EVALUATION PARAMETREE DE FORMULES DE LOGIQUE TEMPORELLE

Serge Haddad, Isabelle Vernier
Laboratoire MASI, Institut Blaise Pascal
Université Pierre et Marie Curie - CNRS
4, place Jussieu, 75252 Paris Cedex 05
E-mail: haddad@masi.ibp.fr, vernier@masi.ibp.fr

1 Introduction

La logique temporelle a obtenu un rapide succès comme méthode de vérification de programmes parallèles. Ce succès est dû à plusieurs facteurs: les propriétés usuelles des programmes parallèles s'expriment à l'aide de formules simples de logique temporelle, les propriétés de programmes à états finis sont vérifiées grâce à une exploration efficace du graphe d'état et d'une spécification donnée sous la forme d'un ensemble de formules de logique temporelle on peut obtenir la synthèse d'un programme respectant cette spécification.
Cependant, les algorithmes parallèles intègrent habituellement un nombre inconnu mais fini de processus. Par conséquent, les méthodes habituelles de vérification ne sont plus applicables dans le cadre d'une validation paramétrée car elles exigent de fixer le nombre de processus. Différentes approches ont été développées pour résoudre ce problème.
Dans [Clarke87], l'idée de base est de trouver le nombre minimal de processus représentant le comportement général de l'algorithme. Les méthodes habituelles de vérification sont ensuite appliquées avec ce nombre de processus. Cependant, la recherche du nombre minimal de processus n'est que partiellement automatisée. Dans [Sistla87], une restriction du modèle ou du langage des formules permet d'obtenir une vérification automatique. Dans [Wolper89], une autre approche est proposée: les propriétés sont déduites de l'existence d'invariants dont la vérification ne dépend pas du nombre de processus.

Dans cet article, nous présentons un modèle d'algorithmes parallèles paramétrés proche de celui de [Clarke87]. Ce modèle est composé de deux automates synchronisés dont l'un représente le comportement des processus et l'autre celui du contrôle. La sémantique du modèle est donnée pour n'importe quel nombre de processus. Nous construisons directement un graphe paramétré dont chaque état représente une famille d'états d'un graphe ordinaire. Cette procédure se termine avec succès pour un sous-ensemble du modèle représentant un grand nombre d'algorithmes. Lors d'une terminaison avec succès, la vérification d'une formule de CTL [Clarke86] est alors décidable. De plus, ce graphe est une représentation de haut niveau du comportement de l'algorithme et facilite sa compréhension. Le graphe paramétré permet de définir immédiatement l'ensemble des états accessibles pour n'importe quel nombre de processus suffisamment grand.

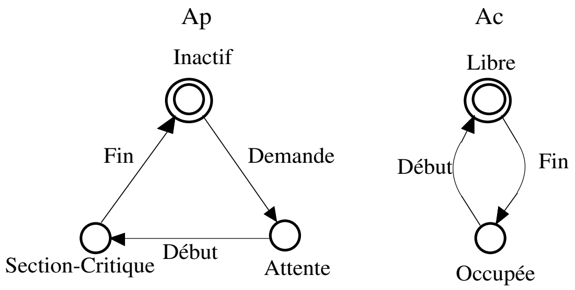
Dans la section 2, nous décrivons le modèle utilisé pour les programmes parallèles paramétrés. Dans la section 3, nous présentons l'algorithme permettant de construire le graphe d'accessibilité paramétré. La section 4 présente l'algorithme d'évaluation des formules de logique temporelle. Nous concluons dans la section 5 et présentons les futures extensions de notre travail.

2 Modèle générique pour les programmes parallèles paramétrés

Beaucoup d'algorithmes parallèles assurant les sections critiques, les terminaisons distribuées, … définissent le comportement identique d'un nombre indéfini de processus communiquant par une structure de contrôle (variables partagées, échanges de messages, …). Le comportement de ces algorithmes doit être correct quel que soit le nombre de processus. Nous définissons un modèle qui permet d'établir une preuve formelle indépendamment du nombre de processus.

Le modèle que nous définissons est appelé MPPP (Modèle pour les Programmes Parallèles Paramétrés). Il est composé de deux automates: le premier représente le comportement des processus (A_p) et le second celui de la structure de contrôle (A_c) qui autorise l'accès aux ressources ou la communication entre processus. En réalité un MPPP est une famille de programmes parallèles indicée par le nombre de processus. On note P le programme définit par le couple <A_p, A_c>. Nous présentons ce modèle à l'aide d'un exemple.

Considérons le problème d'accès à une section critique. Les automates des processus et du contrôle sont les suivants:



Les états initiaux sont représentés par des doubles cercles et les actions de synchronisation sont identifiées par leur nom, identique dans les deux automates. Ici, il s'agit des transitions de l'ensemble {Début, Fin}. Initialement chaque processus est dans l'état Inactif. Il peut alors soumettre une requête (Demande) et attendre (Attente) jusqu'à ce que le contrôle l'autorise à commencer sa section critique (Début). A la fin de la section critique, le contrôle peut accepter de nouvelles demandes. Les ensembles d'états et de transitions sont notés S_p (S_c) et T_p (T_c). Les états initiaux sont notés $Init_p$ (Inactif) et $Init_c$ (Libre).

P[2] est l'instance de P pour 2 processus. L'état initial de P[2] est <Inactif, Inactif, Libre>. Si le deuxième processus exécute l'action Demande, on arrive alors à l'état <Inactif, Attente, Libre> et si ensuite il exécute l'action Début en synchronisation avec le contrôle, on arrive à l'état <Inactif, Section-Critique, Occupée>.

3 Graphe d'accessibilité paramétré

Comme les modèles que nous étudions sont paramétrés, nous devons construire un graphe d'accessibilité paramétré représentant tous les états accessibles et toutes les exécutions possibles de modèle. Cependant nous devons exclure les comportements propres à un nombre particulier de processus. L'algorithme de construction du graphe nous permet de déterminer la borne minimale (borne du programme) pour laquelle le graphe paramétré représente l'ensemble des états accessibles.

Nous définissons d'abord les états paramétrés qui représentent un ensemble d'états instanciés pour chaque P[n]. A chaque état de S_p nous associons un marquage et un opérateur relationnel "=" ou "≥". Si "=x" est associée à un état cela signifie qu'il y a exactement x processus dans cet état et si "≥x" lui est associé, cela signifie qu'il y a au moins x processus dans l'état. Les mêmes attributs sont associés aux états de S_c avec les restrictions découlant du fait qu'il n'y a qu'un seul processus de contrôle: seul l'opérateur "=" est utilisé, les marquages peuvent être 1 ou 0. Si le marquage associé à un état est 0, l'opérateur est "=". L'état paramétré initial est tel que le marquage 1 est associé aux états $Init_{p \text{ et } c}$ et celui associé aux autres états est 0. L'opérateur associé à $Init_p$ est "≥" et celui associé à $Init_c$ est "=". Pour l'exemple considéré on a comme marquage initial {Inactif≥1, Libre=1}. Un état d'un programme instancié est représenté par un état paramétré s'il respecte les conditions imposées à chaque état par l'opérateur et le marquage qui lui sont associés. Pour qu'un état de P[n] puisse être représenté par un état paramétré, il faut que n soit supérieur ou égal à la somme des marquages. Ce qui permet de définir la borne du programme.
La construction du graphe paramétré se fait en calculant les états paramétrés accessibles depuis l'état initial. Pour cela, il est nécessaire de définir une règle de succession entre deux états paramétrés. Une transition permet d'atteindre un état paramétré but depuis un état paramétré source si:

- les marquages sont suffisants dans l'état source (pour exécuter la transition Début, les marquages associés aux places Attente et Libre doivent être supérieurs ou égaux à 1),
- l'état source est obtenu par des augmentations et diminutions consistantes (après l'exécution de la transition Début, il doit y avoir un processus de moins dans l'état Attente et un de plus dans l'état Section-Critique, tandis que le contrôle passe de l'état Libre à l'état Occupée).

Lorsqu'un état s auquel est associé "≥1" doit être diminué, deux cas peuvent se produire :

- un seul processus était dans l'état s, dans l'état but le marquage 0 est associé à s. Ceci n'est possible que si l'opérateur "≥" est associé à un autre état de processus' dans l'état source. Sinon une valeur serait affectée au paramètre n,
- plus d'un processus se trouvait dans l'état s, alors dans l'état but "≥1" est associé à s.

Si les deux cas sont vérifiés simultanément, un état paramétré a deux états successeurs pour une même transition.

Une fois le marquage initial et la règle de liaison donnés, il est facile de construire un arbre (ou graphe) d'accessibilité. Cependant une telle structure est généralement infinie. En conséquence, il est nécessaire de prendre certaines décisions lors de la construction du graphe. La procédure permettant d'éviter un développement infini est similaire à celle de Karp et Miller [Karp69].

Dès que nous trouvons deux états paramétrés ps et ps' sur la même branche de l'arbre tels que, quel que' soit l'état de processus s, le marquage associé à s dans ps est inférieur ou égal à celui associé à s dans ps', trois cas peuvent arriver:

(a) ps=ps', nous arrêtons la construction de cette branche,

(b) ps≠ps' et la séquence de transitions qui mène de ps à ps' est identifiable à "un changement d'état de s à s' pour un processus", alors ps' est transformé en ps sauf pour l'opérateur associé à s' qui prend la valeur "≥". Si le marquage associé à s dans ps est 0 alors celui associé à s dans le nouveau ps' est 1,

(c) sinon nous arrêtons la construction de l'arbre. C'est le cas d'échec de l'algorithme.

Une séquence de transitions est "un changement d'état de s à s' pour un processus" si après son exécution:

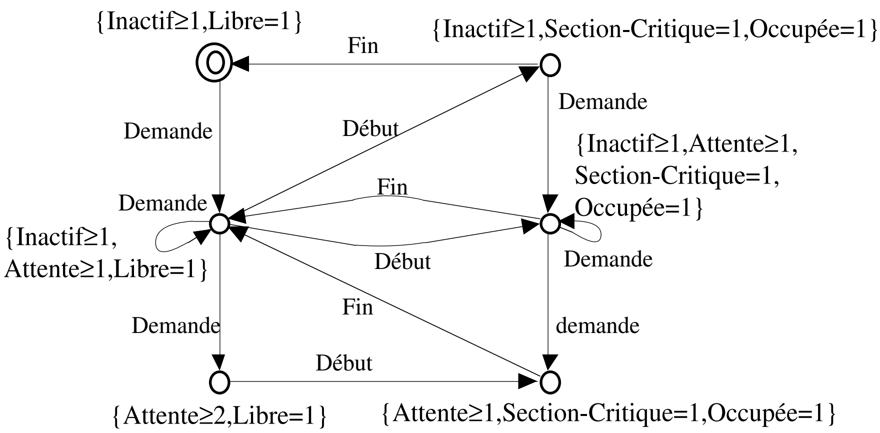
- un processus est passé de l'état s à l'état s',
- tous les autres processus n'ont pas changé d'état.

La séquence de transition peut donc être répétée, ce qui autorise la transformation du "=" en "≥".

En pratique, le cas (c) est rarement rencontré. Il correspond principalement aux cas où l'exécution d'un programme dépend de certaines propriétés du paramètre, e.g. si le comportement du programme est dépendant de la parité du nombre de processus. Dans ces cas, nous ne pouvons pas représenter, par un seul graphe, le comportement du programme quel que soit le nombre de processus.

La terminaison de notre algorithme est assurée par les propriétés d'ordre sur $\mathbb{N}^k$. Si la construction s'achève avec succès, nous identifions les nœuds identiques de l'arbre pour obtenir le graphe.

Le graphe suivant est le graphe paramétré du problème de l'exclusion mutuelle.



4 Evaluation de formules de logique temporelle sur le graphe

Nous avons choisi une logique arborescente car, comparée à une logique linéaire, elle est plus adaptée à la structure du graphe d'état et aux propriétés que nous souhaitions vérifier.

Bien que CTL soit d'une puissance d'expression moindre que d'autres langages (e.g. CTL*), nous l'avons choisi pour les raisons suivantes:

• la vérification d'une formule sur le graphe d'état a une complexité linéaire en fonction de la taille du graphe et de la formule,

• CTL est suffisant pour exprimer les propriétés usuelles telles que la vivacité et la sûreté.

Avant de pouvoir vérifier une formule, il est nécessaire d'augmenter le graphe avec de nouveaux circuits. En effet, les circuits du graphe paramétré ne sont pas obligatoirement des circuits "réels" (circuits qui apparaissent sur le graphe d'accessibilité d'une instance du programme). Par exemple, la boucle "Demande" autour du nœud {Inactif≥1, Attente≥1, Libre=1} n'est pas un circuit réel puisqu'il diminue le nombre de processus inactifs. Il est aussi possible que des circuits réels n'apparaissent pas dans le graphe paramétré. Toutefois, ils peuvent être calculés à partir des circuits initiaux en résolvant des équations entières positives [Ginsburg64]. Les nouveaux circuits sont ajoutés au graphe.

Une fois les circuits ajoutés, nous pouvons vérifier les formules de CTL. Notre algorithme de vérification est similaire mais différent de ceux appliqués lorsqu'il s'agit d'un simple automate. En effet, les algorithmes existant sont des algorithmes de recherche en profondeur d'abord. Ils sont basés sur le principe suivant: ils ne réitèrent pas les procédures d'évaluation sur des nœuds qui ont déjà été visités. Cette dernière propriété n'est pas vérifiée pour les nœuds du graphe paramétré. L'examen d'un nœud déjà visité peut permettre de détecter de nouveaux états instanciés qui vérifient la formule.

Pour résoudre ce problème, l'algorithme associe à chaque nœud une expression qui définit le sous-ensemble des états qui vérifient la formule. Lorsqu'un nœud est visité, le sous-ensemble est mis à jour. S'il est stable, l'algorithme ne poursuit pas l'examen à partir de ce nœud. L'expression associée à chaque nœud et à chaque sous-formule est un état paramétré. On montre que le nombre d'examens possibles d'un nœud est linéaire en fonction de la borne du programme.

5 Conclusion

Dans cet article, nous avons introduit un modèle permettant de considérer des programmes parallèles concernant un nombre fini mais inconnu de processus. Ce modèle est composé de deux automates synchronisés représentant le comportement des processus et du contrôle. Une sémantique est donnée à ce modèle pour n'importe quel nombre fixé de processus.

Nous avons ensuite développé un algorithme construisant une représentation paramétrée de l'ensemble d'accessibilité qui permet d'évaluer des formules du langage CTL.

En comparaison avec le travail présenté dans [Clarke87], nous définissons aussi le nombre suffisant de processus permettant d'obtenir le comportement global du programme paramétré. Clarke et Grümberg ne trouvent pas cette borne automatiquement et ne précisent pas comment déterminer si elle existe. Dans notre travail, nous la trouvons lors de la construction du graphe paramétré. Un autre intérêt de notre travail est que nous construisons le graphe des états accessibles pour n'importe quel nombre de processus supérieur ou égal à la borne et pas uniquement pour le nombre suffisant de processus.

Les apports futurs à ce travail doivent permettre de distinguer le comportement d'un processus pour pouvoir prouver des propriétés d'équité, et de prendre en compte des langages plus complexes.

Bibliographie

[Clarke86] Clarke E.M., Emerson E.A., Sistla J., *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification*, ACM Transactions on Programming languages and systems, vol 8, n°2, April 1986, pages 244-263.

[Clarke87] Clarke E.M., Grümberg O., *Avoiding The State Explosion Problem in Temporal Logic Model Checking Algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, August 1987, pages 294-303.

[Ginsburg64] Ginsburg S., Spanier E., *Bounded Algol - like languages*, Trans. Amer. Math. Soc. 113, 1964, pages 333-368.

[Karp69] R.M. Karp, R.E Miller, *Parallel Program Schemata*, Journal of Computer System science, vol. 3, n°2, May, 1969.

[Sistla87] Sistla A.P., German S.M., *Reasoning with Many Processes*, Proc. Symposium on Logic in Computer Science, Ithaca, New-York, June 1987, pages 138-152.

[Wolper89] Wolper P., Lovinfosse V.,*Verifying Properties of Large Sets of Processes with Network Invariants*, Automatic, Proc. International Workshop on automatic Verification Methods for Finite State Systems, Grenoble, France, 1989, LNCS 407, Springer Verlag, pages 68-80