Modèles formels pour les programmes récursifs-parallèles

O. Kouchnarenko, Ph. Schnoebelen [a]

[a]Institut IMAG, Laboratoire Leibniz
46, av. Félix Viallet, 38031, Grenoble, CEDEX, France
e-mail: {olgak,phs}@imag.fr

Résumé

Nous proposons un modèle formel du parallélisme inspiré par le langage récursif-parallèle RPC. Cette modélisation permet de valider la stratégie d'implémentation de RPC. Plus généralement, il s'agit d'un modèle original et nouveau présentant de bonnes propriétés de décidabilité.

1. Introduction

Nous proposons un modèle formel pour les programmes écrits dans le langage RPC, langage récursif-parallèle impératif, actuellement utilisé à l'Institut des Problèmes d'Informatique (IPI) de l'Académie des Sciences de Russie où un compilateur RPC a été développé pour une machine parallèle [8].

Ce modèle, décrit en détail dans [4], nous permet (1) de donner une sémantique au langage RPC, (2) de décrire formellement la stratégie sous-jacente à l'implémentation parallèle de RPC, (3) d'énoncer et de démontrer mathématiquement la correction de cette stratégie d'implémentation, (4) de développer des outils de vérification pour les programmes RPC [7].

Un programme RPC est un ensemble de procédures parallèles (ou *co-routines*) qui peuvent s'invoquer récursivement. Cf. exemple Fig. 1. Les spécificités du langage tiennent essentiellement à son choix de primitives pour la synchronisation. L'invocation (via `pcall`) d'une procédure donne lieu au déclenchement d'un processus *fils* parallèle, placé hiérarchiquement sous le contrôle du processus *père* qui l'a invoqué. Les processus fils s'exécutent librement mais leur père peut demander (via une instruction spéciale de synchronisation, le `wait`) d'attendre la terminaison de tous ses fils.

2. Schémas de programmes récursifs-parallèles

Les schémas des programmes [2, 3] formalisent la *structure de contrôle* de programmes en s'abstrayant des données et de l'effet des instructions de base. Nous considérons ici des schémas de programmes récursifs-parallèles basés sur les approches de Keller et de Podlovtchenko.

Soit une *base* $\Lambda = (A, B, C, \{\mathtt{wait}, \mathtt{end}\})$ d'ensembles (disjoints) de symboles où

- $A = \{a_1, a_2, \ldots\}$ est un alphabet (de noms) d'*actions*,
- $B = \{b_1, b_2, \ldots\}$ est un alphabet (de noms) de *tests*,
- $C = \{c_1, c_2, \ldots\}$ est un alphabet (de noms) d'*invocations*.

```
program main
      a1;
 l1:  pcall subr1 c1;
      a2;
      if b1 then goto l1;
      wait;
      a3;
      end;

procedure subr1
      if b2 then a4; end;
      pcall subr1 c2;
      a5;
      wait;
      end;
```

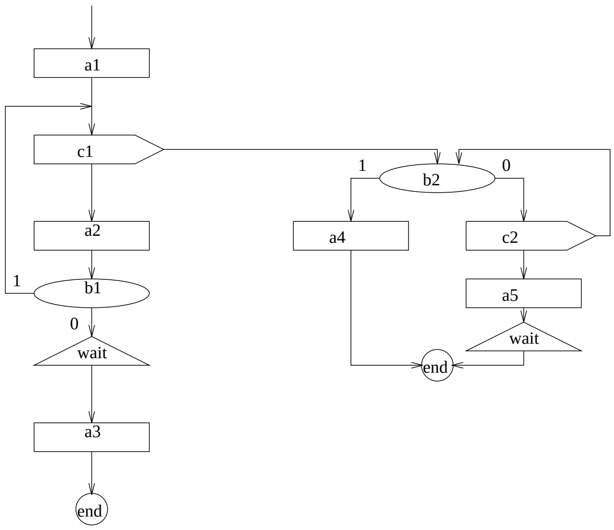

FIG. 1 - . Un programme RPC et le schéma associé

Définition 2.1 *Un* schéma de programme *(sur Λ) est un graphe $G = (Q, q_0, L)$ où*
-$Q = \{q_0, q_1, \ldots, q_n\}$ est un ensemble fini de sommets *(ou* noeuds*),*
-$q_0 \in Q$ est le noeud initial*,*
-L est un étiquetage *des noeuds du graphe tel que:*
$L\ : Q\ \rightarrow\ (A \times Q)\ \cup\ (B \times Q^2)\ \cup\ (C \times Q^2)\ \cup\ (\{\mathtt{wait}\} \times Q)\ \cup\ \{\mathtt{end}\}.$

Ainsi, L donne à la fois le symbole porté par un noeud du graphe et le ou les arcs (ordonnés) reliant ce noeud à d'autres noeuds. On note $RPSS_\Lambda$ l'ensemble des schémas. La Fig. 1 donne (sous une forme graphique) le schéma correspondant au programme RPC utilisé en exemple.

3. Sémantique d'états emboîtés

Le comportement des schémas de $RPPS_\Lambda$ utilise le concept d'*état emboîté.*

Définition 3.1 *M_G dénote l'ensemble des* états emboîtés *de $G \in RPPS_\Lambda$. Ce sont les multi-ensembles finis définis inductivement par:*
- si $q_1, \ldots, q_n \in Q_G$, $\xi_1, \ldots, \xi_n \in M_G$, alors $\xi \stackrel{def}{=} \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\} \in M_G$.

En particulier $\emptyset \in M_G$. L'état $\{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ correspond à l'activation de n processus indépendants. Chacun de ces processus (q_i, ξ_i) a une composante père dans l'état q_i et un ensemble ξ_i de sous-composantes filles parallèles, desquelles dépendrait l'exécutabilité d'une instruction `wait`.

À un schéma G on associe un système de transitions étiqueté [5] $\mathcal{M}_G = (M_G, \Lambda, \longmapsto, \xi_0)$, où $\xi_0 \stackrel{def}{=} (q_0, \emptyset)$ est l'*état initial* et où la *relation de transition* $\longmapsto\ \subseteq M_G \times \Lambda \times M_G$ est définie par des règles à la Plotkin [6]. Les règles correspondant à `pcall`, `wait` et `end` sont:

(R1)	Si $L(q) = \texttt{wait}, q'$ alors $(q, \xi) \overset{\texttt{wait}}{\longmapsto} (q', \xi)$ seulement si $\xi = \emptyset$.
(R2)	Si $L(q) = c, q', q''$ (avec $c \in C$) alors $(q, \xi) \overset{c}{\longmapsto} (q', (\xi + (q'', \emptyset)))$.
(R3)	Si $L(q) = \texttt{end}$ alors $(q, \xi) \overset{\texttt{end}}{\longmapsto} \xi$.

Elles montrent comment la structure hiérarhcique des états est créée et exploitée. Les règles omises sont assez naturelles: les composantes peuvent évoluer indépendamment les unes des autres. Nous renvoyons à [4] pour la définition complète. Ce modèle n'est pas à états-finis. Il est plus puissant que les Réseaux de Petri étiquetés (RdP) par la façon dont il utilise le `wait` et la hiérarchie des invocations; toutefois il ne contient pas la synchronisation générale des RdP. Il est néanmoins possible de le munir d'une structure de système de transitions bien structuré au sens de [1] ce qui établit la décidabilité de nombreuses questions: atteignabilité, terminaison, simulation d'un (resp. par un) automate fini, etc.

4. Modèle d'implémentation avec parallélisme borné

La stratégie utilisée dans [8] pour implémenter le langage RPC sur une architecture à m processeurs parallèles (et mémoire partagée) utilise, pour chaque processeur, une file d'attente (locale) et une *dequeue* (file à double entrée) pour les invocations parallèles en attente d'allocation. Cette stratégie peut être formalisée au moyen d'*états vectoriels*.

Définition 4.1 *Un* v-état *(de taille m) est une séquence $\pi = < \rho_1; \ldots; \rho_m >$ dans laquelle chaque composante ρ_j est de la forme ω_j, f_j, d_j où*
- $\omega_j \in Bloc \cup \{\perp\}$ est un bloc *(ou bien $\perp$),*
- $f_j \in Bloc^*$ est une file *de blocs,*
- $d_j \in Bloc^*$ est une dequeue *de blocs.*
Un bloc *a la forme $l : q, n, r$ où*
- $l \in \{l_1, \ldots\}$ est un tag*,*
- $q \in Q_G \cup \{\perp\}$ est un noeud de G (ou $\perp$)
- $n \in \mathbb{N}$ est un nombre de fils*,*
- $r \in \{l_1, \ldots\} \cup \{\perp\}$ est un tag de père *(ou bien $\perp$).*

Informellement, un bloc ω de la forme $l : q, n, r$ correspond à une invocation parallèle où q est l'instruction en cours d'exécution, n est le nombre de processus fils invoqués depuis ce processus, r repère le bloc correspondant au processus ayant invoqué ω, l est le tag de ω. On notera que ω ne connait que le nombre de ses fils, pas leur identité ni leur état.

Dans l'état π, le processeur j est en train d'exécuter ω_j et les blocs appartenant à f_j ou d_j sont en attente. Les blocs de f_j (gérés avec une discipline FIFO) sont des processus qui ont été interrompus (sur le processeur j) lors d'une instruction `wait`. Ils seront (peut-être) réactivés plus tard. Les blocs de d_j (file accessible par tous les processeurs) sont des invocations créées par les instructions `pcall`. Elles peuvent être exécutées par n'importe quel processeur.

Soit Π_G l'ensemble de tous les v-états. On définit un système de transitions étiqueté $\mathcal{P}_G = (\Pi_G, \Lambda_\tau, \mapsto, \pi_0)$ où $\mapsto \subseteq \Pi_G \times \Lambda_\tau \times \Pi_G$ est donné par des règles à la Plotkin permettant de décrire précisément et simplement la stratégie d'activation/déactivation et de synchronisation des processus (Cf. [4]) et où $\pi_0 = < (l_1 : q_0, 0, \perp), (), (); \ \perp; \ \ldots \perp >$ est le v-état *initial.*

Il est alors possible de relier formellement le comportement des deux modèles, et ainsi d'énoncer en quel sens précis l'implémentation de [8] est correcte. Le critère que nous utilisons est une notion de τ-simulation au sens de [5] préservant à la fois les propriétés de sûreté *et la non-terminaison*. Nous notons $\sqsubseteq_{\vee}$ ce préordre d'implémentation [4]. Il est alors possible de démontrer

Théorème 4.2 [4] $\mathcal{P}_G \sqsubseteq_{\vee} \mathcal{M}_G$.

L'intérêt de ce résultat ne se limite pas aux propriétés de préservation de $\sqsubseteq_{\vee}$. En effet, la même relation d'implémentation relie le comportement $\mathcal{P}_G$ des programmes et leur sémantique "interprétée". En effet, il est possible de développer notre modèle d'états emboîtés de façon à prendre en compte une interprétation des actions de base. Alors une action $a \in A$ aura un effet précis sur l'état-mémoire global [4]. Le mécanisme de synchronisation reste le même. On obtient les systèmes de transitions $\mathcal{M}_G^I$ et $\mathcal{P}_G^I$ et le théorème:

$$\mathcal{M}_G^I \sqsubseteq_{\vee} \mathcal{M}_G \text{ et } \mathcal{P}_G^I \sqsubseteq_{\vee} \mathcal{P}_G$$

correspondant principalement au fait que les instructions conditionnelles suivent une direction déterministe, dépendant de l'état-mémoire. Alors, on en déduit $\mathcal{P}_G^I \sqsubseteq_{\vee} \mathcal{M}_G$, ce qui permet d'affirmer que toute propriété de sûreté ou de terminaison démontrée pour $\mathcal{M}_G$ est vraie de la réalisation effective modélisée par $\mathcal{P}_G^I$.

5. Conclusion

Le modèle formel que nous avons développé pour les programmes RPC nous permet:
- de comprendre la discipline de synchronisation de RPC,
- de décrire la stratégie d'implémentation et d'énoncer en quel sens elle est correcte,
- d'expliquer ce que l'analyse des schémas nous dit sur les exécutions réelles,
- de mettre au point de nouvelles méthodes d'analyse,
- de valider la démarche d'analyse.

Bibliographie

1. A. Finkel:*Reduction and covering of infinite reachability trees*, Information and Computation, 1990, 89(2), p.144-179.
2. R.M.Karp, R.E.Miller:*Parallal Program Schemata*, JCSS, 1969, vol.3, N 2, p.147-195.
3. R.M.Keller:*Parallel Program Schemata and Maximal Parallelism*, JACM, 1973, vol. 20, N 3, p.514-537; vol. 20, N 4, p.696-710.
4. O.Kouchnarenko: *Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse*, Thèse de Doctorat, Univ. Grenoble, 1996 (à paraître).
5. R.Milner: *Communication and Concurrency*, Prentice Hall Int., 1989.
6. G.Plotkin: *A Structural Approach to Operational Semantics*, Daimi FN-19, Computer Science Department, Aarhus University, 1981.
7. V.Sokolov,O.Kouchnarenko: *Analysis of semantic properties of a class of recursive-parallel programs*, in: Proc. IV Conf. on Application Logic, Irkutsk, 1995, p.72-73.
8. V.Vasilchikov, V.Emielyn, V.Kurchidis, Y.Mamatov:*Recursive-parallel programming and work in RPMSHELL*, Preprint IPVT RAN , Yaroslavl, 1994, 80p.