in 2^\Sigma\ \exists e_1 \in \delta(q, \alpha_1), e_2 \in \delta(q, \alpha_2)$ *tels que* $\alpha_1 \subseteq \alpha_2$ *et* $e_2 \subseteq e_1$ *mais* $\alpha_1 \neq \alpha_2$ *ou* $e_2 \neq e_1$*, alors on peut supprimer* e_1 *de* $\delta(q, \alpha_1)$*.*

Démonstration Soit σ une exécution acceptante de $\mathcal{A}$ sur un mot $u_0 \dots u_n \dots$ contenant un nœud N d'étiquette q et de profondeur i, tel que $u_i \in \alpha_1$ et l'ensemble des étiquettes des fils de N est e_1. En supprimant de σ les sous-arbres issus des fils de N dont l'étiquette n'est pas dans e_2, et en répétant cette opération pour tous les nœuds de ce type, l'exécution reste valide et acceptante, car $u_i \in \alpha_2$ et $e_2 \in \delta(q, \alpha_2)$, et n'utilise plus la transition (q, α_1, e_1).

Réciproquement, toute exécution acceptante de l'automate modifié sur un mot u est aussi une exécution acceptante de $\mathcal{A}$ sur u.

Cette suppression ne modifie donc pas le langage reconnu par l'automate $\mathcal{A}$. $\square$

Théorème 2.9 (états) *Si* $\exists q_1, q_2 \in Q\ q_1 \neq q_2$ *tels que* $\forall \alpha \in 2^\Sigma\ \delta(q_1, \alpha) = \delta(q_2, \alpha)$ *et* $q_1, q_2 \in F$ *ou* $q_1, q_2 \notin F$ *alors on peut supprimer de* Q *l'état* q_1*, en remplaçant chaque occurence de* q_1 *dans* I *et dans* δ *par* q_2*.*

Démonstration Soit σ une exécution acceptante de $\mathcal{A}$ sur un mot $u_0 \dots u_n \dots$ Si on remplace dans σ toutes les étiquettes q_1 par q_2, on obtient clairement une exécution acceptante de l'automate modifié $\mathcal{A}'$ sur le même mot.

Réciproquement, soit σ une exécution de $\mathcal{A}'$ sur un mot $u_0 \dots u_n \dots$ Soit N un nœud d'étiquette q et de profondeur i. $\exists \alpha \in 2^\Sigma\ u_i \in \alpha$ et l'ensemble des étiquettes des fils de N est dans $\delta'(q, \alpha)$. Par construction, en étiquetant par q_1 certains fils de N étiquetés par q_2, l'ensemble des étiquettes des fils de N est dans $\delta(q, \alpha)$. En réétiquetant de même les racines, on reconstruit une exécution acceptante de $\mathcal{A}$ sur u.

Cette suppression ne modifie donc pas le langage reconnu par l'automate $\mathcal{A}$. $\square$

3 Passage d'une formule LTL à un automate alternant très faible

Le but de cette section est de construire à partir d'une formule LTL φ sur l'ensemble $Prop$ un automate alternant très faible qui reconnaît exactement les mots qui satisfont φ.

3.1 Construction de l'automate alternant très faible

On définit préalablement la fonction $\widetilde{\ }$ sur les formules LTL par :

$$\begin{cases} \widetilde{\psi_1} = & \{\{\psi_1\}\} \text{ si } \psi_1 \text{ est une formule temporelle} \\ \widetilde{\psi_1 \vee \psi_2} = & \widetilde{\psi_1} \ \cup \ \widetilde{\psi_2} \\ \widetilde{\psi_1 \wedge \psi_2} = & \widetilde{\psi_1} \ \otimes \ \widetilde{\psi_2} \end{cases}$$

Soit φ une formule LTL. L'automate alternant très faible co-Büchi $\mathcal{A}_\varphi$ est défini par :

- Q est l'ensemble des sous-formules temporelles de φ
- $\Sigma = 2^{Prop}$
- $I = \widetilde{\varphi}$
- F est l'ensemble des sous-formules de φ de la forme $\psi_1 \ \mathcal{U} \ \psi_2$
- δ et $\bar{\delta}$ sont définies comme suit (α prend ses valeurs dans 2^Σ) :

$$\begin{cases} \delta(\mathrm{tt}, \Sigma) = & \{\emptyset\} \\ \delta(p, \Sigma_p) = & \{\emptyset\} \text{ où } \Sigma_p = \{\alpha \in 2^\Sigma \mid p \in \alpha\} \\ \delta(\neg p, \overline{\Sigma_p}) = & \{\emptyset\} \text{ où } \overline{\Sigma_p} = 2^\Sigma \backslash \Sigma_p \\ \delta(\mathcal{X} \ \psi_1, \Sigma) = & \widetilde{\psi_1} \\ \delta(\psi_1 \ \mathcal{U} \ \psi_2, \alpha) = & (\bar{\delta}(\psi_1, \alpha) \otimes \{\{\psi_1 \ \mathcal{U} \ \psi_2\}\}) \ \cup \ \bar{\delta}(\psi_2, \alpha) \\ \delta(\psi_1 \ \mathcal{V} \ \psi_2, \alpha) = & (\bar{\delta}(\psi_2, \alpha) \otimes \{\{\psi_1 \ \mathcal{V} \ \psi_2\}\}) \ \cup \bigcup_{\alpha_1 \cap \alpha_2 = \alpha} \bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2) \end{cases}$$

$$\begin{cases} \bar{\delta}(\psi_1, \alpha) = & \delta(\psi_1, \alpha) \text{ si } \psi_1 \text{ est une formule temporelle} \\ \bar{\delta}(\psi_1 \vee \psi_2, \alpha) = & \bar{\delta}(\psi_1, \alpha) \ \cup \ \bar{\delta}(\psi_2, \alpha) \\ \bar{\delta}(\psi_1 \wedge \psi_2, \alpha) = & \bigcup_{\alpha_1 \cap \alpha_2 = \alpha} \bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2) \end{cases}$$

N.B. : On considère que partout où δ n'a pas été définie ci-dessus, sa valeur est en fait $\emptyset$. On remarquera que δ, fonction de transition de l'automate construit, n'est définie que sur Q (les sous-formules temporelles de φ), et qu'elle est étendue aux autres sous-formules de φ par $\bar{\delta}$, ce qui permet une écriture plus légère de la définition.

En utilisant la relation ($\psi_1 \preccurlyeq \psi_2$ si ψ_1 est une sous-formule de ψ_2), on montre aisément par induction sur les sous-formules de φ que $\mathcal{A}_\varphi$ est un automate alternant très faible .

3.2 Démonstration de l'équivalence

Lemme 3.1 $\forall \psi$ *sous-formule de* $\varphi, \mathcal{L}(\widetilde{\psi}) = \bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\bar{\delta}(\psi, \alpha))$

Démonstration par induction sur ψ

- Si ψ est une formule temporelle, on a $\widetilde{\psi} = \{\{\psi\}\}$ et $\bar{\delta}(\psi, \alpha) = \delta(\psi, \alpha)$.

  Si $\psi = \text{ff}$, $\mathcal{L}(\widetilde{\text{ff}}) = \emptyset$ et $\forall \alpha \in 2^\Sigma$ $\mathcal{L}(\bar{\delta}(\text{ff}, \alpha)) = \emptyset$. On supposera par la suite que $\psi \neq \text{ff}$

  - Si $u = u_0 \cdot v \in \mathcal{L}(\{\{\psi\}\})$, il existe une exécution acceptante σ de $\mathcal{A}_\varphi$ sur u avec $i(\sigma) = \{\psi\}$. Par définition, il existe $\alpha \in 2^\Sigma$ tel que $u_0 \in \alpha$ et l'ensemble des fils de la racine soit dans $\delta(\psi, \alpha)$. Si on enlève la racine de l'arbre, on obtient une forêt σ', exécution acceptante de $\mathcal{A}_\varphi$ sur le mot v avec $i(\sigma') \in \delta(\psi, \alpha)$, ce qui signifie que $v \in \mathcal{L}(\delta(\psi, \alpha))$. Ainsi, $\exists \alpha \in 2^\Sigma$ $u \in \alpha \cdot \mathcal{L}(\delta(\psi, \alpha))$.
  - Réciproquement, prenons $u = u_0 \cdot v \in \bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\delta(\psi, \alpha))$. Il existe $\alpha \in 2^\Sigma$ tel que $u_0 \in \alpha$ et $v \in \mathcal{L}(\delta(\psi, \alpha))$. Ainsi il existe une exécution acceptante σ' de $\mathcal{A}_\varphi$ sur le mot v avec $i(\sigma') \in \delta(\psi, \alpha)$. Si on ajoute un nœud étiqueté par ψ, devenant la racine d'un arbre σ, et ayant pour fils les racines de σ', σ est une exécution acceptante de $\mathcal{A}_\varphi$ sur u avec $i(\sigma) = \{\psi\}$. Ainsi $u \in \mathcal{L}(\{\{\psi\}\})$.

  La propriété est vérifiée pour ψ formule temporelle.

- Si $\psi = \psi_1 \vee \psi_2$, on a $\widetilde{\psi} = \widetilde{\psi_1} \cup \widetilde{\psi_2}$ et $\bar{\delta}(\psi, \alpha) = \bar{\delta}(\psi_1, \alpha) \cup \bar{\delta}(\psi_2, \alpha)$.

$$\begin{aligned}
\bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\bar{\delta}(\psi, \alpha)) &= \bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \left(\mathcal{L}(\bar{\delta}(\psi_1, \alpha)) \cup \mathcal{L}(\bar{\delta}(\psi_2, \alpha))\right) && \text{lemme 2.6} \\
&= \left(\bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\bar{\delta}(\psi_1, \alpha))\right) \cup \left(\bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\bar{\delta}(\psi_2, \alpha))\right) \\
&= \mathcal{L}(\widetilde{\psi_1}) \cup \mathcal{L}(\widetilde{\psi_2}) && \text{par hypothèse d'induction sur } \psi_1 \text{ et } \psi_2 \\
&= \mathcal{L}(\widetilde{\psi_1} \cup \widetilde{\psi_2}) && \text{lemme 2.6} \\
&= \mathcal{L}(\widetilde{\psi}).
\end{aligned}$$

  La propriété est vérifiée pour $\psi = \psi_1 \vee \psi_2$.

- Si $\psi = \psi_1 \wedge \psi_2$, on a $\widetilde{\psi} = \widetilde{\psi_1} \otimes \widetilde{\psi_2}$ et $\bar{\delta}(\psi, \alpha) = \bigcup_{\alpha_1 \cap \alpha_2 = \alpha} \bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2)$

$$\begin{aligned}
\bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \mathcal{L}(\bar{\delta}(\psi, \alpha)) &= \bigcup_{\alpha \in 2^\Sigma} \alpha \cdot \bigcup_{\alpha_1 \cap \alpha_2 = \alpha} \mathcal{L}(\bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2)) && \text{lemme 2.6} \\
&= \bigcup_{\alpha_1, \alpha_2 \in 2^\Sigma} (\alpha_1 \cap \alpha_2) \cdot \mathcal{L}(\bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2)) \\
&= \bigcup_{\alpha_1, \alpha_2 \in 2^\Sigma} (\alpha_1 \cap \alpha_2) \cdot \left(\mathcal{L}(\bar{\delta}(\psi_1, \alpha_1)) \cap \mathcal{L}(\bar{\delta}(\psi_2, \alpha_2))\right) && \text{lemme 2.7} \\
&= \bigcup_{\alpha_1, \alpha_2 \in 2^\Sigma} \left(\alpha_1 \cdot \mathcal{L}(\bar{\delta}(\psi_1, \alpha_1))\right) \cap \left(\alpha_2 \cdot \mathcal{L}(\bar{\delta}(\psi_2, \alpha_2))\right) && \text{car } \alpha_1, \alpha_2 \subseteq \Sigma \\
&= \left(\bigcup_{\alpha_1 \in 2^\Sigma} \alpha_1 \cdot \mathcal{L}(\bar{\delta}(\psi_1, \alpha_1))\right) \cap \left(\bigcup_{\alpha_2 \in 2^\Sigma} \alpha_2 \cdot \mathcal{L}(\bar{\delta}(\psi_2, \alpha_2))\right) \\
&= \mathcal{L}(\widetilde{\psi_1}) \cap \mathcal{L}(\widetilde{\psi_2}) && \text{par hypothèse d'induction sur } \psi_1 \text{ et } \psi_2 \\
&= \mathcal{L}(\widetilde{\psi_1} \otimes \widetilde{\psi_2}) && \text{lemme 2.7} \\
&= \mathcal{L}(\widetilde{\psi}).
\end{aligned}$$

  La propriété est vérifiée pour $\psi = \psi_1 \wedge \psi_2$.

$\square$

Lemme 3.2 *$\forall\psi$ sous-formule de $\varphi, \mathcal{L}(\widetilde{\psi}) = \{u \in \Sigma^\omega \mid u \models \psi\}$*

Démonstration par induction sur ψ, en utilisant le lemme 3.1

- Si $\psi = \mathrm{tt}$:
  $\bar{\delta}(\mathrm{tt}, \alpha) = \begin{cases} \{\emptyset\} & \text{si } \alpha = \Sigma \\ \emptyset & \text{sinon} \end{cases}$

  $\mathcal{L}(\widetilde{\mathrm{tt}}) = \Sigma \cdot \Sigma^\omega = \{u \in \Sigma^\omega \mid u \models \mathrm{tt}\}$
- Si $\psi = \mathrm{ff}$:
  $\bar{\delta}(\mathrm{ff}, \alpha) = \emptyset, \forall\alpha \in 2^\Sigma$

  $\mathcal{L}(\widetilde{\mathrm{ff}}) = \emptyset = \{u \in \Sigma^\omega \mid u \models \mathrm{ff}\}$
- Si $\psi = p$ avec $p \in Prop$:
  $\bar{\delta}(p, \alpha) = \begin{cases} \{\emptyset\} & \text{si } \alpha = \Sigma_p \\ \emptyset & \text{sinon} \end{cases}$

  $\mathcal{L}(\widetilde{p}) = \Sigma_p \cdot \Sigma^\omega = \{u \in \Sigma^\omega \mid u \models p\}$
- Si $\psi = \neg p$ avec $p \in Prop$:
  $\bar{\delta}(\neg p, \alpha) = \begin{cases} \{\emptyset\} & \text{si } \alpha = \overline{\Sigma}_p \\ \emptyset & \text{sinon} \end{cases}$

  $\mathcal{L}(\widetilde{\neg p}) = \overline{\Sigma}_p \cdot \Sigma^\omega = \{u \in \Sigma^\omega \mid u \models \neg p\}$
- Si $\psi = \psi_1 \vee \psi_2$:
  $$\begin{aligned} \mathcal{L}(\widetilde{\psi_1 \vee \psi_2}) &= \mathcal{L}(\widetilde{\psi_1} \cup \widetilde{\psi_2}) = \mathcal{L}(\widetilde{\psi_1}) \cup \mathcal{L}(\widetilde{\psi_2}) \text{ - lemme 2.6} \\ &= \{u \in \Sigma^\omega \mid u \models \psi_1\} \cup \{u \in \Sigma^\omega \mid u \models \psi_2\} \text{ - hypothèse d'induction} \\ &= \{u \in \Sigma^\omega \mid u \models \psi_1 \vee \psi_2\} \end{aligned}$$
- Si $\psi = \psi_1 \wedge \psi_2$:
  $$\begin{aligned} \mathcal{L}(\widetilde{\psi_1 \wedge \psi_2}) &= \mathcal{L}(\widetilde{\psi_1} \otimes \widetilde{\psi_2}) = \mathcal{L}(\widetilde{\psi_1}) \cap \mathcal{L}(\widetilde{\psi_2}) \text{ - lemme 2.7} \\ &= \{u \in \Sigma^\omega \mid u \models \psi_1\} \cap \{u \in \Sigma^\omega \mid u \models \psi_2\} \text{ - hypothèse d'induction} \\ &= \{u \in \Sigma^\omega \mid u \models \psi_1 \wedge \psi_2\} \end{aligned}$$
- Si $\psi = \mathcal{X}\ \psi_1$:
  $\bar{\delta}(\mathcal{X}\ \psi_1, \alpha) = \begin{cases} \widetilde{\psi_1} & \text{si } \alpha = \Sigma \\ \emptyset & \text{sinon} \end{cases}$

  $\mathcal{L}(\widetilde{\mathcal{X}\ \psi_1}) = \Sigma \cdot \mathcal{L}(\widetilde{\psi_1}) = \Sigma \cdot \{u \in \Sigma^\omega \mid u \models \psi_1\} = \{u \in \Sigma^\omega \mid u \models \mathcal{X}\ \psi_1\}$
  en utilisant l'hypothèse d'induction sur ψ_1.
- Si $\psi = \psi_1\ \mathcal{U}\ \psi_2$:
  $\bar{\delta}(\psi_1\ \mathcal{U}\ \psi_2, \alpha) = (\bar{\delta}(\psi_1, \alpha) \otimes \{\{\psi_1\ \mathcal{U}\ \psi_2\}\}) \ \cup\ \bar{\delta}(\psi_2, \alpha)$

  Considérons une exécution acceptante σ de $\mathcal{A}_\varphi$ sur un mot u avec $i(\sigma) = \{\psi\}$.

  Si l'ensemble des étiquettes des fils de la racine est dans $\bar{\delta}(\psi_1, \alpha) \otimes \{\{\psi_1\ \mathcal{U}\ \psi_2\}\}$, alors un des fils de la racine a pour étiquette ψ et se trouve dans la même situation que la racine. Ce cas ne peut pas toujours se produire, car $\psi \in F$ donc σ ne peut contenir de branche infinie dont les nœuds sont tous étiquetés par ψ.

  Ainsi la racine est le premier élément d'une succession de $k+1$ nœuds $n_0, \ldots, n_k$ tous étiquetés par ψ tels que $\forall 0 \leq i < k\ \exists\alpha_i \in 2^\Sigma$ avec $u_i \in \alpha_i$ et l'ensemble des étiquettes des fils de n_i, privé de n_{i+1}, est dans $\bar{\delta}(\psi_1, \alpha_i)$, et $\exists\alpha_k \in 2^\Sigma$ avec $u_k \in \alpha_k$ et l'ensemble des étiquettes des fils de n_k est dans $\bar{\delta}(\psi_2, \alpha_k)$ (Fig. 1).

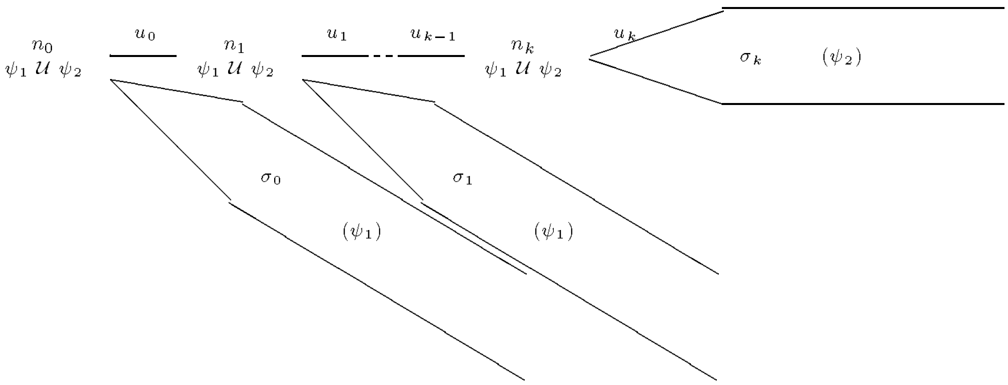

FIG. 1 – *Calcul pour* $\psi_1 \;\mathcal{U}\; \psi_2$

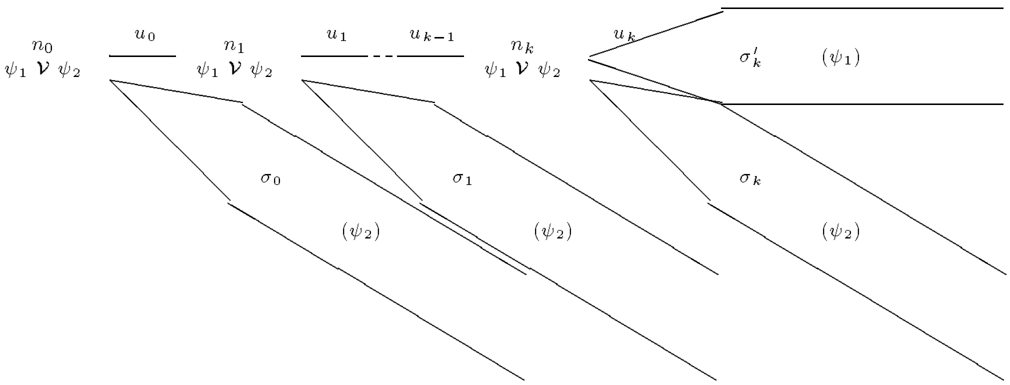

FIG. 2 – *Calcul pour* $\psi_1 \;\mathcal{V}\; \psi_2$ *- cas 1*

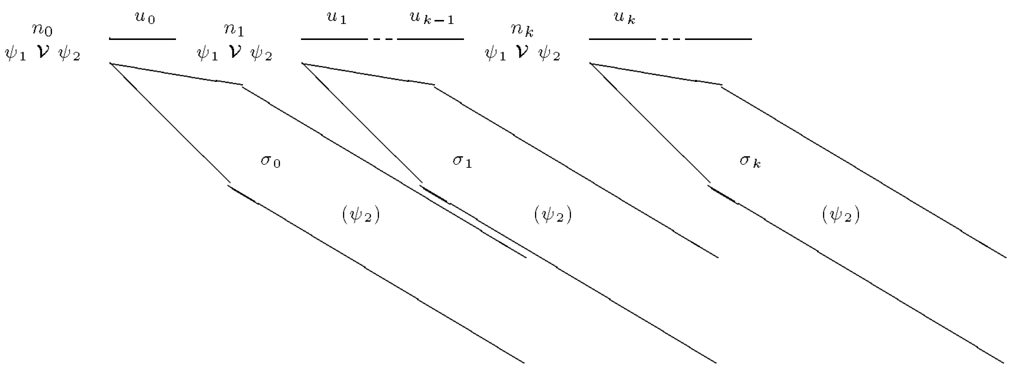

FIG. 3 – *Calcul pour* $\psi_1 \;\mathcal{V}\; \psi_2$ *- cas 2*

D'après le lemme 3.1,
- $\forall 0 \leq i < k$ $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_1, \alpha_i))$ donc $u_i \ldots u_n \ldots \in \mathcal{L}(\widetilde{\psi_1})$.
- $u_k \in \alpha_k$ et $u_{k+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_k))$ donc $u_k \ldots u_n \ldots \in \mathcal{L}(\widetilde{\psi_2})$.

Par hypothèse d'induction, $\forall 0 \leq i < k$ $u_i \ldots u_n \ldots \models \psi_1$ et $u_k \ldots u_n \ldots \models \psi_2$. D'où $u \models \psi$.

Réciproquement, si $u \models \psi_1 \; \mathcal{U} \; \psi_2$, $\exists k \; \forall i < k$ $u_i \ldots u_n \ldots \models \psi_1$ et $u_k \ldots u_n \ldots \models \psi_2$. En utilisant l'hypothèse d'induction puis le lemme 3.1, $\forall 0 \leq i \leq k$ $\exists \alpha_i \in 2^{\Sigma}$ tels que $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_1, \alpha_i))$ si $i \neq k$ et $u_{k+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_k))$.

Ainsi on peut construire $k+1$ exécutions $\sigma_0, \ldots, \sigma_k$ de $\mathcal{A}_\varphi$ sur $u_i \ldots u_n \ldots, 0 \leq i \leq k$ avec $i(\sigma_i) \in \bar{\delta}(\psi_1, \alpha_i)$ si $i \neq k$ et $i(\sigma_k) \in \bar{\delta}(\psi_2, \alpha_k)$. En créant $k+1$ nœuds $n_0, \ldots, n_k$ étiquetés par ψ, chaque n_i ayant pour fils les racines de la forêt σ_i ainsi que le nœud n_{i+1} si $i \neq k$, n_0 devient l'unique racine de cet arbre et on obtient une exécution σ de $\mathcal{A}_\varphi$ sur u avec $i(\sigma) = \{\psi\}$, acceptante car $\sigma_0, \ldots, \sigma_k$ le sont et que les nœuds et branchements ajoutés sont en nombre fini. D'où $u \in \mathcal{L}(\widetilde{\psi})$.

Ainsi $\mathcal{L}(\widetilde{\psi_1 \; \mathcal{U} \; \psi_2}) = \{u \in \Sigma^\omega \mid u \models \psi_1 \; \mathcal{U} \; \psi_2\}$

– Si $\psi = \psi_1 \; \mathcal{V} \; \psi_2$:
$\bar{\delta}(\psi_1 \; \mathcal{V} \; \psi_2, \alpha) = (\bar{\delta}(\psi_2, \alpha) \otimes \{\{\psi_1 \; \mathcal{V} \; \psi_2\}\}) \; \cup \bigcup_{\alpha_1 \cap \alpha_2 = \alpha} \bar{\delta}(\psi_1, \alpha_1) \otimes \bar{\delta}(\psi_2, \alpha_2)$

Considérons une exécution acceptante σ de $\mathcal{A}_\varphi$ sur un mot u avec $i(\sigma) = \{\psi\}$. Ici, il est possible que σ contienne une branche infinie dont tous les nœuds sont étiquetés par ψ.

Premier cas : la racine est le premier élément d'une succession de $k+1$ nœuds $n_0, \ldots, n_k$ tous étiquetés par ψ tels que $\forall 0 \leq i < k$ $\exists \alpha_i \in 2^{\Sigma}$ avec $u_i \in \alpha_i$ et l'ensemble des étiquettes des fils de n_i, privé de n_{i+1}, est dans $\bar{\delta}(\psi_2, \alpha_i)$, et l'ensemble des étiquettes des fils de n_k est la réunion d'un élement de $\bar{\delta}(\psi_1, \alpha_k)$ et d'un élément de $\bar{\delta}(\psi_2, \alpha_k)$ (Fig. 2).

D'après le lemme 3.1,
- $\forall 0 \leq i \leq k$ $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_i))$ donc $u_i \ldots u_n \ldots \in \mathcal{L}(\widetilde{\psi_2})$.
- $u_k \in \alpha_k$ et $u_{k+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_1, \alpha_k))$ donc $u_k \ldots u_n \ldots \in \mathcal{L}(\widetilde{\psi_1})$.

Par hypothèse d'induction, $\forall 0 \leq i \leq k$ $u_i \ldots u_n \ldots \models \psi_2$ et $u_k \ldots u_n \ldots \models \psi_1$. D'où $u \models \psi$.

Deuxième cas : la racine est le premier élément d'une succession infinie de nœuds $n_0, \ldots$ tous étiquetés par ψ tels que $\forall i \geq 0$ $\exists \alpha_i \in 2^{\Sigma}$ avec $u_i \in \alpha_i$ et l'ensemble des étiquettes des fils de n_i, privé de n_{i+1}, est dans $\bar{\delta}(\psi_2, \alpha_i)$ (Fig. 3).

D'après le lemme 3.1,
$\forall i \geq 0$ $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_i))$ donc $u_i \ldots u_n \ldots \in \mathcal{L}(\widetilde{\psi_2})$.

Par hypothèse d'induction, $\forall i \geq 0$ $u_i \ldots u_n \ldots \models \psi_2$. D'où $u \models \psi$.

Réciproquement, si $u \models \psi$, deux cas se présentent :

Premier cas : $\exists k \; \forall i \leq k$ $u_i \ldots u_n \ldots \models \psi_2$ et $u_k \ldots u_n \ldots \models \psi_1$. En utilisant l'hypothèse d'induction puis le lemme 3.1, $\forall 0 \leq i \leq k$ $\exists \alpha_i \in 2^{\Sigma}$ tels que $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_i))$ et $u_{k+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_1, \alpha_k))$.

Ainsi on peut construire $k+2$ exécutions $\sigma_0, \ldots, \sigma_k, \sigma'_k$ de $\mathcal{A}_\varphi$ sur $u_i \ldots u_n \ldots, 0 \leq i \leq k$ avec $i(\sigma_i) \in \bar{\delta}(\psi_2, \alpha_i)$ et $i(\sigma'_k) \in \bar{\delta}(\psi_1, \alpha_k)$. En créant $k+1$ nœuds $n_0, \ldots, n_k$ étiquetés par ψ, chaque n_i ayant pour fils les racines de la forêt σ_i ainsi que le nœud n_{i+1} si $i \neq k$, et les racines de σ'_k sinon, n_0 devient l'unique racine de cet arbre et on obtient une exécution σ de $\mathcal{A}_\varphi$ sur u avec $i(\sigma) = \{\psi\}$, acceptante car $\sigma_0, \ldots, \sigma_k, \sigma'_k$ le sont et que les nœuds et branchements ajoutés sont en nombre fini. D'où $u \in \mathcal{L}(\widetilde{\psi})$.

Deuxième cas : $\forall i \geq 0$ $u_i \ldots u_n \ldots \models \psi_2$. En utilisant l'hypothèse d'induction puis le lemme 3.1, $\forall i \geq 0$ $\exists \alpha_i \in 2^{\Sigma}$ tels que $u_i \in \alpha_i$ et $u_{i+1} \ldots u_n \ldots \in \mathcal{L}(\bar{\delta}(\psi_2, \alpha_i))$.

Ainsi on peut construire une infinité d'exécutions $\sigma_0, \ldots$ de $\mathcal{A}_\varphi$ sur $u_i \ldots u_n \ldots, i \geq 0$ avec $i(\sigma_i) \in \bar{\delta}(\psi_2, \alpha_i)$. En créant une infinité de nœuds $n_0, \ldots$ étiquetés par ψ, chaque n_i ayant pour fils les racines de la forêt σ_i ainsi que le nœud n_{i+1}, n_0 devient l'unique racine de cet arbre et on obtient une exécution σ de $\mathcal{A}_\varphi$ sur u avec $i(\sigma) = \{\psi\}$, acceptante car $\sigma_0, \ldots$ le sont et que la seule branche infinie qui ne se termine pas comme une branche inifinie d'un σ_i est étiquetée par des nœuds de $Q \backslash F$. D'où $u \in \mathcal{L}(\widetilde{\psi})$.

Ainsi $\mathcal{L}(\widetilde{\psi_1 \ \mathcal{V} \ \psi_2}) = \{u \in \Sigma^\omega \mid u \models \psi_1 \ \mathcal{V} \ \psi_2\}$

□

Théorème 3.3 $\mathcal{L}(\mathcal{A}_\varphi) = \{u \in \Sigma^\omega \mid u \models \varphi\}$

Démonstration Application du lemme précédent au cas $\psi = \varphi$ □

3.3 Complexité

L'automate $\mathcal{A}_\varphi$ possède au maximum $|\varphi|$ états, où $|\varphi|$ est la taille de la formule φ.

4 Automates de Büchi généralisés

4.1 Définition d'un automate de Büchi généralisé

Un automate de Büchi généralisé avec acceptation sur les transitions, appelé par la suite simplement automate de Büchi généralisé est un quintuplet $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ où

- Q est l'ensemble des états
- Σ est l'alphabet
- $\delta : Q \times 2^{\Sigma} \to 2^{Q}$ est la fonction de transition
- $I \subseteq Q$ est l'ensemble des états initiaux
- $\mathcal{T} = T_1, \ldots, T_r$ avec $T_j \subseteq Q \times 2^{\Sigma} \times Q$ donne les conditions d'acceptations.

Une exécution σ d'un automate de Büchi généralisé sur un mot $u_0 u_1 \ldots u_n \ldots$ est une suite $q_0 q_1 \ldots q_n \ldots$ d'éléments de Q tels que

- $q_0 \in I$
- $\forall i \geq 0 \; \exists \alpha \in 2^{\Sigma} \; u_i \in \alpha$ et $q_{i+1} \in \delta(q_i, \alpha)$

Elle est acceptante si il existe une infinité de transitions (q_i, α, q_{i+1}) dans chacun des T_j, et dans ce cas on dit que l'automate accepte le mot $u_0 u_1 \ldots u_n \ldots$ On note $\mathcal{L}(\mathcal{G})$ l'ensemble des mots acceptés par $\mathcal{G}$.

4.2 Simplification d'un automate de Büchi généralisé

Théorème 4.1 (transitions) *Si* $\exists q \in Q \; \exists \alpha_1, \alpha_2 \in 2^{\Sigma} \; \exists q' \in \delta(q, \alpha_1) \cap \delta(q, \alpha_2)$ *tels que* $\forall 1 \leq j \leq r$ $(q, \alpha_1, q') \in T_j \iff (q, \alpha_2, q') \in T_j$, $\alpha_1 \subsetneq \alpha_2$, *alors on peut supprimer* q' *de* $\delta(q, \alpha_1)$.

Démonstration Ce résultat est clair car si $u_i \in \alpha_1$ alors $u_i \in \alpha_2$, et ainsi toute exécution acceptante de $\mathcal{G}$ peut être considérée comme une exécution acceptante de l'automate modifié et réciproquement. □

Théorème 4.2 (états) *Si* $\exists q_1, q_2 \in Q \; q_1 \neq q_2$ *tels que* $\forall \alpha \in 2^{\Sigma} \; \delta(q_1, \alpha) = \delta(q_2, \alpha)$ *et* $\forall q' \in Q$ $\forall 1 \leq j \leq r \; (q_1, \alpha, q') \in T_j \iff (q_2, \alpha, q') \in T_j$, *alors on peut supprimer de* Q *l'état* q_1, *en remplaçant chaque occurence de* q_1 *dans* I *et dans* δ *par* q_2.

Démonstration Voir la démonstration du théorème 2.9 □

5 Passage d'un automate alternant très faible à un automate de Büchi généralisé

Le but de cette section est de construire à partir d'un automate alternant très faible un automate de Büchi généralisé qui reconnaît exactement le même langage.

5.1 Construction de l'automate de Büchi généralisé

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ un automate alternant très faible co-Büchi (la construction reste valable pour un automate alternant très faible de Büchi en remplaçant F par $Q \backslash F$, voir la proposition 2.2). On définit l'automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}} = (Q', \Sigma', \delta', I', \mathcal{T})$ par

- $Q' = 2^Q$
- $\Sigma' = \Sigma$
- $$\begin{aligned}\delta'(\{p_1, \ldots, p_n\}, \alpha) &= \bigcup_{\alpha_1 \cap \ldots \alpha_n = \alpha} \bigotimes_{i=1}^{n} \delta(p_i, \alpha_i) \\ &= \left\{ \bigcup_{i=1}^{n} e_i \mid \exists \alpha_1, \ldots, \alpha_n \ \alpha = \bigcap_{i=1}^{n} \alpha_i \text{ et } \forall 1 \leq i \leq n \ e_i \in \delta(p_i, \alpha_i) \right\}\end{aligned}$$
- $I' = I$
- si $F = \{f_1, \ldots, f_r\}$ alors $\mathcal{T} = T_1, \ldots, T_r$ avec
  $T_j = \big\{(q, \alpha, q') \mid f_j \notin q' \text{ ou } \exists \beta \in 2^\Sigma \ \exists e \in \delta(f_j, \beta) \ \alpha \subseteq \beta \text{ et } e \subseteq q' \backslash \{f_j\}\big\}$

Avec la condition $f_j \notin q$ au lieu de $f_j \notin q'$, les transitions de T_j seraient celles qui ne correspondent pas à un arc $f_j \to f_j$ dans une exécution de $\mathcal{A}$. Toutefois la définition utilisée reste correcte, et permet en pratique que la simplification de l'automate de Büchi généralisé obtenu soit plus efficace.

5.2 Démonstration de l'équivalence

Théorème 5.1 $\mathcal{L}(\mathcal{G}_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$

Démonstration Soit $u = u_0 \ldots u_n \ldots \in \mathcal{L}(\mathcal{A})$. D'après la proposition 2.5, il existe un Q-DAG σ, exécution acceptante de $\mathcal{A}$ sur u. Soit i un entier positif fixé. Soit $q_i = \{p_i^1, \ldots, p_i^{n_i}\}$ l'ensemble des étiquettes des nœuds de σ de profondeur i, et $\forall 0 \leq k \leq n_i$ soit e_i^k l'ensemble des étiquettes des fils de p_i^k : $\exists \alpha_i^k \in 2^\Sigma \ u_i \in \alpha_i^k$ et $e_i^k \in \delta(p_i^k, \alpha_i^k)$. Posons $\alpha_i = \bigcap_{k=1}^{n_i} \alpha_i^k$. Par définition l'ensemble des fils des nœuds de profondeur i est l'ensemble des nœuds de profondeur $i+1$. Ainsi, $q_{i+1} = \bigcup_{k=1}^{n_i} e_i^k$. De tout ce qui précède on déduit aisément que $q_{i+1} \in \delta'(q_i, \alpha_i)$ et $u_i \in \alpha_i$. De plus $q_0 = i(\sigma) \in I$. Ainsi $\sigma' = q_0 \ldots q_n \ldots$ est une exécution de $\mathcal{G}_{\mathcal{A}}$ sur $u_0 \ldots u_n \ldots$

Soient $n \geq 0$ et $1 \leq j \leq r$. Si $f_j \notin q_{n+1}$ alors $(q_n, \alpha_n, q_{n+1}) \in T_j$. Sinon, il existe dans σ un nœud N d'étiquette f_j et de profondeur $n+1$. Si il existait dans σ une branche infinie partant de N et dont tous les nœuds aient pour étiquette f_j, σ ne serait pas acceptante. Ainsi, il existe un successeur N' de N de profondeur n' qui a pour étiquette f_j et dont aucun fils n'a pour étiquette f_j. Soit e l'ensemble des étiquettes des fils de N' : $\exists \beta \in 2^\Sigma \ u_{n'} \in \beta$ et $e \in \delta(f_j, \beta)$. De plus, par construction, $\alpha_{n'} \subseteq \beta$ et $e \subseteq q_{n'+1}$ avec $f_j \notin e$ donc $e \subseteq q_{n'+1} \backslash \{f_j\}$. Ainsi $\forall n \geq 0 \ \forall 1 \leq j \leq r$ $\exists n' \geq n \ (q_{n'}, \alpha_{n'}, q_{n'+1}) \in T_j$. L'exécution σ' est acceptante.

Réciproquement, soit $\sigma' = q_0, \ldots, q_n, \ldots$ une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur un mot $u_0 \ldots u_n \ldots$ On construit un Q-DAG σ de la façon suivante : pour i entier positif fixé, si $q_i = \{p_i^1, \ldots, p_i^{n_i}\}$, on construit n_i nœuds de niveau i d'étiquettes respectives $p_i^1, \ldots, p_i^{n_i}$. Par hypothèse, il existe $\alpha_i^1, \ldots, \alpha_i^{n_i} \in 2^\Sigma$ et $e_i^1, \ldots, e_i^{n_i} \in 2^Q$ tels que $q_{i+1} = \bigcup_{k=1}^{n_i} e_i^k$ et $\forall 1 \leq k \leq n_i$ $u_i \in \alpha_i^k$ et $e_i^k \in \delta(p_i^k, \alpha_i^k)$. De plus si $(q_i, \bigcap_{k=1}^{n_i} \alpha_i^k, q_{i+1}) \in T_j$, alors $f_j \notin q_{i+1}$ ou si $\exists 1 \leq k \leq n_i$ tel que $p_i^k = f_j$ alors on peut choisir e_i^k tel que $f_j \notin e_i^k$. On prend alors pour successeurs du nœud étiqueté par p_i^k l'ensemble des nœuds du niveau $i+1$ dont l'étiquette est dans e_i^k. Ainsi par construction, l'ensemble des étiquettes des racines est un élément de I et pour tout nœud d'étiquette p et de profondeur i, il existe $\alpha \in 2^\Sigma$ tel que $u_i \in \alpha$ et l'ensemble des étiquettes des fils de ce nœud est un élément de $\delta(q, \alpha)$. Ainsi σ est une exécution de $\mathcal{A}$ sur le mot $u_0 \ldots u_n \ldots$.

Soient $n \geq 0$ et $1 \leq j \leq r$. σ' est une exécution acceptante donc $\exists n' > n$ $(q_{n'}, \alpha_{n'}, q_{n'+1}) \in T_j$. Par construction, soit aucun nœud de profondeur $n' + 1$ n'a pour étiquette f_j, soit s'il existe un nœud de profondeur n' et d'étiquette f_j, aucun de ses fils n'a pour étiquette f_j. Ainsi, sur chaque branche de σ, nécéssairement un des deux nœuds de profondeur n' ou $n' + 1$ n'a pas pour étiquette f_j : $\forall 1 \leq j \leq r$, toute branche infinie contient une infinité de nœuds non étiquetés par f_j. Or $\mathcal{A}$ est un automate alternant très faible donc, d'après la proposition 2.4, σ est une exécution acceptante. D'après la proposition 2.5, $u \in \mathcal{L}(\mathcal{A})$. □

5.3 Simplification de $\mathcal{G}_\mathcal{A}$

Théorème 5.2 *Si $\exists q \in Q'$ $\exists \alpha_1, \alpha_2 \in 2^\Sigma$ $\exists q_1 \in \delta'(q, \alpha_1), q_2 \in \delta'(q, \alpha_2)$ tels que $\forall 1 \leq j \leq r$ $(q, \alpha_1, q_1) \in T_j \iff (q, \alpha_2, q_2) \in T_j$, $\alpha_1 \subseteq \alpha_2$ et $q_2 \subseteq q_1$ mais $\alpha_1 \neq \alpha_2$ ou $q_2 \neq q_1$, alors on peut supprimer q_1 de $\delta'(q, \alpha_1)$.*

Démonstration À partir d'une exécution acceptante σ de $\mathcal{G}_\mathcal{A}$ sur un mot u, on construit comme dans la démonstration du théorème 5.1 un Q-DAG acceptant de $\mathcal{A}$ sur u. Dans ce Q-DAG, si l'ensemble des étiquettes des nœuds à une profondeur i donnée est q, et si $u_i \in \alpha_1$ et l'ensemble des étiquettes des nœuds à la profondeur $i+1$ est q_1, alors on supprime les nœuds à la profondeur $i + 1$ étiquetés dans $q_1 \backslash q_2$ ainsi que les nœuds devenus non accessibles suite à cette modification. Le nouveau Q-DAG obtenu en répétant cette opération autant de fois que possible dans l'ordre croissant des profondeurs est encore une exécution acceptante de $\mathcal{A}$ sur u. Comme dans la démonstration du théorème 5.1, on le transforme en une exécution acceptante de l'automate modifié sur u, car elle n'utilise plus la transition (q, α_1, q_1).

Réciproquement, toute exécution acceptante de l'automate modifié sur un mot u est aussi une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u.

Cette suppression ne modifie donc pas le langage reconnu par l'automate $\mathcal{G}_\mathcal{A}$. □

5.4 Complexité

L'automate $\mathcal{G}_\mathcal{A}$ possède au maximum $2^{|Q|}$ états et $|Q|$ conditions d'acceptations, où $|Q|$ est le nombre d'états de $\mathcal{A}$.

6 Automates de Büchi

6.1 Définition d'un automate de Büchi

Un automate de Büchi est un quintuplet $\mathcal{B} = (Q, \Sigma, \delta, I, F)$ où

- Q est l'ensemble des états
- Σ est l'alphabet
- $\delta : Q \times 2^{\Sigma} \to 2^{Q}$ est la fonction de transition
- $I \subseteq Q$ est l'ensemble des états initiaux
- $F \subseteq Q$ est l'ensemble des états répétés

Une exécution σ d'un automate de Büchi sur un mot $u_0 u_1 \ldots u_n \ldots$ est une suite $q_0 \ldots q_n \ldots$ d'éléments de Q tels que

- $q_0 \in I$
- $\forall i \geq 0\ \exists \alpha \in 2^{\Sigma}\ u_i \in \alpha$ et $q_{i+1} \in \delta(q_i, \alpha)$

Elle est acceptante si il existe une infinité de i tels que $q_i \in F$. On note $\mathcal{L}(\mathcal{B})$ l'ensemble des mots acceptés par $\mathcal{B}$.

6.2 Simplification d'un automate de Büchi

Théorème 6.1 (transitions) *Si $\exists q \in Q\ \exists \alpha_1, \alpha_2 \in 2^{\Sigma}\ \exists q' \in \delta(q, \alpha_1) \cap \delta(q, \alpha_2)$ tels que $\alpha_1 \subsetneq \alpha_2$, alors on peut supprimer q' de $\delta(q, \alpha_1)$.*

Démonstration Ce résultat est clair car si $u_i \in \alpha_1$ alors $u_i \in \alpha_2$, et ainsi toute exécution acceptante de $\mathcal{B}$ peut être considérée comme une exécution acceptante de l'automate modifié et réciproquement. □

Théorème 6.2 (états) *Si $\exists q_1, q_2 \in Q\ q_1 \neq q_2$ tels que $\forall \alpha \in 2^{\Sigma}\ \delta(q_1, \alpha) = \delta(q_2, \alpha)$ et $q_1, q_2 \in F$ ou $q_1, q_2 \notin F$ alors on peut supprimer de Q l'état q_1, en remplaçant chaque occurence de q_1 dans I et dans δ par q_2.*

Démonstration Voir la démonstration du théorème 2.9 □

7 Passage d'un automate de Büchi généralisé à un automate de Büchi

Le but de cette section est de construire à partir d'un automate de Büchi généralisé un automate de Büchi qui reconnaît exactement le même langage.

7.1 Construction de l'automate de Büchi

Soit $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ un automate de Büchi généralisé avec $\mathcal{T} = T_1, \ldots, T_r$. On définit l'automate de Büchi $\mathcal{B}_{\mathcal{G}} = (Q', \Sigma', \delta', I', F)$ par

- $Q' = Q \times \{0, \ldots, r\}$
- $\Sigma' = \Sigma$
- $\delta'((q, j), \alpha) = \{(q', j') \mid q' \in \delta(q, \alpha) \text{ et } j' = \text{next}(j, (q, \alpha, q'))\}$
  avec $\text{next}(j, (q, \alpha, q')) = \begin{cases} \max\{j \leq i \leq r \mid \forall j < k \leq i \ (q, \alpha, q') \in T_k\} & \text{si } j \neq r \\ \max\{0 \leq i \leq r \mid \forall 0 < k \leq i \ (q, \alpha, q') \in T_k\} & \text{si } j = r \end{cases}$
- $I' = I \times \{0\}$
- $F = Q \times \{r\}$

7.2 Démonstration de l'équivalence

Théorème 7.1 $\mathcal{L}(\mathcal{B}_{\mathcal{G}}) = \mathcal{L}(\mathcal{G})$

Démonstration

Soit $\sigma = q_0, \ldots, q_n, \ldots$ une exécution acceptante de $\mathcal{G}$ sur un mot $u_0 \ldots u_n \ldots$: $q_o \in I$ et $\forall i \geq 0 \ \exists \alpha_i \in 2^{\Sigma}$ tel que $u_i \in \alpha_i$ et $q_{i+1} \in \delta(q_i, \alpha_i)$. Posons $j_0 = 0$ et $\forall i \geq 0 \ j_{i+1} = \text{next}(j_i, (q_i, \alpha_i, q_{i+1}))$. Il est clair que $\sigma' = (q_0, j_0), \ldots, (q_n, j_n), \ldots$ est une exécution de $\mathcal{B}_{\mathcal{G}}$ sur $u_0 \ldots u_n \ldots$

Supposons que σ' n'est pas acceptante : à partir d'un certain rang, tous les j_i sont différents de r, donc à partir d'un certain rang, la suite $(j_i)_i \geq 0$ est croissante à valeurs entières, et majorée strictement par r. Par conséquent $\exists 0 \leq k < r \ \exists n \geq 0 \ \forall i \geq n \ j_i = k$. Or σ est acceptante donc $\exists i \geq n \ (q_i, \alpha_i, q_{i+1}) \in T_k$. Dans ce cas on devrait avoir $j_{i+1} > j_i$ ce qui est absurde. σ' est donc acceptante.

Réciproquement, soit $\sigma' = (q_0, j_0), \ldots, (q_n, j_n), \ldots$ est une exécution acceptante de $\mathcal{B}_{\mathcal{G}}$ sur un mot $u_0 \ldots u_n \ldots$: $q_o \in I$, $j_0 = 0$ et $\forall i \geq 0 \ \exists \alpha_i \in 2^{\Sigma}$ tel que $u_i \in \alpha_i$, $q_{i+1} \in \delta(q_i, \alpha_i)$ et $j_{i+1} = \text{next}(j_i, (q_i, \alpha_i, q_{i+1}))$. Ainsi $\sigma = q_0, \ldots, q_n, \ldots$ est une exécution de $\mathcal{G}$ sur $u_0 \ldots u_n \ldots$

$\forall i \geq 0$ si $j_i \neq r$ alors $\forall j_i < k \leq j_{i+1} \ (q_i, \alpha_i, q_{i+1}) \in T_k$ et si $j_i = r$ alors $\forall 0 < k \leq j_{i+1} \ (q_i, \alpha_i, q_{i+1}) \in T_k$. Il apparaît clairement que si $i_1 \neq i_2$ et $j_{i_1} = j_{i_2} = r$, alors entre q_{i_1} et q_{i_2} il existe au moins une transition (q_i, α_i, q_{i+1}) dans chacun de T_k, $1 \leq k \leq r$. Ainsi, puisque σ est acceptante, il existe une infinité de $i \geq 0$ tels que $j_i = r$ et il en résulte que σ' est acceptante.
$\square$

7.3 Complexité

L'automate $\mathcal{B}_{\mathcal{G}}$ possède au maximum $|Q| \times (r+1)$ états, où $|Q|$ est le nombre d'états et r le nombre de conditions d'acceptation de $\mathcal{G}$.

8 Récapitulatif : passage d'une formule LTL à un automate de Büchi

Il suffit d'appliquer successivement les étapes exposées dans les sections 3, 5 et 7, en y ajoutant les algorithmes de simplification proposés ci-dessous. La taille maximale de l'automate de Büchi obtenu est $(|\varphi|+1) \times 2^{|\varphi|}$, mais en pratique l'automate obtenu est souvent de taille bien inférieure.

8.1 Algorithme de simplification à la volée

Dans les deux dernières étapes, il est important de simplifier les automates à la volée, c'est-à-dire au fur et à mesure de leur construction. En pratique, on applique donc l'algorithme ci-dessous.

```
créer une liste vide d'états à traiter
créer une liste vide d'états traités
calculer les états initiaux et les ajouter à la liste d'états à traiter

tant que la liste d'états à traiter n'est pas vide
    prendre un état dans la liste d'états à traiter et l'en enlever
    tant que cet état a des transitions
        calculer une nouvelle transition de l'état
        comparer cette transition aux transitions précédemment calculées
        si cette transition est inutile
            ne rien faire
        sinon si cette transition rend d'autres transitions inutiles
            enlever les transitions rendues inutiles
            si cette opération rend des états à traiter non accessibles
                les enlever de la liste d'états à traiter
        sinon
            ajouter cette transition aux transitions de l'état
            si l'état de destination n'est dans aucune liste
                l'ajouter à la liste d'états à traiter
    ajouter cet état à la liste d'états traités
    comparer cet état à ceux de la liste d'états traités
    s'il est équivalent à un autre état
        le marquer comme inutile
pour chaque état de la liste d'états traités
    si cet état est marqué comme inutile
        supprimer cet état et ses transitions
    sinon pour chaque transition de cet état
        si cette transition pointe vers un état marqué comme inutile
        la faire pointer vers l'état qui rend cet état inutile
```

FIG. 4 – *Algorithme de simplification à la volée*

Cet algorithme est plus efficace si les transitions sont générées séquentiellement, et non en bloc, car ainsi les transitions inutiles peuvent être effacées avant que toutes les transitions ne soient créées, et l'espace mémoire total utilisé est alors réduit.

Notion de « non accessibilité » : on suppose que chaque état contient un entier qui indique le nombre de transitions qui pointent vers lui. Lorsqu'un état est créé, cet entier est initialisé à la valeur 1. À chaque fois qu'une nouvelle transition pointe vers cet état, l'entier est incrémenté. À chaque fois qu'une transition pointant vers cet état est supprimée, l'entier est décrémenté. Si cet entier atteint la valeur 0, l'état devient « non accessible ».

Les notions de « transitions en rendant une autre inutile » et d' « états équivalents » ont été expliquées pour chaque type d'automate dans les sections correspondantes.

On remarquera qu'un état inutile n'est pas supprimé directement, afin que s'il est pointé ultérieurement par une nouvelle transition, il ne soit pas recalculé. Les seuls états que l'on peut supprimer au fur et à mesure sont les états non traités devenus non accesibles, car leurs transitions n'ont jamais été calculées. Par contre un état traité devenu non accessible est conservé, car s'il devient à nouveau accessible par la suite, il ne faut pas refaire tous les calculs. S'il est encore non accessible à la fin de l'algorithme, il est alors supprimé.

Quand un état devient inutile, on garde en mémoire l'état qui le rend inutile, car c'est vers ce nouvel état que toutes les transitions pointant précédemment vers l'ancien état devront pointer.

8.2 Algorithmes de simplification additionnelle

On remarque que la simplification à la volée n'est pas suffisante. En effet, supprimer des états peut créer des transitions redondantes et rendre de nouveaux états équivalents, supprimer des transitions peut rendre des états équivalents et en rendre d'autres non accessibles.

Ainsi, pour simplifier plus efficacement chacun des automates obtenus avant d'exécuter l'étape suivante de l'algorithme, il faut alterner simplification des états, simplification des transitions, et réduction aux états accessibles et co-accessibles, ceci jusqu'à ce qu'aucune de ces opérations n'aie plus aucun effet. Ci-dessous sont proposés des algorithmes additionnels de simplification des transitions et des états pouvant être utilisés.

```
pour chaque état
    pour chaque paire de transitions de cet état
        si une transition est rendue inutile par l'autre
            la supprimer
```

FIG. 5 – *Algorithme de simplification des transitions*

```
pour chaque état
    comparer cet état aux états précédents
    s'il est équivalent à un autre état
        le marquer comme inutile
pour chaque état
    si cet état est marqué comme inutile
        supprimer cet état et ses transitions
    sinon pour chaque transition de cet état
        si cette transition pointe vers un état marqué comme inutile
        la faire pointer vers l'état qui rend cet état inutile
```

FIG. 6 – *Algorithme de simplification des états*

9 Exemple

Pour comprendre le fonctionnement de notre algorithme, le plus simple est de traiter le cas d'un exemple. Nous développons donc ci-dessous les différentes étapes de l'algorithme appliqué à $\theta_1 = \neg(GFp \to G(q \to Fr)) = GFp \wedge F(q \wedge G\neg r) = (\mathrm{ff}\ \mathcal{V}\ (\mathrm{tt}\ \mathcal{U}\ p)) \wedge (\mathrm{tt}\ \mathcal{U}\ (q \wedge (\mathrm{ff}\ \mathcal{V}\ \neg r)))$

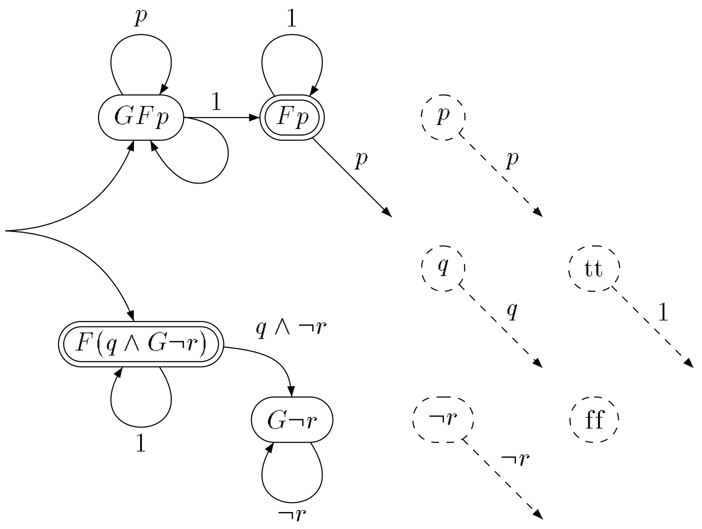

FIG. 7 – *Automate $\mathcal{A}_{\theta_1}$. Les états en pointillés sont les états non accessibles*

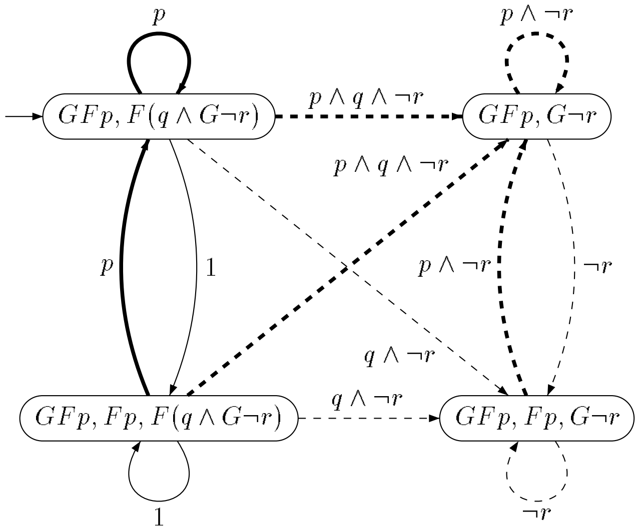

FIG. 8 – *Automate $\mathcal{G}_{\mathcal{A}_{\theta_1}}$ avant simplification. Les transitions en pointillés sont dans $T_{F(q\wedge G\neg r)}$. Les transitions en gras sont dans T_{Fp}*

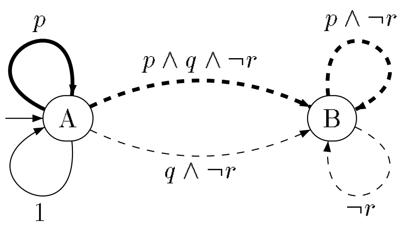

Fig. 9 – *Automate* $\mathcal{G}_{\mathcal{A}_{\theta_1}}$ *après simplification. Les transitions en pointillés sont dans* $T_{F(q \wedge G\neg r)}$. *Les transitions en gras sont dans* T_{Fp}

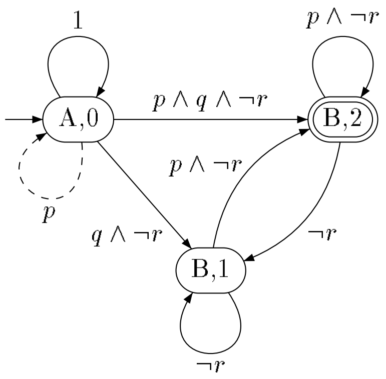

Fig. 10 – *Automate* $\mathcal{B}_{\mathcal{G}_{\mathcal{A}_{\theta_1}}}$. *La transition en pointillés est enlevée lors de la simplification*

10 Performances et comparaison aux travaux existants

Spin est un logiciel libre développé par Bell Labs depuis 1980 ([Hol97]), permettant de faire du model-checking. Il comporte un module qui construit à partir d'une formule LTL φ un automate de Büchi $\mathcal{B}_\varphi$ qui reconnaît exactement les mots qui satisfont cette formule, basé sur un algorithme sensiblement différent de celui que nous utilisons. Cet algorithme est explicité dans [GPVW95].

Les statistiques données par la suite on été effectuées sur un station Sun Ultra 10. La version de Spin utilisée est la version 3.3.10, dernière version publiée à cette date. Spin et Alternating-LTL sont programmés en C, tous deux avec un niveau d'optimisation égal. Les temps sont des temps « utilisateur » Unix. La mémoire consommée est donnée par un module interne de Spin qui compte toute la mémoire allouée pour le module concerné uniquement.

10.1 Formules aléatoires

Les statistiques du tableau 2 sont issues de l'utilisation d'un programme de test développé par H. Tauriainen, conçu pour comparer différents programmes de construction d'un automate de Büchi à partir d'une formule LTL, et pour vérifier ces programmes en effectuant un certain nombre de tests sur les automates générés pour une formule tirée aléatoirement et sa négation. Pour plus de renseignements, voir [Tau99].

	Alternating-LTL		Spin	
φ_1	0.03 sec	23 états	18 min	47 états
$\neg\varphi_1$	0.03 sec	13 états	9 min	28 états
φ_2	0.04 sec	5 états	2.53 sec	6 états
$\neg\varphi_2$	0.04 sec	9 états	0.16 sec	6 états
φ_3	0.02 sec	9 états	0.2 sec	7 états
$\neg\varphi_3$	0.03 sec	11 états	7.17 sec	13 états
φ_4	0.01 sec	2 états	0.15 sec	3 états
$\neg\varphi_4$	0.02 sec	1 état	0.04 sec	2 états
φ_5	0.08 sec	30 états	Stoppé après 1 h	
$\neg\varphi_5$	0.54 sec	177 états	Stoppé après 1 h	
φ_6	0.03 sec	15 états	13.96 sec	30 états
$\neg\varphi_6$	0.03 sec	7 états	7.59 sec	24 états
φ_7	0.01 sec	2 états	0.01 sec	3 états
$\neg\varphi_7$	0.01 sec	2 états	0.01 sec	3 états
φ_8	0.03 sec	19 états	0.46 sec	19 états
$\neg\varphi_8$	0.03 sec	20 états	0.33 sec	15 états
φ_9	0.06 sec	22 états	Stoppé après 1 h	
$\neg\varphi_9$	0.03 sec	20 états	1.93 sec	19 états
φ_{10}	0.04 sec	9 états	0.10 sec	11 états
$\neg\varphi_{10}$	0.06 sec	7 états	0.07 sec	8 états

TAB. 2 – *Comparaison en temps et en espace sur 10 formules aléatoires de taille 15 avec 5 propositions atomiques et leurs négations*

10.2 Formules usuelles de model-checking

Les statistiques du tableau 3 ont été effectuées sur les formules θ_n. On remarque l'explosion de Spin en temps de calcul et en mémoire. Il est commun d'utiliser des formules avec cinq ou six conditions d'équité, et on comprend bien qu'il est impossible d'utiliser Spin pour de telles formules, aussi bien à cause du temps de calcul qu'à cause de la mémoire consommée. En comparaison, Alternating-LTL possède une consommation très raisonnable et huit ou neuf conditions d'équité ne sont pas un obstacle au calcul.

	Alternating-LTL			Spin		
θ_1	$<$ 0.01 sec	9 ko	3 états	0.3 sec	460 ko	3 états
θ_2	$<$ 0.01 sec	11 ko	4 états	8 sec	4,2 Mo	4 états
θ_3	$<$ 0.01 sec	19 ko	5 états	4 min	52 Mo	5 états
θ_4	0.08 sec	38 ko	6 états	5 h 30	970 Mo	6 états
θ_5	0.6 sec	48 ko	7 états			
θ_6	6.7 sec	88 ko	8 états			
θ_7	54 sec	176 ko	9 états			
θ_8	10 min	248 ko	10 états			
θ_9	1 h 30	490 ko	11 états			

TAB. 3 – *Comparaison en temps et en espace sur les formules θ_n pour $1 \leq n \leq 8$*

Conclusion, ouvertures

La dernière section montre des résultats très prometteurs. Cependant, comme nous l'avons précisé dans l'introduction, certains travaux améliorant Spin ont été effectués, et il faudra comparer de la même façon notre implémentation aux autres programmes existants.

D'autre part, la simplification que nous utilisons reste simple, et nous obtenons parfois des automates un peu plus gros que ceux que propose Spin. Il faudra donc prendre en compte les travaux récents sur la simplification des automates de Büchi (par exemple [SB00] et [EH00]) et les appliquer à notre algorithme, en les modifiant éventuellement pour qu'ils puissent aussi être utilisés sur les automates de Büchi généralisés.

Références

[DGV99] M. Daniele, F. Guinchiglia, and M. Y. Vardi. Improved automata generation for linear time temporal logic. In *Proc of 11th conference on computer aided verification (CAV99)*, pages 249–260, Springer-Verlag, Berlin, March 1999. LNCS 1633.

[EH00] K. Etessami and G. J. Holzmann. Optimizing buchi automata. In *Proc. CONCUR2000.* Springer Verlag / LNCS, Aug 2000.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Hol97] G. J. Holtzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, May 1997.

[SB00] F. Somenzi and R. Bloem. Efficient büchi automata from ltl formulae. In *Proceedings of 12th Int. Conf. on computer aided verification*, 2000.

[Tau99] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into büchi automata. In *Workshop concurrency, specifications and programming*, pages 251–262, Warsaw, Poland, Sept. 1999.

[Var96] M. Y. Vardi. *An automata-theoretic approach to linear temporal logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag Inc., 1996.

Partie 4

Thèse d'informatique

Article présenté à la conférence CAV'01
cosigné par P. Gastin

juillet 2001

Fast LTL to Büchi Automata Translation

Paul Gastin and Denis Oddoux

LIAFA, Université Paris 7, Paris, France
{Paul.Gastin,Denis.Oddoux}@liafa.jussieu.fr

Abstract. We present an algorithm to generate Büchi automata from LTL formulae. This algorithm generates a very weak alternating co-Büchi automaton and then transforms it into a Büchi automaton, using a generalized Büchi automaton as an intermediate step. Each automaton is simplified on-the-fly in order to save memory and time. As usual we simplify the LTL formula before any treatment. We implemented this algorithm and compared it with Spin: the experiments show that our algorithm is much more efficient than Spin. The criteria of comparison are the size of the resulting automaton, the time of the computation and the memory used. Our implementation is available on the web at the following address: `http://verif.liafa.jussieu.fr/ltl2ba`

1 Introduction

To prove that a program satisfies some property, a standard method is to use Linear Time Logic (LTL) model checking. When the property is expressed with an LTL formula, the model checker usually transforms the negation of this formula into a Büchi automaton, builds the product of that automaton with the program, and checks this product for emptiness. In this paper we focus on the generation of a Büchi automaton from an LTL formula, trying to improve the time and space of the computation and the size of the resulting automaton.

Spin [4] is a very popular LTL model checker. However, the algorithm it uses to generate a Büchi automaton from an LTL formula, presented in [3], may be quite slow and may need a large amount of memory, even for some usual LTL formulae. In particular, this algorithm has a very bad behaviour on formulae with fairness conditions: it is almost impossible to use Spin to generate a Büchi automaton from a formula containing 5 or more fairness conditions, both because of the computation time and of the memory needed. For example, consider a simple response formula $\mathrm{G}(q \to \mathrm{F}\, r)$ with n fairness conditions:

$$\theta_n = \neg((\mathrm{G}\,\mathrm{F}\, p_1 \wedge \ldots \wedge \mathrm{G}\,\mathrm{F}\, p_n) \to \mathrm{G}(q \to \mathrm{F}\, r)) \ . \tag{1}$$

A formula of this type is very often encountered in LTL model checking. Moreover, the fairness conditions and the right-hand side property are usually more complex. The value of n is very often greater than 5. Alas, in this case, Spin fails to produce the Büchi automaton within a reasonable amount of time and memory (see Table 1).

	Spin		Wring		EQLTL	LTL2BA−		LTL2BA	
	time	space	time	space	time	time	space	time	space
θ_1	0.18	460	0.56	4,100	16	0.01	9	0.01	9
θ_2	4.6	4,200	2.6	4,100	16	0.01	19	0.01	11
θ_3	170	52,000	16	4,200	18	0.01	86	0.01	19
θ_4	9,600	970,000	110	4,700	25	0.07	336	0.06	38
θ_5			1,000	6,500	135	0.70	1,600	0.37	48
θ_6			8,400	13,000	N/A	12	8,300	4.0	88
θ_7			72,000†	43,000†		220	44,000	32	175
θ_8						4,200	260,000	360	250
θ_9						97,000	1,600,000	3,000	490
θ_{10}								36,000	970

Table 1. Comparison on the formulae θ_n for $1 \leq n \leq 10$. Time is in sec, space in kB. (*N/A*) : no answer from the server within 24 h. (†) : the program died, giving no result

Spin's algorithm was improved by [1] (LTL2AUT), [2] (EQLTL), [10] (Wring): these papers did not modify the basis of the algorithm, but improved it using the same core algorithm, rewriting LTL formulae, and simplifying the resulting Büchi automaton. These improvements are quite efficient but the actual transformation of the LTL formula to a Büchi automaton, which is similar to the tableau construction explained in [3], may still perform badly on some natural formulae such as θ_n. Some experiments are presented in Table 1. Note that Wring is written in Perl while Spin and LTL2BA are written in C and that EQLTL is used through a web server. Hence the figures are still relevant but should not be compared litterally. See Sect. 7 for more details.

In this paper, we present a new algorithm to generate a Büchi automaton from an LTL formula. Our algorithm is not based on the tableau construction presented in [3]. Instead, using the classical construction (see e.g. [12]), we first produce an alternating automaton from the LTL formula, with n states where n is less than the size of the formula. This alternating automaton turns out to be *very weak* as shown by Rohde [9]. Thanks to that property, instead of generating directly a Büchi automaton with $2^n \times 2^n$ states, we are able to build first a generalized Büchi automaton, that is a Büchi automaton with labels and accepting conditions on transitions instead of states, with at most 2^n states. Using a generalized Büchi automaton is one of the most important improvements of our algorithm. The best solution would be to design a model-checking algorithm using directly this generalized Büchi automaton, but in order to compare our work with other ones and to use existing model-checking algorithms, we transform this automaton into a classical Büchi automaton. The method we use is very classical, and we obtain a Büchi automaton with at most $n \times 2^n$ states.

The second main improvement stems from our simplifications of the automata. Since our construction goes in several steps, we are able to simplify the automata at each step, improving the efficiency of the following steps. The simplifications dramatically reduce the number of states and transitions of the automata, especially of the generalized Büchi automaton. Moreover, each simplification is performed on-the-fly during the construction of each automaton. This is a major improvement on a posteriori simplifications. The amount of memory used is about the size of the simplified automaton, instead of being the size of the unsimplified automaton which may be quite huge. The time needed is also reduced dramatically because we are exploring a much smaller part of the automaton during the construction.

Using our new algorithm, we built a tool which is available on the web at `http://verif.liafa.jussieu.fr/ltl2ba`. Our tool is much more efficient than any other tool we have tried, in computation time and especially in memory. The results of our algorithm on the formulae θ_n with on-the-fly simplifications (LTL2BA) and with a posteriori simplifications (LTL2BA–) are detailed in Table 1. More experimental results are presented in Sect. 7. There we also discuss the size of the generated automaton. From this point of view also our algorithm is usually better than Spin though occasionally it may produce a bigger automaton. Note that Spin, LTL2BA– and LTL2BA give exactly the same resulting automaton on the formulae θ_n. Wring and EQLTL give bigger automata.

The paper is organized as follows. Section 2 begins with some preliminaries defining linear temporal logic and its semantics. Sections 3 to 5 describe our algorithm and some proofs of its correctness. Section 6 presents our simplification methods and Sect. 7 describes some experimental results.

2 Preliminaries: Linear Temporal Logic (LTL)

LTL was introduced to specify the properties of the executions of a system. A finite set Prop contains all atomic properties of states. With the standard Boolean operators ($\neg$, $\wedge$, $\vee$) we can only express static properties. For dynamical properties, we use temporal operators such as X (next), U (until), R (release), F (eventually) and G (always).

Definition 1 (syntax). *The set of* LTL formulae *on the set* Prop *is defined by the grammar* $\varphi ::= p \mid \neg\,\varphi \mid \varphi \vee \varphi \mid \mathrm{X}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi$, *where* p *ranges over* Prop.

The semantics of LTL usually defines whether an execution σ of a given system satisfies a formula. Actually the semantics only depends on the atomic propositions that stand in each state of σ. Then for our purpose we consider only sequences of sets of atomic propositions.

Definition 2 (semantics). *Let* $u = u_0u_1\dots$ *be a word in* Σ^ω *with* $\Sigma = 2^{\mathrm{Prop}}$. *Let* φ *be an LTL formula. The relation* $u \models \varphi$ *(*u models φ*) is defined as follows:*

- $u \models p$ *if* $p \in u_0$,
- $u \models \neg\,\varphi_1$ *if* $u \not\models \varphi_1$,
- $u \models \varphi_1 \vee \varphi_2$ *if* $u \models \varphi_1$ *or* $u \models \varphi_2$,
- $u \models \mathrm{X}\,\varphi_1$ *if* $u_1u_2\dots \models \varphi_1$,
- $u \models \varphi_1\,\mathrm{U}\,\varphi_2$ *if* $\exists k \geq 0$, $u_ku_{k+1}\dots \models \varphi_2$ *and* $\forall 0 \leq i < k$, $u_iu_{i+1}\dots \models \varphi_1$.

Only basic operators have been defined above. We will of course also use the derived operators defined by:

$$\mathrm{tt} \stackrel{def}{=} p \vee \neg p \ , \ \ \mathrm{ff} \stackrel{def}{=} \neg\, \mathrm{tt} \ , \ \ \varphi_1 \wedge \varphi_2 \stackrel{def}{=} \neg(\neg \varphi_1 \vee \neg \varphi_2) \ , (2)$$

$$\varphi_1 \mathrel{\mathrm{R}} \varphi_2 \stackrel{def}{=} \neg(\neg \varphi_1 \mathrel{\mathrm{U}} \neg \varphi_2) \ , \ \ \mathrm{F}\,\varphi \stackrel{def}{=} \mathrm{tt} \mathrel{\mathrm{U}} \varphi \ \text{ and } \ \mathrm{G}\,\varphi \stackrel{def}{=} \mathrm{ff} \mathrel{\mathrm{R}} \varphi = \neg\,\mathrm{F} \neg \varphi \ .(3)$$

An LTL formula that is neither a disjunction ($\vee$) nor a conjunction ($\wedge$) is called a *temporal formula*.

An LTL formula can be written in *negative normal form*, using only the predicates in Prop, their negations, and the operators $\vee$, $\wedge$, X, U, and R. Notice that this operation does not change the number of temporal operators of the formula. From now on, we suppose that every LTL formula is in negative normal form.

Example 1. Let $\theta = \neg(\mathrm{G}\,\mathrm{F}\,p \rightarrow \mathrm{G}(q \rightarrow \mathrm{F}\,r))$ be our running example along the paper. The negative normal form of θ is $(\mathrm{ff} \mathrel{\mathrm{R}} (\mathrm{tt} \mathrel{\mathrm{U}} p)) \wedge (tt \mathrel{\mathrm{U}} (q \wedge (\mathrm{ff} \mathrel{\mathrm{R}} \neg r)))$.

Before any construction our algorithm simplifies the formula, using a set of rewriting rules that reduce the number of temporal operators. This is relevant since the complexity of our algorithm is based on this number. Some of these rules are presented in [2],[10]. We will not discuss them in this paper.

3 LTL to Very Weak Alternating Automata (VWAA)

This section explains a classical construction: building a VWAA from an LTL formula. Alternating automata have been introduced by Muller and Schupp in [6],[7],[8]. Then in [9], Rohde defined VWAA as he needed them for a work on transfinite words. VWAA were also described in [5]. However, our definition is somewhat different from the classical one.

Definition 3. *A co-Büchi* very weak alternating co-Büchi automaton *is a five-tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where:*

- *Q is the set of states,*
- *Let Q' be the set of conjunctions of elements of Q. The empty conjunction is denoted by tt. We identify Q' with 2^Q in the following,*
- *Σ is the alphabet, and we let $\Sigma' = 2^\Sigma$,*
- *$\delta : Q \rightarrow 2^{\Sigma' \times Q'}$ is the transition function,*
- *$I \subseteq Q'$ is the set of initial states,*
- *$F \subseteq Q$ is the set of final states (co-Büchi),*
- *there exists a partial order on Q such that $\forall q \in Q$, all the states appearing in $\delta(q)$ are lower or equal to q.*

The definition of a classical alternating automaton would be the same except for the last condition on the partial order.

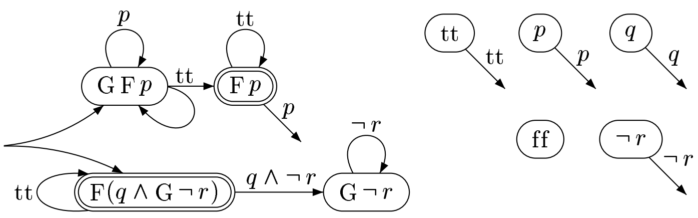

Fig. 1. Automaton $\mathcal{A}_\theta$. Some states (*right*) are unaccessible, they will be removed

Remark 1. The transition function looks different from the usual definition ($\Delta : Q \times \Sigma \to \mathcal{B}^+(Q)$). We made those changes for implementation reasons, in order to ease the manipulation of the data structures and to save time and space during the computation. The classical representation of our transition function is given by:

$$\Delta(q, a) = \bigvee_{\substack{(\alpha,e)\in\delta(q)\\ a\in\alpha}} e \ . \tag{4}$$

Conversely we may obtain our definition from the classical one, essentially by taking the disjunctive normal form. Hence the two definitions are equivalent.

Notice that in the transition function we use Σ' instead of Σ: so that transitions that differ only by the action can be gathered. In practice, this usually reduces a lot the number of transitions. However the automaton still reads words in Σ^ω.

Example 2. You can see the representation of a VWAA on Fig. 1. States in F are circled twice. Notice that arrows with the same origin represent one transition to a conjunction of states. In this example, we have:

- $I = \{\mathrm{G}\,\mathrm{F}\,p \wedge \mathrm{F}(q \wedge \mathrm{G}\,\neg r)\}$,
- $\delta(p) = \{(\Sigma_p, \mathrm{tt})\}$ where $\Sigma_p = \{a \in \Sigma \mid p \in a\}$,
- $\delta(\mathrm{G}\,\mathrm{F}\,p) = \{(\Sigma_p, \mathrm{G}\,\mathrm{F}\,p), (\Sigma, \mathrm{G}\,\mathrm{F}\,p \wedge \mathrm{F}\,p)\}$.

A *run* σ of $\mathcal{A}$ on a word $u_0u_1 \ldots \in \Sigma^\omega$ is a labeled DAG (V, E, λ) such that :

- V is partitioned in $\bigcup_{i=0}^{\infty} V_i$ with $E \subseteq \bigcup_{i=0}^{\infty} V_i \times V_{i+1}$,
- λ: $V \to Q$ is the labeling function,
- $\lambda(V_0) \in I$ and $\forall x \in V_i$, $\exists(\alpha, e) \in \delta(\lambda(x))$, $u_i \in \alpha$ and $e = \lambda(E(x))$.

A run σ is *accepting* if any (infinite) branch in σ has only a finite number of nodes labeled in F (co-Büchi acceptance condition). $\mathcal{L}(\mathcal{A})$ is the set of words on which there exists an accepting run of $\mathcal{A}$. Note that, Büchi and co-Büchi acceptance conditions are equivalent for VWAA; one only has to replace F by $Q \setminus F$.

Example 3. Here is an example of an accepting run of the automaton $\mathcal{A}_\theta$:

$$
\begin{tikzcd}
\mathrm{G\,F}\,p \arrow[r, "\emptyset"] \arrow[dr] & \mathrm{G\,F}\,p \arrow[r, "\{q,r\}"] \arrow[dr] & \mathrm{G\,F}\,p \arrow[r, "\{p,q\}"] & \mathrm{G\,F}\,p \arrow[r, "\{p\}"] & \mathrm{G\,F}\,p \arrow[r, dashed, "\{p\}"] & {} \\
& \mathrm{F}\,p \arrow[r] & \mathrm{F}\,p & & & \\
\mathrm{F}(q \wedge \mathrm{G}\,\neg r) \arrow[r] & \mathrm{F}(q \wedge \mathrm{G}\,\neg r) \arrow[r] & \mathrm{F}(q \wedge \mathrm{G}\,\neg r) \arrow[dr] & & & \\
& & & \mathrm{G}\,\neg r \arrow[r] & \mathrm{G}\,\neg r \arrow[r, dashed] & {}
\end{tikzcd}
$$

In the definition of the VWAA associated with an LTL formula, we use two new operators. $\otimes$ helps treating conjunctions, and $\overline{\psi}$ gives roughly the DNF of ψ, allowing us to restrict the states of the automaton to the temporal subformulae of φ.

Definition 4. *For $J_1, J_2 \in 2^{\Sigma' \times Q'}$ we define*
$J_1 \otimes J_2 = \{(\alpha_1 \cap \alpha_2, e_1 \wedge e_2) \mid (\alpha_1, e_1) \in J_1$ *and* $(\alpha_2, e_2) \in J_2\}$,
For an LTL formula ψ we define $\overline{\psi}$ by: $\overline{\psi} = \{\psi\}$ if ψ is a temporal formula, $\overline{\psi_1 \wedge \psi_2} = \{e_1 \wedge e_2 \mid e_1 \in \overline{\psi_1}$ *and* $e_2 \in \overline{\psi_2}\}$ *and* $\overline{\psi_1 \vee \psi_2} = \overline{\psi_1} \cup \overline{\psi_2}$.

Here is the first step of our algorithm, building a VWAA from an LTL formula. Notice that the number of states of this automaton is at most the size of the formula.

Step 1. Let φ be an LTL formula on a set Prop. We define the VWAA $\mathcal{A}_\varphi$ by:

- Q is the set of temporal subformulae of φ,
- $\Sigma = 2^{\mathrm{Prop}}$,
- $I = \overline{\varphi}$,
- F is the set of until subformulae of φ, that is formulae of type $\psi_1 \,\mathrm{U}\, \psi_2$,
- δ is defined as follows (Δ extends δ to all subformulae of φ):

$$
\begin{cases}
\delta(\mathrm{tt}) = \{(\Sigma, \mathrm{tt})\} \\
\delta(p) = \{(\Sigma_p, \mathrm{tt})\} \text{ where } \Sigma_p = \{a \in \Sigma \mid p \in a\} \\
\delta(\neg p) = \{(\Sigma_{\neg p}, \mathrm{tt})\} \text{ where } \Sigma_{\neg p} = \Sigma \backslash \Sigma_p \\
\delta(\mathrm{X}\,\psi) = \{(\Sigma, e) \mid e \in \overline{\psi}\} \\
\delta(\psi_1 \,\mathrm{U}\, \psi_2) = \Delta(\psi_2) \;\cup (\Delta(\psi_1) \otimes \{(\Sigma, \psi_1 \,\mathrm{U}\, \psi_2)\}) \\
\delta(\psi_1 \,\mathrm{R}\, \psi_2) = \Delta(\psi_2) \;\otimes (\Delta(\psi_1) \cup \{(\Sigma, \psi_1 \,\mathrm{R}\, \psi_2)\})
\end{cases}
$$

$$
\begin{cases}
\Delta(\psi) = \delta(\psi) \text{ if } \psi \text{ is a temporal formula} \\
\Delta(\psi_1 \vee \psi_2) = \Delta(\psi_1) \cup \Delta(\psi_2) \\
\Delta(\psi_1 \wedge \psi_2) = \Delta(\psi_1) \otimes \Delta(\psi_2)
\end{cases}
$$

Using the partial order "subformula of" it is easy to prove that $\mathcal{A}_\varphi$ is very weak.

Remark 2. One can notice that the elements of Σ' used in our definition are intersections of the sets Σ, Σ_p and $\Sigma_{\neg p}$. Hence, they can be denoted by conjunctions of literals, as in the following examples : $p \wedge q \wedge \neg r$ for $\Sigma_p \cap \Sigma_q \cap \Sigma_{\neg r}$, tt for Σ. Note that intersection and test for inclusion can be easily performed with this representation.

Example 4. Figure 1 shows the result of Step 1 on the formula θ defined in Ex. 1.

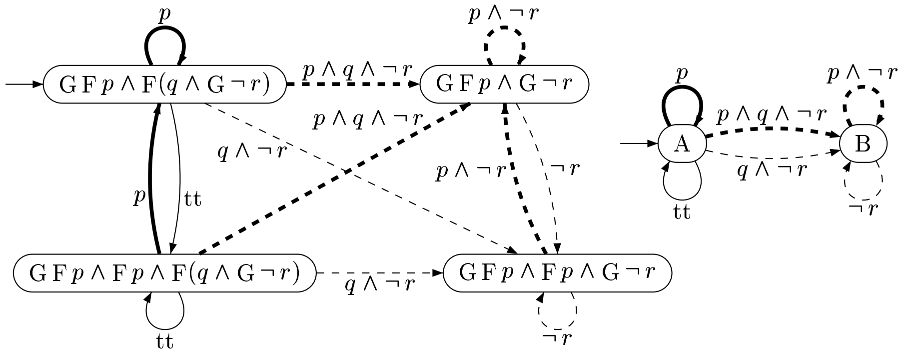

Fig. 2. Automaton $\mathcal{G}_{\mathcal{A}_\theta}$, before (*left*) and after (*right*) simplification

Theorem 1. $\mathcal{L}(\mathcal{A}_\varphi) = \{u \in \Sigma^\omega \mid u \models \varphi\}$.

Proof. The idea of the proof is to show recursively that for any subformula ψ of φ, the language accepted by $\mathcal{A}_\varphi$ with $I = \overline{\psi}$ is equal to $\{u \in \Sigma^\omega \mid u \models \psi\}$. The main difficulties are encountered for $\psi = \psi_1 \mathrm{\,U\,} \psi_2$ (this is where the acceptance condition comes into play) and $\psi = \psi_1 \mathrm{\,R\,} \psi_2$. □

4 VWAA to Generalized Büchi Automata (GBA)

At that point we have obtained a VWAA for our LTL formula φ. The problem is that the usual method to transform an alternating automaton into a Büchi automaton produces an automaton that is much too big. This is why we generate first a GBA, which is a Büchi automaton with *several* acceptance conditions on *transitions* instead of states.

Definition 5. *A* generalized Büchi automaton *is a five-tuple* $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ *where :*

- Q *is the set of states,*
- Σ *is the alphabet, and we let* $\Sigma' \subseteq 2^\Sigma$*,*
- $\delta : Q \to 2^{\Sigma' \times Q}$ *is the transition function,*
- $I \subseteq Q$ *is the set of initial states,*
- $\mathcal{T} = \{T_1, \ldots, T_r\}$ *where* $T_j \subseteq Q \times \Sigma' \times Q$ *are the accepting transitions.*

Example 5. The automata on Fig. 2 are examples of GBAs. In these examples, $r = 2$: dashed transitions are in T_1 and bold transitions are in T_2. An accepting run has to use infinitely many dashed transitions and infinitely many bold transitions.

A *run* σ of $\mathcal{G}$ on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$ of elements of Q such that $q_0 \in I$ and $\forall i \geq 0$, $\exists \alpha_i \in \Sigma'$ such that $u_i \in \alpha_i$ and $(\alpha_i, q_{i+1}) \in \delta(q_i)$. A run σ is *accepting* if for each $1 \leq j \leq r$ it uses infinitely many transitions from T_j. $\mathcal{L}(\mathcal{G})$ is the set of words on which there exists an accepting run of $\mathcal{G}$.

Here is the second step of our algorithm, building a GBA from a co-Büchi VWAA. It can be of course applied to any VWAA, and not only to an automaton issued from Step 1. $\mathcal{G}_\mathcal{A}$ has at most $2^{|Q|}$ states and $|F|$ acceptance sets.

Step 2. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a VWAA with co-Büchi acceptance conditions. We define the GBA $\mathcal{G}_\mathcal{A} = (Q', \Sigma, \delta', I, \mathcal{T})$ where:

- $Q' = 2^Q$ is identified with conjunctions of states as explained in Definition 3,
- $\delta''(q_1 \wedge \ldots \wedge q_n) = \bigotimes_{i=1}^{n} \delta(q_i)$,
- δ' is the set of $\preccurlyeq$-minimal transitions of δ'' where the relation $\preccurlyeq$ is defined by $t' \preccurlyeq t$ if $t = (e, \alpha, e')$, $t' = (e, \alpha', e'')$, $\alpha \subseteq \alpha'$, $e'' \subseteq e'$, and $\forall T \in \mathcal{T}$, $t \in T \Rightarrow t' \in T$,
- $\mathcal{T} = \{T_f \mid f \in F\}$ where
  $T_f = \{(e, \alpha, e') \mid f \notin e' \text{ or } \exists(\beta, e'') \in \delta(f),\ \alpha \subseteq \beta \text{ and } f \notin e'' \subseteq e'\}$.

Remark 3. One may notice that using $f \notin e$ instead of $f \notin e'$ in the definition of T_f would have been more intuitive, since it corresponds to the case where in the run of $\mathcal{A}$ there is no edge with both ends labeled by f. But our definition is also correct. The proof of the following main theorem is more complicated with this definition but the experimental results are much better with it, especially regarding the simplifications.

Example 6. Figure 2 shows the result of Step 2 on the automaton $\mathcal{A}_\theta$ of Fig. 1.

Theorem 2. $\mathcal{L}(\mathcal{G}_\mathcal{A}) = \mathcal{L}(\mathcal{A})$.

Remark 4. This is the point where we need the alternating automaton to be very weak (this theorem is false for classical alternating automata). Consider an infinite branch in a run of $\mathcal{A}$ on a given word : since $\mathcal{A}$ is very weak, the sequence of the labels on this branch is decreasing, and has to be ultimately constant since Q is finite. Then "having only a finite number of nodes labelled in F" is equivalent to "having an infinite number of nodes labelled in $Q \backslash F$". This is crucial in the proof.

Proof. Let $\sigma = (V, E, \lambda)$ be an accepting run of $\mathcal{A}$ on a word $u = u_0 u_1 \ldots$ $V = \bigcup_{i \geq 0} V_i$, $E = \bigcup_{i \geq 0} E_i$, with $E_i \subseteq V_i \times V_{i+1}$. We are first going to build a new run of $\mathcal{A}$ on u, redefining gradually the sets V_i and E_i to $V'_i \subseteq V_i$ and E'_i, $\forall i \geq 0$.

Let $V'_0 = V_0$. Now suppose that V'_i has been defined. By definition of a run, $\forall x \in V'_i$, $\exists \alpha_x$ such that $u_i \in \alpha_x$, and $(\alpha_x, e_x) \in \delta(\lambda(x))$ where $e_x = \lambda(E_i(x))$. Let $\alpha = \bigcap_{x \in V'_i} \alpha_x$ and $e = \bigcup_{x \in V'_i} e_x$.

By definition of δ'', $t = (\lambda(V'_i), \alpha, e)$ is in δ'': there exists a transition $t' = (\lambda(V'_i), \alpha', e')$ in δ' such that $t' \preccurlyeq t$ and t' is minimal. Note that t' is a transition of $\mathcal{G}_\mathcal{A}$, and that $u_i \in \alpha \subseteq \alpha'$. Since $t' \in \delta' \subseteq \delta''$, $\forall x \in V'_i$, $\exists(\alpha'_x, e'_x) \in \delta(\lambda(x))$ such that $\alpha' = \bigcap_{x \in V'_i} \alpha'_x$ and $e'_x = \bigcup_{x \in V'_i} e'$.

Moreover $\forall x \in V'_i$ such that $\lambda(x) = f \in F$ and $t' \in T_f$, there exists $(\alpha''_x, e''_x) \in \delta(f)$ such that $f \notin e''_x \subseteq e'$ and $\alpha' \subseteq \alpha''_x$. For all other elements x of V'_i, let $e''_x = e'_x$ and $\alpha''_x = \alpha'_x$.

Let $V'_{i+1} = \{y \in V_{i+1} \mid \lambda(y) \in e'\}$ and $E'_i = \{(x,y) \in V'_i \times V'_{i+1} \mid \lambda(y) \in e''_x\}$. Note that $\lambda(E'_i(x)) = e''_x$ since $e''_x \subseteq e' = \lambda(V'_i)$ and that $E'_i(V'_i)$ may be *strictly* contained in V'_{i+1}.

Claim. $\forall i \geq 0, \forall f \in F$, the following property holds:
if $\exists (x,y) \in E'_i$, $\lambda(x) = \lambda(y) = f$ then $\exists (x,y) \in E_i$, $\lambda(x) = \lambda(y) = f$.

Proof. If $\forall x \in V'_i$, $\lambda(x) \neq f$ then the claim is true. Otherwise $\exists x \in V'_i \subseteq V_i$, $\lambda(x) = f$. Assume that $\exists y \in E'_i(x)$ with $\lambda(y) = f$. Then we have $f \in e''_x$, and by definition of e''_x we deduce that $t' \notin T_f$. Since $t' \preccurlyeq t$ we have $t \notin T_f$ and we deduce easily that $f \in e_x$, which proves the claim.

Let $V' = \bigcup_{i \geq 0} V'_i$, $E' = \bigcup_{i \geq 0} E'_i$ and λ' be the restriction of λ to V'. From the construction, one can easily see that $\sigma' = (V', E', \lambda')$ is a new run of $\mathcal{A}$ on u. We show first that σ' is an accepting run. Suppose that σ' is not accepting: since $\mathcal{A}$ is very weak, the labels on an infinite branch of a run are ultimately constant. Hence if σ' is not accepting, then there exists an infinite branch of σ' ultimately labeled by some $f \in F$. Using the claim, there exists in σ an infinite branch which is ultimately labeled by f. This is impossible since σ is accepting.

Let $e_i = \lambda(V'_i)$, $\forall i \geq 0$. We have $e_0 = \lambda(V_0) \in I$ and from our construction we get $\forall i \geq 0$, $\exists \alpha_i$ such that $u_i \in \alpha_i$ and (e_i, α_i, e_{i+1}) is a transition of $\mathcal{G}_{\mathcal{A}}$: $\sigma'' = e_0, e_1, \ldots$ is a run of $\mathcal{G}_{\mathcal{A}}$ on u. Now let us prove that σ'' is accepting. Let $i \geq 0$ and $f \in F$. We intend to prove that at some depth $j \geq i$ the transition (e_j, α_j, e_{j+1}) is in T_f.

If $f \notin e_{i+1}$ then $j = i$ will do. Otherwise let $j > i$ be the smallest depth where $(f, f) \notin \lambda(E'_j)$. Note that j exists, otherwise there would be an infinite branch in σ' ultimately labeled by f and σ' would not be accepting. Since we know that $f \in e_j$, let x be the node of V'_j labeled by f. From our construction we know that $\exists (e''_x, \alpha''_x) \in \delta(f)$, $f \notin e''_x \subseteq e_{j+1}$ and $\alpha_j \subseteq \alpha''_x$. We can conclude that (e_j, α_j, e_{j+1}) is in T_f.

Therefore, from any accepting run σ of $\mathcal{A}$, we have built an accepting run σ'' of $\mathcal{G}_{\mathcal{A}}$ on the same word and we get the first inclusion $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{G}_{\mathcal{A}})$.

Conversely let $\sigma' = e_0, e_1, \ldots$ be an accepting run of $\mathcal{G}_{\mathcal{A}}$ on a word $u = u_0 u_1 \ldots$ Hence $e_0 \in I$ and $\forall i \geq 0$, $\exists \alpha_i$, $u_i \in \alpha_i$ and $(\alpha_i, e_{i+1}) \in \delta'(e_i)$. Let $V = \bigcup_{i \geq 0} V_i$ where $V_i = \{(p,i) \mid p \in e_i\}$ and let $\lambda(p,i) = p$ so that $\lambda(V_i) = e_i$.

By definition of δ', $\forall x \in V_i$, $\exists (\alpha_x, e_x) \in \delta(\lambda(x))$ such that $e_x \subseteq e_{i+1}$ and $\alpha_i \subseteq \alpha_x$. Moreover $\forall f \in F$, if $(e_i, \alpha_i, e_{i+1}) \in T_f$ then either $f \notin e_i$, or $\lambda(x) = f$ for some x in V_i and in that case we can choose α_x and e_x such that $f \notin e_x$. Let E be defined by $(x,y) \in E$ if $\exists i \geq 0$, $x \in V_i$, $y \in V_{i+1}$ and $\lambda(y) \in e_x$.

We can easily see that $\sigma = (V, E, \lambda)$ is a run of $\mathcal{A}$ on u. Now suppose that σ is not accepting: as we proved before, there would exist in σ an infinite branch with all nodes ultimately labeled by some $f \in F$. But σ' is accepting so it has

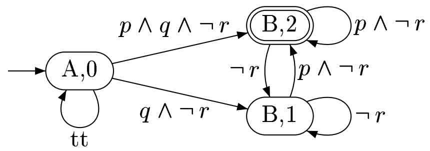

Fig. 3. Automaton $\mathcal{B}_{\mathcal{G}_{\mathcal{A}_\theta}}$ after simplification

infinitely many transitions in T_f, and for each such transition there is no edge in E with both ends labeled by f at the corresponding depth. Hence this is impossible.

Therefore from any accepting run σ' of $\mathcal{G}_\mathcal{A}$, we have built an accepting run σ of $\mathcal{A}$ on the same word, proving the converse inclusion $\mathcal{L}(\mathcal{G}_\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A})$. $\square$

5 GBA to Büchi Automata (BA)

At that point we have obtained a GBA for our LTL formula φ. We simply have to transform it into a BA to complete our algorithm. This construction is quite easy and well-known, but for the sake of completeness we explain it briefly. We will begin by defining a BA, using once more the same modifications concerning the alphabet and the transition function.

Definition 6. *A* Büchi automaton *is a five-tuple $\mathcal{B} = (Q, \Sigma, \delta, I, F)$ where:*

- *Q is the set of states,*
- *Σ is the alphabet, and we let $\Sigma' \subseteq 2^\Sigma$,*
- *$\delta : Q \to 2^{\Sigma' \times Q}$ is the transition function,*
- *$I \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of repeated states (Büchi condition).*

A *run* σ of $\mathcal{B}$ on a word $u_0 u_1 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$ of elements of Q such that $q_0 \in I$ and $\forall i \geq 0, \exists \alpha_i \in \Sigma'$ such that $u_i \in \alpha_i$ and $(\alpha_i, q_{i+1}) \in \delta(q_i)$. A run σ is *accepting* if there exists infinitely many states in F. $\mathcal{L}(\mathcal{B})$ is the set of words on which there exists an accepting run of $\mathcal{B}$.

Here is the third step of our algorithm, building a BA from a GBA. If $\mathcal{B}$ has n states and r acceptance conditions, then $\mathcal{B}_\mathcal{G}$ has at most $(r+1) \times n$ states.

Step 3. Let $\mathcal{G} = (Q, \Sigma, \delta, I, \mathcal{T})$ be a GBA with $\mathcal{T} = \{T_1, \ldots, T_r\}$. We define the BA $\mathcal{B}_\mathcal{G} = (Q \times \{0, \ldots, r\}, \Sigma, \delta', I \times \{0\}, Q \times \{r\})$ where:

- $\delta'((q,j)) = \{(\alpha, (q', j')) \mid (\alpha, q') \in \delta(q) \text{ and } j' = next(j, (q, \alpha, q'))\}$.

with $\text{next}(j,t) = \begin{cases} \max\{j \leq i \leq r \mid \forall j < k \leq i,\ t \in T_k\} \text{ if } j \neq r \\ \max\{0 \leq i \leq r \mid \forall 0 < k \leq i,\ t \in T_k\} \text{ if } j = r \end{cases}$

Example 7. Figure 3 shows the result of Step 3 on the automaton $\mathcal{G}_{\mathcal{A}_\theta}$ of Fig. 2.

Theorem 3. $\mathcal{L}(\mathcal{B}_\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

Remark 5. There exist many similar algorithms transform a GBA into a BA. They often consist in building the synchronous product of the GBA with some automaton verifying that every acceptance condition is verified infinitely often. This automaton differs from one algorithm to another. We chose one that gives good results for the size of the resulting BA after simplification.

6 Simplification

Simplification is really important in our algorithm. Since each step produces a new automaton from the result of the previous step, the more we simplify each result, the faster our algorithm is and the least memory it uses.

After each step, we simplify the automaton obtained, using iteeratively three rules until no more simplification occurs:

- A state that is not accessible can be removed,
- If a transition t_1 implies a transition t_2, then t_2 can be removed,

	$t_1 = (q, \alpha_1, q_1)$ implies $t_2 = (q, \alpha_2, q_2)$ if
In a VWAA,	$\alpha_2 \subseteq \alpha_1$ and $q_1 \subseteq q_2$
In a GBA,	$\alpha_2 \subseteq \alpha_1$, $q_1 = q_2$ and $\forall t \in \mathcal{T}$, $t_2 \in T \Rightarrow t_1 \in T$
In a BA,	$\alpha_2 \subseteq \alpha_1$ and $q_1 = q_2$

- If two states q_1 and q_2 are equivalent, then they can be merged.

	q_1 and q_2 are equivalent if
In a VWAA,	$\delta(q_1) = \delta(q_2)$ and $q_1 \in F \iff q_2 \in F$
In a GBA,	$\delta(q_1) = \delta(q_2)$ and $\forall (\alpha, q') \in \delta(q_1)$, $\forall T \in \mathcal{T}$, $(q_1, \alpha, q') \in T \iff (q_2, \alpha, q') \in T$
In a BA,	$\delta(q_1) = \delta(q_2)$ and $q_1 \in F \iff q_2 \in F$

Note that for a GBA issued from Step 2, the condition $(q_1, \alpha, q') \in T_j$ does not depend on q_1 so that the condition simply becomes $\delta(q_1) = \delta(q_2)$.

This simplification procedure is really efficient to reduce the size of the automata. But the strength of our algorithm is that the last two simplification rules are also used *on-the-fly*: after a transition has been created, it is compared with the other transitions already calculated from the same state, and the ones that become useless are immediately deleted; after all the transitions of a state have been created, that state is compared with the other states that have already been created, and is merged to one of those states if possible. This method is important since usually many states and transitions are to be simplified, and simplifying them on-the-fly saves a lot of time and space.

In Table 1, the results of the algorithm with or without on-the-fly simplification are compared (LTL2BA– is our algorithm with a posteriori simplification only). For the formula θ_n defined in (1), the unsimplified GBA has 2^{n+1} states, whereas the simplified GBA has only 2 states. Using on-the-fly simplification avoids the intermediary exponential automaton which explains the great improvement, even if the time and memory used by LTL2BA are still exponential.

7 Experimental Results

In this section we compare the results of some recent algorithms transforming an LTL formula into a BA.

Spin is a model-checker developed by Bell Labs since 1980. It contains an algorithm transforming an LTL formula into a BA, presented in [3]. The program is written in C, and we used version 3.4.1. (released Aug 2000).

Wring is an algorithm presented in [10]. The program is written in Perl, so the comparison with our work cannot be read litterally, and the amount of memory used is just an approximation we made using the Unix command '`top`'.

EQLTL is an algorithm presented in [2]. The program is not publicly available, but a demo is proposed on the web. All we could do was to measure the time needed by the web interface to start responding to our request. We do not even know what type of machine handles the request. Consequently the times we gave should be taken with caution.

LTL2BA is a program writen in C as Spin, in order to make reliable comparison between the two programs. **LTL2BA**− is the same program, with a prosteriori simplification only.

Tests were made on a Sun Ultra 10 station with 1 GB of RAM.

As explained in the introduction, we compared the tools on usual LTL formulae, taking the example of the formula θ_n defined in (1). The result of the comparison is detailed in Table 1.

Another type of usual LTL formulae, often encountered in model-checking, is formulae like: $\varphi_n = \neg(p_1 \text{ U } (p_2 \text{ U } (\ldots \text{ U } p_n)\ldots)$. We made the same tests on these fomulae in Table 2. Again our algorithm outperforms the other ones.

	Spin		Wring		EQLTL	LTL2BA	
	time	space	time	space	time	time	space
φ_2	0.01	8	0.07	4,100	8	0.01	3.2
φ_3	0.03	110	0.29	4,100	8	0.01	5.5
φ_4	0.75	1,700	1.34	4,200	9	0.01	11
φ_5	43	51,000	10	4,200	11	0.01	13
φ_6	1,200	920,000	92	4,500	15	0.15	25
φ_7			720	6,000	27	9.2	48
φ_8					92	1,200	93

Table 2. Comparison on the formulae φ_n for $2 \leq n \leq 8$. Time is in sec, space in kB

We also compared the algorithms on random LTL formulae of a fixed size, using a tool presented in [11]. For compatibility reasons, the only comparison we could realize was between our alogrithm and Spin's. Here the results are issued from a test on 200 random formulae of size 10, where both algorithms are compared on the *same* formulae. See Table 3 for details.

	Spin		LTL2BA	
	avg.	max.	avg.	max.
time of computation (seconds)	14.23	4521.65	0.01	0.04
number of states	5.74	56	4.51	39
number of transitions	14.73	223	9.67	112

Table 3. Comparison on random formulae of a fixed size

References

1. M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11th International Computer Aided Verification Conference*, pages 249–260, 1999.
2. K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of 11th Int. Conf. on Concurrency Theory (CONCUR)*, 2000.
3. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
4. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
5. O. Kupferman and M. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symposium on Theory of Computing and Systems ISTCS'97*, pages 147–158. IEEE, 1997.
6. D. Muller and P. Schupp. Alternating automata on infinite objects: Determinacy and Rabin's theorem. In *Proceedings of the Ecole de Printemps d'Informatique Théoretique on Automata on Infinite Words*, volume 192 of *LNCS*, pages 100–107, Le Mont Dore, France, May 1984. Springer.
7. D. Muller and P. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, October 1987.
8. D. Muller and P. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1–2):69–107, April 1995.
9. S. Rohde. Alternating automata and the temporal logic of ordinals. *PhD Thesis in Mathematics, University of Illinois at Urbana-Champaign*, 1997.
10. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV: International Conference on Computer Aided Verification*, 2000.
11. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Workshop Concurrency, Specifications and Programming*, pages 251–262, Warsaw, Poland, 1999.
12. M. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag Inc., New York, NY, USA, 1996.