Nouvelles techniques de *model checking* pour la vérification de systèmes complexes

Yann THIERRY-MIEG Soheib BAARIR Alexandre DURET-LUTZ
Fabrice KORDON

Laboratoire d'Informatique de Paris 6,
Université P. & M. Curie,
4 Place Jussieu, 75252 Paris Cedex 05, France

1 Introduction

Les tests et la simulation contribuent depuis longtemps à la validation de systèmes. Cependant, ces techniques ne permettent d'explorer qu'une partie des comportements possibles. Elles diffèrent en cela des techniques de *vérification formelle*, qui garantissent qu'une propriété est vérifiée par la totalité des exécutions possibles du système.

Les deux principales techniques de vérification formelle sont la preuve et le *model checking*. Dans la première un modèle du système à vérifier est vu comme un ensemble d'axiomes qui servent à prouver une propriété. Cette technique n'est pas automatique ; des outils d'aide à la preuve existent mais demandent une forte expertise. Le *model checking*, introduit il y a 20 ans, regroupe plusieurs techniques entièrement automatiques dans lesquelles la propriété à vérifier est testée de façon exhaustive sur l'ensemble des exécutions possibles du système.

Les différentes techniques de *model checking* diffèrent par leur façon de représenter ces ensembles (infinis) d'exécutions. Citons le *model checking* dit *symbolique*, qui travaille de façon ensembliste sur les configurations que peut prendre le système, et le *model checking* par l'*approche automate* [8], dans lequel l'ensemble des exécutions du système ainsi que la propriété sont représentés par des automates. C'est à cette dernière technique que nous nous intéressons ici.

Nous présenterons cette approche automate section 2, et expliquerons son principal inconvénient : l'explosion de l'espace d'état [7], qui limite la taille de systèmes vérifiables. En dépit de cet obstacle, le *model checking* est utilisé avec succès pour la vérification de circuits, aussi bien que pour la vérification de programmes. L'objectif de cet article est de montrer deux nouvelles méthodes réduisant cet espace d'états en exploitant ses symétries, afin de pouvoir traiter des systèmes plus importants. Ces techniques sont le graphe quotient (section 3), et le produit synchronisé symbolique (section 4). Section 5 nous évaluons ces deux méthodes sur un exemple conséquent à l'aide de nos outils.

2 Approche automate du *model checking*

Dans l'approche automate du *model checking* (Fig. 1), le modèle M d'un système à vérifier est vu comme un automate A_M reconnaissant des mots de longueur infinie (on parle d'ω-mots et d'ω-automates). Les lettres de ces ω-mots correspondent chacune à une configuration du système. Un ω-mot représente alors une séquence d'exécution du système qui traverse chacune de ces configurations. Le lan-

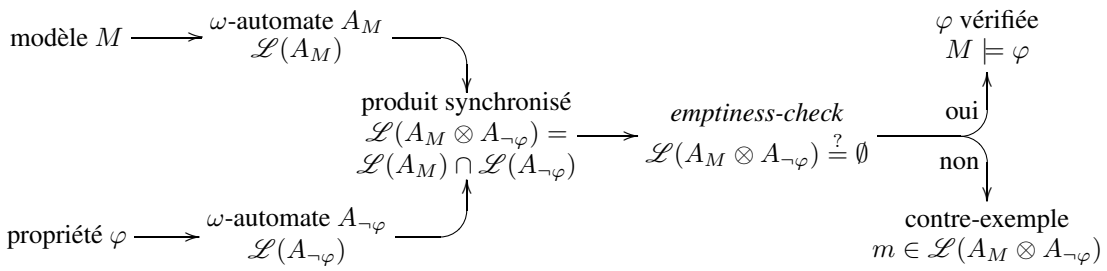

FIG. 1 – Approche automate du *model checking*.

gage de l'automate (noté $\mathscr{L}(A_M)$) est l'ensemble des ω-mots qu'il reconnaît ; il représente l'ensemble des comportements possibles du système.

Par ailleurs, la propriété comportementale φ que l'on veut vérifier sur M est elle-même exprimée par un ω-automate $A_{\neg\varphi}$ dont le langage est l'ensemble des comportements qui *invalident* la propriété φ.

Un enchaînement de deux opérations permet de déterminer si $\mathscr{L}(A_M)$ et $\mathscr{L}(A_{\neg\varphi})$ partagent un ω-mot, c'est-à-dire s'il existe une exécution de M qui ne vérifie pas φ. D'abord le produit synchronisé des deux automates est construit, il s'agit d'un ω-automate reconnaissant l'intersection des deux langages. Enfin un *emptiness-check* de ce produit est effectué, cette opération détermine si le langage reconnu par un automate est vide.

Le langage du produit est vide lorsqu'il n'existe aucune séquence d'exécution de M qui invalide φ. Dans le cas contraire un contre-exemple, c'est-à-dire un ω-mot représentant une exécution du système interdite par φ, peut être retourné.

Cette approche permet de manipuler des ensembles infinis (les langages) en les représentant par des structures finies (les ω-automates). Plusieurs types d'ω-automates existent et se distinguent par la forme de leur *conditions d'acceptation*, c'est-à-dire la façon d'indiquer quels chemins infinis de l'automate correspondent à des ω-mots acceptés. Les plus couramment utilisés, et qui nous intéresseront ici, sont appelés automates de Büchi.

Cette approche permet de vérifier toute propriété exprimable par un automate de Büchi. Cependant, spécifier des propriétés directement sous cette forme n'est pas toujours aisé. On a recours à des logiques comme la *logique temporelle linéaire* (LTL) [8] dont la sémantique est simple et dont on sait transformer les énoncés en automates.

Les formules LTL sont construites à partir de variables propositionnelles appelées *propriétés atomiques*, d'opérateurs booléens ($\neg$, $\vee$, $\wedge$, $\Rightarrow$, ...), et d'*opérateurs temporels* (F, G, U, X, ...).

Les propriétés atomiques permettent de caractériser les configurations du système. Ils s'agit de variables telles qu'$alarme$ (indiquant p.ex. que l'alarme est activée) ou $capteur$ (le capteur est activé) dont la valuation est connue dans chaque configuration du système. La formule $capteur \wedge \neg alarme$ est vraie dans toutes les configurations où le capteur est activé sans que l'alarme ne le soit.

Les opérateurs temporels permettent de relier des configurations au sein d'une séquence d'exécution. $\mathsf{G}\, f$ signifie que la sous-formule LTL f doit toujours (*Generally*) être vérifiée à partir de cet instant. $\mathsf{F}\, f$ indique que f doit être vérifiée à un instant ultérieur (*Future*). $\mathsf{X}\, f$ impose que f soit vérifiée à l'instant immédiatement suivant (*neXt*). Enfin $f\, \mathsf{U}\, g$ est vraie si f est vérifiée jusqu'à ce que g le soit (*Until*). Par exemple la formule $\mathsf{G}(capteur \Rightarrow \mathsf{F}\, alarme)$ spécifie que toute activation du capteur doit être suivie du déclenchement de l'alarme (mais par exemple le capteur pourrait être activé trois fois avant que l'alarme ne se déclenche).

À titre d'exemple, la figure 2 présente un automate de Büchi acceptant toutes les séquences d'exécution qui invalident $\mathsf{G}(capteur \Rightarrow \mathsf{F}\, alarme)$. Pour qu'une séquence d'exécution soit acceptée par

cet automate, les configurations qui la composent doivent chacune satisfaire une formule étiquetant une transition de l'automate, et la séquence de transitions ainsi formée doit débuter par l'un des états initiaux (indiqués par des flèches sans source) et passer infiniment souvent par l'un des états d'acceptation (représentés par des doubles cercles). L'automate de la figure 2 accepte donc uniquement les séquences composées d'un nombre fini de configurations quelconques (la formule $true$ est toujours satisfaite), suivies d'une configuration dans laquelle le capteur est activé sans que l'alarme ne le soit, suivie d'une infinité de configurations dans lesquelles l'alarme n'est pas activée.

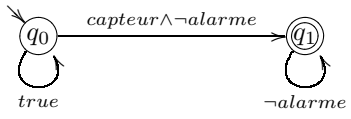

FIG. 2 – $A_{\neg \mathsf{G}(capteur \Rightarrow \mathsf{F}\, alarme)}$.

En pratique cet automate devra être synchronisé avec l'automate A_M représentant tous les comportements du système, pour déterminer ensuite s'il existe un comportement invalidant la formule.

L'*emptiness-check* d'un automate de Büchi peut se faire en temps et espace linéaire par rapport à la taille du produit synchronisé. La taille de ce produit est au pire le produit des tailles des deux automates. La taille de l'automate $A_{\neg\varphi}$ dépend de la propriété à vérifier, mais il ne contient en général que quelques états. En revanche, l'automate A_M qui possède autant d'états que le système a de configurations croît de façon exponentielle avec la complexité du système. Cette explosion du nombre d'états est l'obstacle principal de l'approche automate, et est à l'origine de nombreuses techniques de réduction [7].

Les deux sections qui suivent présentent deux de ces réductions, et nous les illustrerons sur l'exemple de la figure 3. Il s'agit d'un système à deux clients (C_1, C_2) et un serveur (S). Les clients sont identiques : ils construisent un message $m \in \{1, 2\}$ avec `build_msg()`, l'envoient au serveur avec leur identité puis attendent un accusé de réception. Lorsque le serveur reçoit une requête il la traite en fonction de la valeur de m et envoie l'accusé. Tous ces programmes travaillent en boucle. Les communications sont supposées asynchrones et sans perte, mais l'ordre des messages n'est pas garanti (l'ordre de réception des messages par le serveur ne correspond pas forcément à celui de l'envoi par les clients).

Clients (C_1 et C_2)

```
    while True:
①     m = build_msg()  # 1 ou 2
      send(S, (self, m))
②     OK = receive()
```

Serveur (S)

```
    while True:
①     (cl, m) = receive()
      treat(m)
②     send(cl, OK)
```

FIG. 3 – Un système simple à deux clients et un serveur.

Afin que cet exemple conserve des proportions raisonnables, nous avons choisi de ne représenter que deux états par processus. Un client est dans l'état 1 avant de contacter le serveur, il est dans l'état 2 lorsqu'il attend sa réponse. De même, nous considérerons que le serveur est dans l'état 1 lorsqu'il attend une requête, et dans l'état 2 lorsqu'après avoir traité le message il est sur le point de répondre. Un état global de notre système sera constitué du triplet des états de C_1, C_2, et S, ainsi que des messages en transit sur le réseau (R). On précisera de plus l'état de la variable locale `cl` quand le serveur est dans l'état 2. La figure 4 présente le début du graphe des états accessibles de ce système ; le graphe complet comporte 24 états. L'état s_3 correspond à la configuration du système dans laquelle C_1 est dans l'état 1, C_2 dans l'état 2, S dans l'état 1, et le message $\langle C_2, m_2 \rangle$ est en transit sur le réseau.

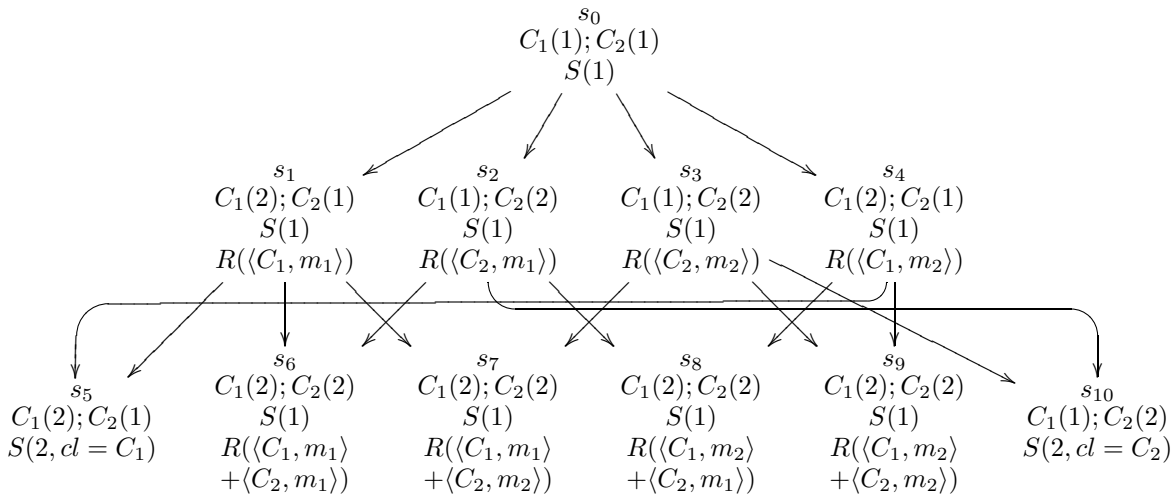

FIG. 4 – 11 premiers états (parmi 24) du graphe d'accessibilité de l'exemple présenté figure 3.

3 Graphe quotient

Comme le montre l'exemple client/serveur (figures 3 et 4), même un système réparti de taille réduite peut engendrer un espace d'états de grande taille. Cet taille croît exponentiellement avec le nombre d'instances : ce même système avec 3 clients, 3 types de messages et 2 serveurs possède 944 états. Une des causes principales de cette *explosion combinatoire* de l'espace d'états est l'entrelacement des exécutions, caractéristique des systèmes répartis. En effet, si un programme séquentiel ne peut se trouver à un instant donné que dans une seule configuration possible, chaque partie d'un système parallèle évolue relativement indépendamment des autres. De ce fait, le nombre de configurations globales du système est lié au produit du nombre de configurations de chaque composant du système. Le nombre de séquences d'exécutions possibles explose davantage en général.

Pour lutter contre cet effet, une approche consiste à exploiter des symétries du système [3] : de nombreux systèmes répartis peuvent être vus comme la composition parallèle de n processus au comportement analogue. Par exemple les clients de notre système se comportent de façon identique. Il est alors possible de construire un graphe d'accessibilité réduit, dit *graphe quotient*, dont les nœuds sont des *classes d'équivalences de configurations*. Le principe consiste à ne pas distinguer les clients les uns des autres, de ce fait au lieu de considérer l'état s_1 où le client 1 vient d'envoyer une requête de type m_1 et, séparément, l'état s_2 où le client 2 vient d'envoyer une requête de type m_1, on considérera la classe d'équivalence s'_1 des états où l'*un des clients* vient d'envoyer une requête.

La figure 5 montre l'effet de cette réduction sur la partie du graphe présentée en figure 4. Dans chaque nœud de ce graphe quotient, les états des clients 1 et 2 peuvent être permutés. On place un arc entre deux classes d'équivalences d'état s'_i et s'_j si au moins une des configurations représentées par s'_i permet d'accéder à une des configurations représentées par s'_j. Pour formaliser ce raisonnement, nous dirons qu'un graphe quotient est produit *sous* une relation d'équivalence $\mathscr{R}$ qui précise quelles sont les instances permutables dans chaque état. Ainsi la relation $\mathscr{R}$ utilisée pour construire le graphe de la figure 5 peut se représenter sous la forme : $\mathscr{R} : C = \{C_1, C_2\}, M = \{m_1\} \cup \{m_2\}$, c'est-à-dire que les instances de client sont permutables, mais pas les instances de messages.

Si l'on considère de plus le fait que les traitements sont identiques quel que soit type du message m_1 ou m_2, on peut obtenir un graphe encore plus réduit représenté figure 6.

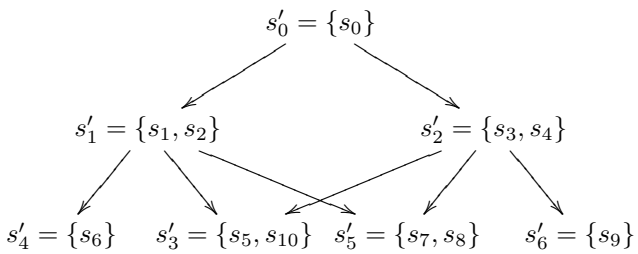

FIG. 5 – 7 premiers états (parmi 14) du graphe quotient de l'exemple présenté figure 4, sous la relation d'équivalence caractérisée par $C = \{C_1, C_2\}$, $M = \{m_1\} \cup \{m_2\}$.

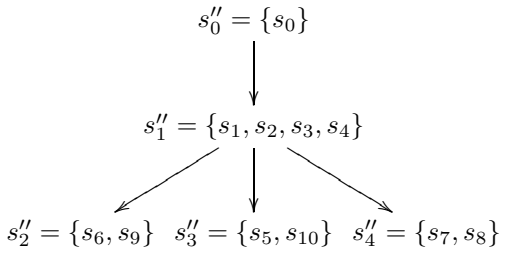

FIG. 6 – 5 premiers états (parmi 10) du graphe quotient de l'exemple présenté figure 4, sous la relation d'équivalence caractérisée par $\mathscr{R} : C = \{C_1, C_2\}, M = \{m_1, m_2\}$.

L'état s''_1 de ce nouveau graphe représente les états ou *un* client à envoyé *un* message ; s''_2 (resp. s''_4) les états où les deux clients ont envoyé une requête *du même type* (resp. *de type différent*) en transit dans le réseau R, et s''_3 les états où le serveur traite une requête.

Comme on peut le voir sur cet exemple, il est primordial pour cette approche de réussir à définir une relation d'équivalence aussi permissive que possible. Nous utilisons à cette fin un module d'analyse structurelle du modèle [6], qui examine une à une chaque opération du modèle. Le postulat de base de cette exploration est que si le modèle était vide, toutes ses instances d'éléments seraient équivalentes, donc permutables. On procède ensuite en examinant pour chaque action les éléments qu'elle distingue les uns des autres : par exemple une action de type `if (x > 3) doA(); else doB();` sépare l'ensemble des entiers en $N = \{0, 1, 2\} \cup \{3, \ldots n\}$. On élabore ensuite la relation d'équivalence sous laquelle le graphe va être construit par raffinement des relations d'équivalences. Ainsi si une autre action du modèle est de type `if (x > 5) doC(); else doD();`, on obtiendra la relation caractérisée par $N = \{0, 1, 2\} \cup \{3, 4, 5\} \cup \{6, \ldots n\}$.

Ce graphe quotient permet de vérifier facilement des propriétés d'accessibilité, en effet tout les états représentés par une classe d'équivalence sont accessibles sur le graphe initial. Cependant, les séquences représentées sur le graphe réduit sont un sur-ensemble des séquences du graphe initial. Par exemple il existe un chemin de s''_1 vers s''_4 figure 6, mais il n'existe pas de chemin de $s_1 \in s''_1$ vers $s_8 \in s''_4$ figure 4.

Il est donc nécessaire d'adapter la relation d'équivalence utilisée à la propriété à vérifier, de façon à s'assurer que les ω-mots acceptés par le graphe réduit et par l'automate de la formule, donc potentiellement contre-exemples, existent réellement sur le graphe initial. Une première approche consiste à raffiner la relation d'équivalence déduite du modèle par une relation d'équivalence déduite de la formule. L'analyse de relation d'équivalence admise par la formule est similaire à celle du modèle : chaque proposition atomique peut donner lieu à la séparation d'éléments *a priori* semblables du modèle, et l'on construit la

relation complète par raffinement des relations découvertes sur chaque proposition atomique.

4 Produit synchronisé symbolique

Le principal défaut de l'approche présentée dans la section précédente est le caractère absolu de la relation d'équivalence construite. En effet, par rapport à la vue d'ensemble de la figure 1, nous avons construit séparément des relations d'équivalence $\mathscr{R}_M$ et $\mathscr{R}_\varphi$ en fonction du modèle et de la formule. Cependant la construction du produit synchronisé nécessite que ces deux relations soient compatibles. En restant à ce niveau, nous sommes donc contraints d'utiliser l'intersection des deux relations d'équivalence (leur PGCD).

Prenons pour exemple la propriété LTL suivante : $\varphi = \mathsf{G}(a \Rightarrow b)$, où $a = R(\langle C_1, m_1 \rangle)$ et $b = C_1(2)$. La formule φ exprime le fait que dans toute exécution du système de la figure 3, si le réseau R contient un message du client C_1 de type m_1 (noté a) alors C_1 attend un acquittement (noté b).

La relation d'équivalence $\mathscr{R}$ commune au système et à φ est donc calculée à partir de l'intersection de $\mathscr{R}_M$ (la relation d'équivalence du système) et des relations d'équivalence déduites de chaque propriété intervenant dans la propriété φ. La propriété a a comme relation d'équivalence $\mathscr{R}_a : C = \{C_1\} \cup \{C_2\}, M = \{m_1\} \cup \{m_2\}$ car a distingue C_1 et m_1 dans sa définition. La relation d'équivalence associée à b est $\mathscr{R}_b : C = \{C_1\} \cup \{C_2\}, M = \{m_1, m_2\}$ car les messages n'interviennent pas dans la description de b.

La relation globale est $\mathscr{R} = \mathscr{R}_M \cap \mathscr{R}_a \cap \mathscr{R}_b$ c'est-à-dire $\mathscr{R} : C = \{C_1\} \cup \{C_2\}, M = \{m_1\} \cup \{m_2\}$. Il s'agit d'une *relation identité*, car aucune permutation d'éléments n'est admise par $\mathscr{R}$. Ceci correspond au pire des cas, où le graphe quotient produit sous $\mathscr{R}$ est égal au graphe d'états classique : les classes d'équivalence du graphe quotient ne représentent chacune qu'un unique état.

Avec la méthode SSP (Symbolic Synchronized Product [1, 2]) nous repoussons le problème : au lieu de chercher à réduire l'espace d'états (l'automate A_M de la figure 1), nous réduisons directement le produit synchronisé. L'intérêt est qu'au sein du produit synchronisé la formule est vue comme un automate où la formule propositionnelle sur chaque transition engendre une relation d'équivalence différente. Le SSP tire parti des différents degrés de permissivité de chacune de ces relations lors des synchronisations. Par exemple, en se synchronisant avec une transition dont la formule est b, seuls C_1 et C_2 sont distingués par la relation d'équivalence $\mathscr{R}_b$. Elle constitue donc une relation plus permissive que $\mathscr{R}_a$ (qui distingue tout objet du système). Par conséquent, l'ensemble considéré des états du système vérifiant b, sera représenté par un minimum de classes d'équivalence.

SSP est défini comme un graphe orienté, dont chaque nœud est une paire constituée d'un état q de l'automate de la formule à vérifier et d'un ensemble d'états s du système vérifiant la formule propositionnelle d'une transition amenant à q, et formant une classe d'équivalence par rapport à une relation d'équivalence calculée localement.

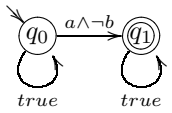

FIG. 7 – $A_{\neg\varphi}$ pour $\varphi = \mathsf{G}(a \Rightarrow b)$.

Considérons l'automate $A_{\neg\varphi}$ de la négation de la formule φ (Fig. 7). À partir de cet automate et du graphe quotient de la figure 6 nous allons construire la partie correspondante du SSP.

L'état initial du SSP est $\langle s''_0, q_0 \rangle$, le produit de l'état initial du système s''_0 et de l'automate q_0. L'étape suivante consiste à chercher les successeurs de s''_0 qui se synchronisent avec ceux de q_0. Le successeur s''_1 de s''_0 satisfait $true$ et aucune relation d'équivalence n'est requise pour le tester (à part la relation la plus permissive $\mathscr{R}_{max}$ permettant de construire les classes d'équivalence du système). Par contre, pour vérifier la formule $a \wedge \neg b$, il faut injecter $\mathscr{R}_a$ et $\mathscr{R}_b$ sur la classe s''_1. Ceci se traduit par le développement de tous ses états : s_1, s_2, s_3 et s_4 (Fig. 4), sur lesquels est testée la validité de la formule $a \wedge \neg b$. Comme aucun de ces états ne la vérifie, on retient $\langle s''_1, q_0 \rangle$ comme unique successeur de $\langle s''_0, q_0 \rangle$. De même pour les successeurs

de s''_1 : les états s''_2, s''_3 et s''_4 se synchronisent avec $true$ mais aucun des états s_5 à s_{10} représentés par les classes s''_2, s''_3 et s''_4, ne satisfait $a \wedge \neg b$. Par conséquent, les successeurs de $\langle s''_1, q_0 \rangle$ sont $\langle s''_2, q_0 \rangle$, $\langle s''_3, q_0 \rangle$ et $\langle s''_4, q_0 \rangle$. Ce processus se répète pour tous les successeurs construits qui continuent à se synchroniser avec la propriété $true$ car la formule $a \wedge \neg b$ n'est jamais satisfaite. On boucle donc sur q_0 avec la transition $true$ jusqu'à l'exploration de tous les états du système en n'utilisant que la relation d'équivalence la plus permissive $\mathscr{R}_{max}$. La formule φ est donc bien vérifiée, car on ne trouve pas de contre-exemple. Le SSP en sa globalité est construit et sa taille est la même que celle du graphe quotient sous $\mathscr{R}_{max}$ (en terme de nombre de nœuds), qui est la taille minimale espérée.

Ainsi l'approche SSP permet d'accomplir une forte réduction par rapport à une vérification basée sur le graphe quotient, même en présence d'une formule qui distingue un maximum d'éléments du système.

5 Outils et exemple

Nous avons mis en œuvre ces méthodes à l'aide des outils GreatSPN[1] et Spot[2]. GreatSPN nous permet de générer l'espace d'états d'un modèle exprimé par un réseau de Petri, nous l'avons équipé des deux réductions précédentes. Spot est une bibliothèque de *model checking* offrant les autres algorithmes nécessaires à l'approche automate. Combinés dans l'environnement CPN-AMI[3], ces outils nous permettent de vérifier une formule LTL sur un réseau de Petri [3]. Nous utilisons cette notation formelle comme langage de spécification pour produire l'espace d'états du système. Son principal intérêt est de permettre le calcul des symétries du système et donc d'automatiser l'exploitation de l'espace des états symboliques.

Ces outils ont été expérimentés dans le cadre d'une étude complexe : la vérification du cœur de l'intergiciel (ou *middleware*) PolyORB [4] [9]. L'architecture de cet intergiciel présente l'avantage d'être particulièrement modulaire : il est construit autour d'un *cœur neutre* qui implémente les fonctions clefs d'un intergiciel indépendamment des spécifications d'intergiciels existants. Des *personnalités* qui sont des composants décrivant soit un protocole (par exemple GIOP ou SOAP) soit une API d'intergiciel « classique » (par exemple CORBA) sont construites à partir des services canoniques de ce cœur neutre. Ce dernier est reconfigurable en fonction des besoins applicatifs (typiquement, le choix d'une politique de gestion des threads alloués au cœur neutre).

Pour assurer la vérification du cœur neutre, nous avons séparé les éléments actifs (comportements gérés sous forme de threads) des éléments passifs (des bibliothèques de sous-programmes assurant des fonctions comme l'identification du prochain thread à utiliser pour traiter un événement). Ainsi, on peut réduire les primitives des bibliothèques à des fonctions rendant systématiquement un résultat facile à modéliser, et vérifiable par une analyse statique de code (qui est séquentiel). Nous avons défini la partie active à l'aide d'un patron de conception appelé μBroker [4]. Ce patron décrit le flot d'exécution d'une requête, en ne considérant que les ressources utiles : tâches, sources, buffers. Les autres fonctions de l'intergiciel rendant possible le transfert et l'exécution de la requête (protocole, gestion du code utilisateur) sont déplacés dans des composants externes.

Le μBroker peut être adapté à une politique particulière. Pour cette étude, nous considérons le modèle « Leader/Followers » [5], un gestionnaire de sources d'où proviennent des requêtes, un thread « leader » qui traite les événements provenant de ces sources, une FIFO qui stocke les requêtes à traiter et un ensemble de threads « follower » qui assurent leur traitement. La vérification du μBroker utilise les techniques présentées dans cet article.

Les mesures faites sur ce modèle ont servi de benchmark pour nos outils de *model checking*. La spécification du μBroker s'appuie sur plusieurs paramètres, nous permettant d'étudier différentes configurations. Les deux paramètres que nous considérons dans l'analyse des gains de nos outils sont : S_{max},

[1] `http://www.di.unito.it/~greatspn/`
[2] `http://spot.lip6.fr/`
[3] `http://www.lip6.fr/cpn-ami`
[4] PolyORB est un projet commun LIP6-SRC/ENST comme indiqué sur `http://libre.act-europe.fr/polyorb`

qui correspond au nombre de sources surveillées par le μBroker et $Threads$, le nombre de threads alloués au μBroker. Les mesures présentées dans les figures 8 et 9 illustrent l'intérêt de ces méthodes. Nous nous sommes d'abord intéressé à la seule génération de l'espace d'états, afin de s'assurer qu'il n'existe pas d'état bloquant dans le système.

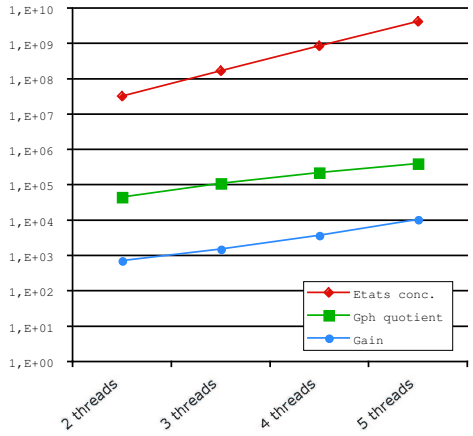

FIG. 8 – Évolution de l'espace d'états concret et du graphe quotient dans la modélisation du μBroker.

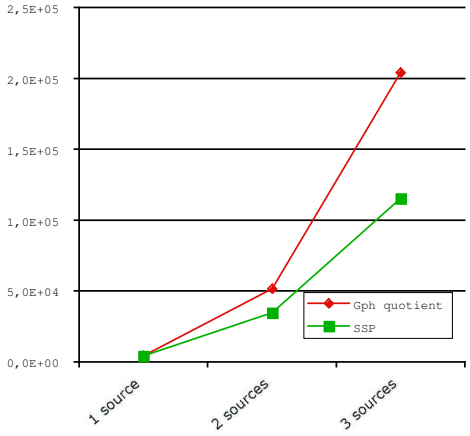

FIG. 9 – Nombre d'états construits pour vérifier la propriété P (graphe quotient et SSP).

La figure 8 présente le nombre d'états possibles du μBroker lorsque l'on fait varier $Threads$ (avec S_{max} fixé à 4 sources). Sur cette courbe logarithmique, on observe que la croissance de l'espace des états concrets est exponentielle. La taille du graphe quotient se situe un ordre de grandeur en dessous. Le plus intéressant est que le gain (rapport des deux tailles) est lui-même exponentiel. Un graphe quotient correspondant à plus de 4 milliards d'états concrets a pu être représenté en mémoire sur une machine disposant de 512Mo de RAM.

Afin de comparer les deux méthodes présentées dans le cadre du *model checking*, nous avons vérifié une propriété P qui spécifie qu'un événement e provenant d'une source particulière s sera forcément traité par un thread t. La distinction de s introduit une asymétrie qui sera traitée globalement par le graphe quotient, et localement par le SSP. La figure 9, où l'on fait varier S_{max} (avec $Threads$ fixé à 3 threads), montre la supériorité de SSP sur le graphe quotient puisque moitié moins de nœuds sont nécessaires à la vérification de P.

Alors que des outils plus classiques ne le permettaient pas (pour cause d'explosion combinatoire), les techniques de *model checking* présentées dans cet article nous ont permis de valider la spécification du μBroker de PolyORB.

6 Conclusion

Nous avons présenté dans cet article l'une des dernières techniques de vérification par *model checking*. Cette technique s'appuie sur le graphe quotient des états d'un système.

Bien que récente, cette théorie est implémentée dans des outils qui se sont montrés capables d'analyser des spécifications de systèmes réels. Bien que prototype, ces outils sont déjà utilisés dans plusieurs coopérations institutionnelles et dans le cadre du projet RNTL MORSE qui s'intéresse à la conception et l'analyse de systèmes répartis critiques.

Les gains que nous avons mesurés montrent clairement un ordre de grandeur supérieur dans la complexité des systèmes analysés, puisque le facteur de gain croît exponentiellement. Cela ouvre des perspectives intéressantes dans un futur proche.

Références

[1] Khalil Ajami, Serge Haddad, and Jean-Michel Ilié. Exploiting symmetry in linear temporal model checking : one step beyond. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), part of Theory and practice of Software (ETAPS'98)*, volume 1384 of *LNCS*, pages 52–67, Lisbon, Portugal, April 1998. Springer-Verlag.

[2] Soheib Baarir, Serge Haddad, and Jean-Michel Ilié. Exploiting partial symmetries in well-formed nets for the reachability and the linear time model checking problems. In *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, September 2004. À paraître.

[3] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. On well-formed coloured nets and their symbolic reachability graph. In Kurt Jensen and Grzegorz Rozenberg, editors, *Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application.* Springer-Verlag, 1991.

[4] Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Refining middleware functions for verification purpose. In *Monterey Workshop on Software Engineering for Embedded Systems : From Requirements to Implementation*, Chicago, Illinois, USA, September 2003.

[5] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C. Schmidt. Evaluating and optimizing thread pool strategies for real-time CORBA. In *Proceedings of the ACM SIGPLAN workshop on Optimization of Middleware and Distributed Systems (OM'01)*, pages 214–222, Snowbird, Utah, USA, June 2001. ACM.

[6] Yann Thierry-Mieg, Claude Dutheillet, and Isabelle Mounier. Automatic symmetry detection in well-formed nets. In Wil van der Aalst and Eike Best, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN'03)*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101, Eindhoven, The Netherlands, June 2003. Springer Verlag.

[7] Antti Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets 1 : Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[8] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.

[9] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB : a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, Palma de Mallorca, Spain, June 2004. À paraître.