Dérivation d'algorithmes sans verrou à partir d'une spécification atomique

Loïc Fejoz, Stephan Merz

Dérivation d'algorithmes sans verrou à partir d'une spécification atomique

Loïc Fejoz* et Stephan Merz

INRIA Lorraine & LORIA, Nancy, France
Loic.Fejoz@loria.fr Stephan.Merz@loria.fr

Résumé Pour gérer les accès de plusieurs processus à des données partagées, on utilise souvent un verrou global. Ici nous nous intéressons aux algorithmes sans verrou qui permettent un accès simultané en lecture et écriture. Malgré une littérature récente abondante, il y a peu de preuves de ces algorithmes.

Nous proposons une méthode modulaire qui permettra de dériver des algorithmes sans verrou à partir d'une spécification atomique qui décrit la fonctionnalité des opérations élémentaires sur une structure de données. Cette méthode peut être utilisée dans un style « à la B » (approche top-down), mais elle permet aussi d'ajouter des points de linéarisation aux algorithmes, en adaptant le style de la méthode « assume-guarantee ». Dans le présent article, on donne une formalisation de la méthode, et on explique son utilisation sur des exemples simplifiés. Nous comparons cette méthode à des méthodes plus classiques telles que B et TLA$^+$.

1 Introduction

Dans un environnement concurrent, l'approche la plus usitée pour gérer l'accès aux ressources partagées est l'utilisation d'un verrou global. Les algorithmes en jeu utilisent donc des primitives de synchronisation (mutexes, sémaphores) qui assurent un accès non-concurrent (*i.e.* en section critique). Malheureusement ceci amène de nombreux problèmes : l'utilisation d'un verrou global qui contrôle la structure entière empêche l'accès concurrent à des parties disjointes de la structure. L'introduction de verrous plus fins complique souvent la logique des applications et est en proie à des problème de blocages dûs à l'attente de la libération d'un ou plusieurs verrous.

Un exemple typique serait donné par deux processus qui s'échangent des messages par l'intermédiaire d'une liste, le premier ajoutant des messages en tête de liste et le second prélevant les messages en queue. Avec un verrou global, seul un des deux processus accède à la liste. L'utilisation d'un verrou plus fin (par exemple sur les deux premiers et les deux derniers éléments de la liste) ne résout pas le problème de famine dans le cas où l'un des processus venait à mourir en section critique. Seule une solution sans verrou assurerait l'indépendance des deux processus.

* This work was supported by Microsoft Research through its European PhD Scholarship Programme.

Les algorithmes sans verrou se basent sur des primitives atomiques fournies par les processeurs modernes. Ces primitives, dont la plus connue est le "Compare-and-swap" (CAS), n'ont pas toujours été disponibles sur toutes les plates-formes. La large diffusion de matériel supportant l'exécution parallèle explique le regain d'intérêt [4,5,11,14,15,16] pour les algorithmes sans verrou. Malgré la complexité de ces algorithmes, peu de travaux sont accompagnés de preuves formelles de correction. Nous nous proposons de définir une méthode spécifique pour la dérivation d'algorithmes sans verrou. En effet, ces algorithmes utilisent souvent des solutions types qui s'avèrent difficiles à formaliser dans des méthodes formelles généralistes telles que B, TLA$^+$ ou l'approche « rely-guarantee ». Notre méthode permettra de disposer d'une bibliothèque de solutions qui pourront être adaptées à un algorithme donné.

Le chapitre 2 introduit les propriétés de correction associées aux algorithmes sans verrou et présente la primitive CAS et ses dérivées. Notre méthode pour le développement conjoint d'un ensemble d'algorithmes opérant sur une même structure de données est décrite dans le chapitre 3. Nous l'appliquons sur des exemples simples mais illustratifs dans le chapitre 4. Enfin, le chapitre 5 montre en quoi notre méthode nous semble plus adaptée au problème que les méthodes TLA$^+$ [8], B [1] et « rely-guarantee » [3].

2 Algorithmes sans verrou

2.1 Propriétés désirables

Une structure de données est manipulée par plusieurs algorithmes. En effet, on en trouve autant que de méthodes supportées par la structure (dans le cas de la liste : ajout, suppression, insertion, lecture, etc). Ces algorithmes peuvent être exécutés de manière séquentielle ou parallèle par plusieurs processus (ou threads).

Afin de simplifier la compréhension des algorithmes par l'utilisateur, il est préférable que chacune des méthodes apparaisse à l'exécution comme étant atomique. Cette propriété est appelée *linéarisabilité* et s'énonce comme suit : une trace d'exécution concurrente est linéarisable s'il existe une trace séquentielle qui permet d'observer le même comportement extérieur. Il est donc intéressant d'avoir une méthode qui ait pour première spécification la description de l'effet atomique des méthodes.

Par ailleurs, les verrous (bloquants) sont très souvent remplacés par une boucle utilisant les primitives évoquées en introduction. Il se pose alors la question de la terminaison. Trois notions sont souvent évoquées dans la littérature.

1. **wait-free** : tous les processus progressent même s'il y a du délai ;
2. **lock-free** : certains processus progressent ;
3. **obstruction-free** : les processus exécutés isolément progressent.

Cependant, nous ne considérons pas ces notions de vivacité dans le présent article.

2.2 CAS

De nombreuses primitives atomiques ont été proposées et peuvent être exploitées pour la construction d'algorithmes sans verrou, comme Load-Link/Store-conditionnal (LL/SC), Test-and-Set (TAS), Fetch-and-Add et la famille des Compare-and-Swap (CAS). Nous ne les détaillerons pas toutes ici. En effet, toutes ne sont pas universelles comme l'a montré Herlihy [6]. Nous nous focaliserons sur la primitive CAS car elle est universelle, dans le sens où l'on peut construire les autres primitives à partir du CAS. La primitive de base, dénotée CAS_1, prend en paramètre un pointeur a et deux valeurs o, n. Si la valeur pointée par a est égal au deuxième argument o, on donne à la valeur pointée la valeur du troisième argument n, sinon, la valeur référencée par a n'est pas modifiée. Autrement dit, on exécute atomiquement la fonction suivante (décrite en C) :

```
word_t CAS1(word_t *a, word_t o, word_t n) {
  word_t old = *a;
  if (old == o) {
    *a = n;
  }
  return old;
}
```

Dans la suite, on utilisera une version modifiée appelée CAS_1. Elle diffère de CAS_1 par le fait qu'au lieu de retourner la valeur précédente du pointeur, elle retourne True si l'affectation a eu lieu, False sinon.

2.3 RDCSS

Harris et al. [5] ont montré comment la primitive CAS_1 peut servir de base à l'implémentation des primitives CAS opérant sur plusieurs valeurs à la fois ; ces variantes sont à la base d'algorithmes sans verrou sur des structures de données complexes. Une étape clef dans leur construction est la réalisation des opérations RDCSS (*restricted double compare single swap*) et RDCSSRead dont les spécifications atomiques sont les suivantes :

```
word_t RDCSS(word_t *a1, word_t o1, word_t *a2, word_t o2, word_t n2) {
  r = *a2;
  if ((r == o2) && (*a1 == o1)) {
    *a2 = n2;
  }
  return r;
}

word_t RDCSSRead(word_t *a2) {
  return *a2;
}
```


L'opération RDCSS prend en paramètre deux pointeurs a_1 et a_2 et trois valeurs o_1, o_2 et n_2. Si les valeurs référencées par a_1 et a_2 sont égales aux valeurs

```
word_t RDCSS(RDCSSDescriptor_t *d) {
  do {
    r = CAS1(d->a2, d->o2, d); // C1
    if (IsDescriptor(r)) Complete(r); // H1
  } while (IsDescriptor(r)); // B1
  if (r == d->o2) Complete(d);
  return r;
}

word_t RDCSSRead(addr_t *addr) {                                    10
  do {
    r = *addr; // R1
    if (IsDescriptor(r)) Complete(r); //H2
  } while (IsDescriptor(r)); // B2
  return r;
}

void Complete(RDCSSDescriptor_t *d) {
  v = *(d->a1); // R2
  if (v == d->o1) {                                                 20
    CAS1(d->a2,d,d->n2); // C2
  } else {
    CAS1(d->a2,d,d->o2); // C3
  }
}
```

Fig. 1. Algorithme implémentant RDCSS

o_1 et o_2, alors on donne à la valeur pointée par a_2 la valeur n. L'opération RDCSSRead renvoie simplement la valeur référencée par l'adresse donnée en argument.

L'implémentation de RDCSS et RDCSSRead proposée par Harris et al. apparaît dans la figure 1. Comme beaucoup de tels algorithmes, le code pour le RDCSS installe d'abord un descripteur qui contient tous les éléments nécessaires pour effectuer l'opération. Ce descripteur est installé à la place du résultat. Le premier processus opérant sur cette adresse de la mémoire effectue l'opération indiquée par le descripteur, avant de faire sa propre opération. (On suppose qu'un descripteur est différent de toute autre valeur.) Dans cette approche, un processus peut être amené à effectuer une opération à la place d'un autre processus. Le processus original verra une valeur différente de son descripteur, signalant que l'opération a déjà été effectuée.

Nous nous posons comme objectif de concevoir une méthode pour démontrer la correction d'algorithmes tels que celui de la figure 1. La méthode doit pouvoir être composable. C'est-à-dire permettre de réutiliser des spécifications effectuées

par ailleurs. Une fois outillée, elle doit pouvoir interagir avec des prouveurs automatiques ou, à défaut, interactifs.

3 Spécification et vérification d'algorithmes parallèles

Nous considérons un système constitué d'un ensemble arbitraire de processus, chacun exécutant un algorithme parmi un nombre fixe d'algorithmes accédant à la structure partagée. Chaque processus possède un espace mémoire propre (environnement local) et interagit avec les autres via une mémoire partagée (environnement global). Ceci nous amène donc à spécifier le système en décrivant les algorithmes executés par les processus. Pour chacun des algorithmes nous allons démontrer qu'il est équivalent, dans le contexte des autres algorithmes, à sa spécification atomique, afin d'assurer la linéarisabilité.

3.1 Formalisation des algorithmes

Un algorithme est modélisé comme un système de transitions ; les noeuds étant appelé places. Les transitions sont supposées être prises atomiquement. Une place est donc un point où des transitions d'autres algorithmes peuvent être effectuées (interleaving semantics). On associe alors à chaque place un invariant local. Celui-ci s'applique à la fois sur l'environnement local et global. Pour avoir une méthode compositionnelle, on associe aussi un prédicat indiquant ce que doivent satisfaire les actions exécutées en parallèle.

Definition 1. *Un contrôle de flot* $CF \equiv (Place, p_{init}, \Delta)$ *consiste en :*
- *un ensemble fini de places* $Place$*,*
- *une place initiale* p_{init} *appartenant à* $Place$ *et*
- *un ensemble de transitions* $\Delta \subseteq Place \times Place$ *sur les places.*

Un algorithme est décrit par le contrôle de flot sous-jacent et des prédicats qui spécifient l'invariant local et la condition de garantie associés à chaque place, et la garde et l'effet de chaque transition. Les environnements global et locaux associent des valeurs aux variables globales et locales ; pour simplifier l'exposition nous représentons tous les environnements par un même type E.

Definition 2. *Un algorithme* $A = (CF, local\_invariant, rely, guard, effect)$ *est décrit par un quintuplet où :*
- CF *est un contrôle de flot* CF*,*
- $local\_invariant : Place \to E \times E \to \mathbb{B}$ *associe un invariant local (portant sur un environnement global et local) à chaque place,*
- $rely : Place \to E \times E \to E \times E \to \mathbb{B}$ *décrit, pour chaque place, une condition de garantie entre les environnements locaux et globaux d'avant et après,*
- $guard : \Delta \to E \times E \to \mathbb{B}$ *décrit la garde pour chaque transition et*
- $effect : \Delta \to E \times E \to E \times E \to \mathbb{B}$ *décrit l'effet des transitions.*

Notons que pour encoder les algorithmes paramétrés (comme RDCSS avec a_1, a_2, o_1, o_2 et n_2), on peut inclure les paramètres dans l'environnement local.

Nous associons des obligations de preuve à un algorithme : toute transition, qu'elle soit due à l'algorithme lui-même ou à son environnement décrit par la condition *rely*, doit respecter les invariants associés aux places. Aussi, nous exigeons que toute transition soit possible à partir des états décrits par l'invariant associé à la place source.

Definition 3. *Un algorithme* $A = (CF, local\_invariant, rely, guard, effect)$ *est* valide *s'il vérifie les conditions suivantes.*

- *Chaque transition* $\delta = (p_1, p_2) \in \Delta$ *respecte l'invariant local : pour tous environnements* $genv_1$, $lenv_1$, $genv_2$, $lenv_2$ *l'implication*
  $$local\_invariant(p_1, genv_1, lenv1) \\ \wedge\ guard(\delta, genv_1, lenv_1) \wedge effect(\delta, genv_1, lenv_1, genv_2, lenv_2) \\ \Rightarrow local\_invariant(p_2, genv_2, lenv_2)$$
  est valide.
- *Chaque transition décrite par le prédicat rely respecte l'invariant local : pour toute place p et tous environnements* $genv_1$, $lenv_1$, $genv_2$, $lenv_2$ *l'implication*
  $$local\_invariant(p, genv_1, lenv1) \wedge rely(p, genv_1, lenv_1, genv_2, lenv_2) \\ \Rightarrow local\_invariant(p, genv_2, lenv_2)$$
  est valide.
- *Chaque transition* $\delta \in \Delta$ *est possible : pour tous environnements* $genv_1$ *et* $lenv_1$ *il existe des environnements* $genv_2$ *et* $lenv_2$ *tels que*
  $$local\_invariant(p_1, genv_1, lenv1) \wedge guard(\delta, genv_1, lenv_1) \\ \Rightarrow effect(\delta, genv_1, lenv_1, genv_2, lenv_2)$$
  est valide.

Le nombre de processus étant variable, il convient de faire la preuve en faisant abstraction de ce nombre. Une spécification est donc seulement constituée d'une liste d'algorithmes. Car ces algorithmes s'exécutent en parallèle, y compris plusieurs instances d'un même algorithme, nous exigeons que chaque transition d'un algorithme satisfasse les garanties *rely* de tous les algorithmes, y compris lui même.

Definition 4. *Une spécification est une liste d'algorithmes* $Spec = [A_1, \ldots, A_n]$.

Une spécification $Spec \equiv [A_1 \ldots A_n]$ *est valide si elle vérifie les conditions suivantes, où* $A_i = (CF_i, local\_invariant_i, rely_i, guard_i, effect_i)$.

- *Chaque algorithme* A_i *est valide et*
- *chaque transition d'un algorithme vérifie les conditions rely de tous les algorithmes. Pour tous* $i, j \in \{1, \ldots, n\}$, *toute place* $p_i \in Place_i$, *toute transition* $\delta_j = (p_j, p'_j) \in \Delta_j$ *et tous environnements* $genv$, $genv'$, $lenv_i$, $lenv_j$, $lenv'_j$ *:*
  $$local\_invariant_i(p_i, genv, lenv_i) \wedge local\_invariant_j(p_j, genv, lenv_j) \\ \wedge\ guard_j(\delta_j, genv, lenv_j) \wedge effect_j(\delta_j, genv, lenv_j, genv', lenv'_j) \\ \Rightarrow rely_i(p_i, genv, lenv_i, genv', lenv_i).$$

3.2 Raffinement

Comme l'on veut que chaque algorithme implémenté soit équivalent à un appel atomique d'une méthode, on va travailler par raffinement. En d'autres termes, on va écrire des spécifications intermédiaires qui introduisent de plus en plus de détails.

La spécification d'un algorithme atomique est composée de deux places et d'une transition, pour laquelle l'annotation *effect* explicite la relation pré/post-conditions. Souvent, les places ne contiendront pas d'invariants locaux ni de *rely*.

Formellement, nous définissons une relation de raffinement entre deux algorithmes $\underline{A}$ et $\overline{A}$ aux niveaux concret et abstrait. Pour faire le lien entre les places abstraites et les places concrètes, on utilise une relation de raffinement $mapp : \underline{Place} \times \overline{Place}$. Nous indiquerons cette relation de raffinement souvent par des flêches pointillées (voir par exemple les figures 2 et 3). La relation entre les environnements abstraits et concrets est décrite par un invariant de collage de la forme $gluing\_invariant : E \times E \to E \times E \to \mathbb{B}$ – afin de simplifier l'exposition, nous ne distinguons pas ici entre les environnements globaux et locaux.

Definition 5. *Un algorithme concret $\underline{A}$ raffine un algorithme abstrait $\overline{A}$ sous mapp et gluing_invariant si :*

- $(\underline{p}_{init}, \overline{p}_{init}) \in mapp$*, c'est à dire la place initiale concrète raffine la place initiale abstraite,*
- *chaque place concrète raffine une place abstraite : pour tout $\underline{p} \in \underline{Place}$ il existe $\overline{p} \in \overline{Place}$ avec $(\underline{p}, \overline{p}) \in mapp$,*
- *l'invariant local initial concret implique l'invariant local initial abstrait :*

$$\begin{aligned} & \underline{local\_invariant}(\underline{p}_{init}, \underline{genv}, \underline{lenv}) \\ & \wedge\ gluing\_invariant(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\ \Rightarrow\ & \overline{local\_invariant}(\overline{p}_{init}, \overline{genv}, \overline{lenv}), \end{aligned}$$

- *chaque pas concret correspond à un pas abstrait ou bien à un bégaiement : pour toute transition concrète $\underline{\delta} = (\underline{p}, \underline{p}') \in \underline{\Delta}$ et toute place abstraite $\overline{p} \in \overline{Place}$ avec $(\underline{p}, \overline{p}) \in mapp$, et pour tous environnements $\underline{genv}$, $\underline{lenv}$, $\underline{genv}'$, $\underline{lenv}'$, $\overline{genv}$, $\overline{lenv}$ il existe des environnements abstraits $\overline{genv}'$ et $\overline{lenv}'$ tels que*
  - *soit il existe une place abstraite $\overline{p}' \in \overline{Place}$ avec $(\underline{p}', \overline{p}') \in mapp$ et $\overline{\delta} = (\overline{p}, \overline{p}') \in \overline{\Delta}$ telle que*

$$\begin{aligned} & \underline{local\_invariant}(\underline{p}, \underline{genv}, \underline{lenv}) \\ & \wedge\ \underline{guard}(\underline{\delta}, \underline{genv}, \underline{lenv}) \wedge \underline{effect}(\underline{\delta}, \underline{genv}, \underline{lenv}, \underline{genv}', \underline{lenv}') \\ & \wedge\ \underline{local\_invariant}(\underline{p}', \underline{genv}', \underline{lenv}') \\ & \wedge\ gluing\_invariant(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\ \Rightarrow\ & gluing\_invariant(\underline{genv}', \underline{lenv}', \overline{genv}', \overline{lenv}') \\ & \wedge\ \overline{local\_invariant}(\overline{p}', \overline{genv}', \overline{lenv}') \\ & \wedge\ \overline{guard}(\overline{\delta}, \overline{genv}, \overline{lenv}) \wedge \overline{effect}(\overline{\delta}, \overline{genv}, \overline{lenv}, \overline{genv}', \overline{lenv}'), \end{aligned}$$

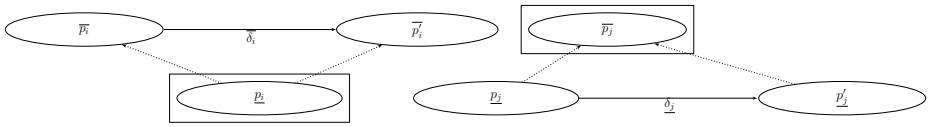

Fig. 2. $\underline{A_j}$ effectue un pas abstrait de $\overline{A_i}$.

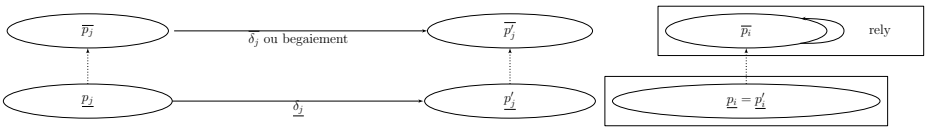

Fig. 3. $\underline{A_j}$ n'a pas d'effet $\overline{A_i}$.

– *soit la transition est invisible au niveau abstrait :*

$$
\begin{aligned}
& \underline{local\_invariant}(\underline{p}, \underline{genv}, \underline{lenv}) \\
& \wedge\ \underline{guard}(\underline{\delta}, \underline{genv}, \underline{lenv}) \wedge \underline{effect}(\underline{\delta}, \underline{genv}, \underline{lenv}, \underline{genv}', \underline{lenv}') \\
& \wedge\ \underline{local\_invariant}(\underline{p}', \underline{genv}' \underline{lenv}') \\
& \wedge\ \underline{gluing\_invariant}(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\
\Rightarrow\ & \underline{gluing\_invariant}(\underline{genv}', \underline{lenv}', \overline{genv}', \overline{lenv}') \\
& \wedge\ \overline{local\_invariant}(\overline{p}', \overline{genv}', \overline{lenv}') \\
& \wedge\ \overline{rely}(\overline{genv}, \overline{lenv}, \overline{genv}', \overline{lenv}')
\end{aligned}
$$

Pour définir le raffinement entre spécifications, il faut prendre en compte les relations entre algorithmes différents. Or, comme expliqué dans le paragraphe 2.3 pour l'algorithme RDCSS, un algorithme peut effectuer un pas abstrait d'un autre. Il faut évidement que quand l'algorithme original reprend pour finir son action (qui a déjà été effectuée par un autre) l'action ne doit avoir aucun effet. On autorise donc un algorithme concret $\underline{A_j}$ à effectuer un pas d'un algorithme abstrait $\overline{A_i}$ dont il n'est pas un raffinement. Les deux situations possibles sont illustrées sur les figures 2 et 3. C'est à dire que soit le pas concret $\underline{\delta_j}$ est un bégaiement pour $\overline{A_j}$ et peut correspondre à un pas $\overline{\delta_i}$, soit $\underline{\delta_j}$ correspond à $\overline{\delta_j}$ et donc garantit la condition $\overline{rely_i}$ de la place considérée. Finalement, on obtient la définition suivante d'un raffinement valide.

Definition 6. *Un raffinement entre les spécifications* $\underline{Spec} = [\underline{A}_1, \ldots, \underline{A}_n]$ *et* $\overline{Spec} = [\underline{A}_1, \ldots, \underline{A}_m]$ *sous mapp et gluing_invariant est dit valide ssi :*

– $\underline{Spec}$ *et* $\overline{Spec}$ *sont valides,*
– $n = m$, i.e. $\underline{Spec}$ *et* $\overline{Spec}$ *sont composés du même nombre d'algorithmes,*
– *Pour tout* $i \in \{1, \ldots, n\}$, *l'algorithme concret* $\underline{A_i}$ *raffine l'algorithme abstrait* $\overline{A_i}$
– *Un algorithme concret* $\underline{A_j}$ *peut aussi effectuer un pas abstrait d'un autre algorithme* $\overline{A_i}$ *: pour tous* $i, j \in \{1, \ldots, n\}$, *transitions* $\underline{\delta_j} = (\underline{p_j}, \underline{pc'_j})$, *places*

$\underline{p}_i$, $\overline{p}_i$, $\overline{p}_j$ *avec* $(\underline{p}_j, \overline{p}_j) \in$ *mapp et* $(\underline{p}_i, \overline{p}_i) \in$ *mapp et tous environnements* $\underline{genv}$, $\underline{genv'}$, $\underline{lenv}_j$, $\underline{lenv'}_j$, $\underline{lenv}_i$, $\overline{genv}$, $\overline{lenv}_j$ *et* $\overline{lenv}_i$ *il existe des places* $\overline{p}'_j$ *et* $\overline{p}'_i$ *avec* $(\underline{p}'_j, \overline{p}'_j) \in$ *mapp et* $(\underline{p}'_i, \overline{p}'_i) \in$ *mapp, ainsi que des environnements* $\overline{genv}'$, $\overline{lenv}'_j$ *et* $\overline{lenv}'_i$ *avec*

$$
\begin{array}{ll}
 & \mathit{gluing\_invariant}(\underline{genv}, \underline{lenv}_j, \overline{genv}, \underline{lenv}_j) \\
 & \wedge\ \mathit{gluing\_invariant}(\underline{genv}, \underline{lenv}_i, \overline{genv}, \underline{lenv}_i) \\
 & \wedge\ \mathit{local\_invariant}_j(\underline{p}_j, \underline{genv}, \underline{lenv}_j) \wedge \mathit{local\_invariant}_i(\underline{p}_i, \underline{genv}, \underline{lenv}_i) \\
 & \wedge\ \underline{\mathit{guard}}_j(\underline{\delta}_j, \underline{genv}, \underline{lenv}_j) \wedge \underline{\mathit{effect}}_j(\underline{\delta}_j, \underline{genv}, \underline{lenv}_j, \underline{genv}', \underline{lenv}'_j) \\
\Rightarrow & \overline{p}'_i = \overline{p}_i \wedge ((\overline{p}_j, \overline{p}'_j) \in \overline{\Delta}_j \vee \overline{p}'_j = \overline{p}_j) \\
 & \wedge\ \overline{lenv}'_i = \overline{lenv}_i \wedge \overline{rely}_i(\overline{p}_i, \overline{genv}, \overline{lenv}_i, \overline{genv}', \overline{lenv}'_i) \\
 & \vee\ (\overline{p}_i, \overline{p}'_i) \in \overline{\Delta}_i \wedge \overline{p}'_j = \overline{p}_j \\
 & \overline{\mathit{guard}}_i((\overline{p}_i, \overline{p}'_i), \overline{genv}, \overline{lenv}_i) \wedge \overline{\mathit{effect}}_i((\overline{p}_i, \overline{p}'_i), \overline{genv}, \overline{lenv}_i, \overline{genv}', \overline{lenv}'_i)
\end{array}
$$

Ces conditions de raffinement nous permettent de reconstruire une trace abstraite de $\overline{Spec}$ pour toute trace concrète de $\underline{Spec}$ de telle manière que les places concrètes et abstraites soient toujours liées par la relation *mapp* et que les environnements soient liés par *gluing_invariant*.

Theorem 1. *Soient mapp, gluing_invariant,* $\underline{Spec}$ *et* $\overline{Spec}$ *tels que* $\underline{Spec}$ *raffine* $\overline{Spec}$ *sous mapp et gluing_invariant, alors pour toute trace concrète de* $\underline{Spec}$ *il existe une trace abstraite de* $\overline{Spec}$ *reliée par mapp et gluing_invariant.*

La reconstitution de la trace abstraite se fait par induction. En effet, pour chaque transition concrète $\underline{\delta}_j$, on cherche à voir si elle effectue une transition $\overline{\delta}_j$ ou alors une transition $\overline{\delta}_i$ d'un autre algorithme abstrait. Dans le deuxième cas la transition est permise car $\overline{\delta}_i$ respecte la condition $\overline{rely}_j$ et on peut construire la trace abstraite correspondante. Cela nous indique au passage que la place $\underline{p}_i$ est aussi un raffinement de $\overline{p}'_i$ donc lorsque la transition $\underline{\delta}_i$ raffinant naturellement la transition abstraite $\overline{\delta}_i$ sera prise, il existera bien un environnement abstrait suivant satisfaisant la condition $\overline{rely}_i$.

Ces définitions et ce théorème nous permettent donc d'assurer la correction de la méthode. Dans le cas d'une spécification abstraite atomique elle nous assure la linéarisabilité de l'implémentation.

4 Exemples

Le premier exemple illustre un raffinement très courant. Il a été developpé dans l'assistant à la preuve Isabelle/HOL [10]. Le deuxième illustre la possibilité pour un algorithme d'effectuer le pas abstrait d'un autre. Pour cet exemple, on a écrit un générateur d'obligations de preuve qui ont ensuite été prouvés automatiquement par Isabelle (en utilisant la tactique auto). Il est prévu par la suite de soumettre ces obligations à un prouveur automatique.

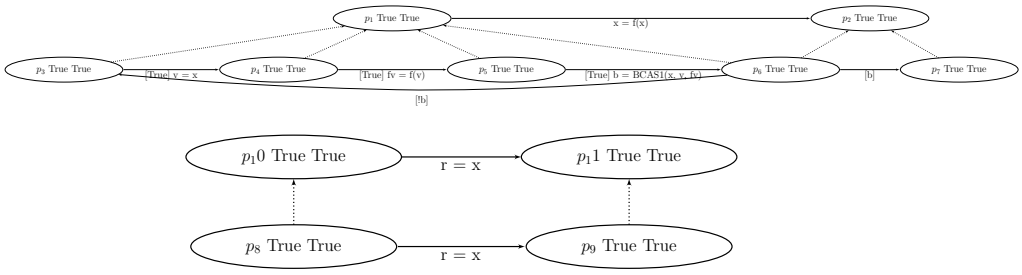

Fig. 4. Raffinement des algorithmes d'affectation.

4.1 Premier exemple : affectation

Pour l'exemple, on va appliquer la méthode sur un autre schéma très utilisé. Le premier algorithme effectue la mise à jour

```
x = f(x);
```

pour une fonction `f` non spécifiée. Le deuxième algorithme accède à la variable `x` en lecture. Comme la fonction `f` peut être complexe et la variable `x` est partagée entre plusieurs processus, on ne peut pas se contenter dans une implémentation de lire `x`, calculer `f(x)`, puis changer `x`. On va alors effectuer le calcul en local et utiliser la commande CAS_1 pour effectuer l'affectation de manière atomique :

```
b = false;
do {
  v = x;
  fv = f(v);
  b = BCAS1(x, v, fv);
} while(!b);
```

Le deuxième algorithme reste inchangé dans le raffinement. Les relations de raffinement entre ces algorithmes concrets et abstraits apparaissent dans la figure 4, et il est facile à vérifier que toutes les conditions des définitions 5 et 6 sont bien vérifiées.

4.2 Autre exemple

Notre deuxième exemple illustre le fait qu'un algorithme puisse effectuer un pas d'un autre. En effet, l'algorithme 2 (voir figure 6) attend une condition a qui est positionnée par l'algorithme 1 (voir figure 5). Dans l'algorithme concret, l'algorithme 1 commence par annoncer qu'il va effectuer l'action en positionnant b. L'algorithme 2 peut alors effectuer l'action lui-même. Cela fonctionne sur cet exemple car l'action a' effectuée deux fois a le même effet que lorsqu'elle

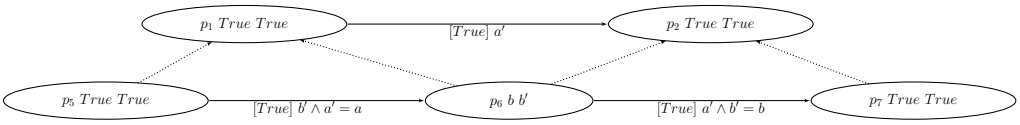

Fig. 5. Algorithmes 1, abstrait et concret.

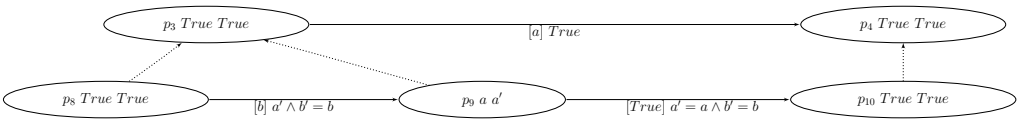

Fig. 6. Algorithmes 2, abstrait et concret.

est effectué une seule fois, en d'autres termes $a' \cdot a' \equiv a'$. Aussi, on vérifie aisément que les versions concrètes de ces deux algorithmes raffinent les versions abstraites.

5 Comparaisons

Il existe de nombreuses méthodes formelles pour prouver et/ou développer des algorithmes et des systèmes. Trois méthodes très connues et liées à nos travaux sont B, TLA$^+$ et l'approche «rely-guarantee». Dans cette section nous mettons en relief les différences entre ces méthodes et celle présentée ici. Les similitudes sont nombreuses puisque nous avons tout d'abord cherché à utiliser ces méthodes avant d'écrire la notre.

5.1 B

Abrial [1] a développé la méthode B ainsi que sa variante B événémentiel [2]. Elle permet de spécifier un système et d'en développer des raffinements. La méthodes est bien outillée et très utilisée dans le domaine industriel. Malheureusement, son orientation système rend complexes et peu lisibles les spécifications d'algorithmes.

En effet, il faut explicitement parler des processus et des places. De plus l'utilisation d'un unique invariant global le rend peu lisible. En effet, pour reconstruire l'invariant global, on a besoin de prédicats de place. Il faut alors compléter *effect* afin d'indiquer le contrôle de flot et de combiner les invariants locaux en un invariant global de taille importante.

Un inconvénient plus conceptuel de la méthode B réside dans le fait que le raffinement se fait toujours par événement. Ceci nous oblige d'anticiper les situations où un algorithme peut effectuer l'action d'un autre algorithme abstrait. Autrement dit, on est obligé d'indiquer les points de linéarisabilité ce qui peut rendre les spécifications assez obscures.

5.2 TLA$^+$ et $^+$CAL

Le langage de spécification TLA$^+$ et le langage $^+$CAL ont été développés par Lamport [8,9]. Les spécifications $^+$CAL sont en fait transformées en TLA$^+$ afin d'être interprétées et analysées par TLC, le model checker de TLA$^+$. Le langage $^+$CAL ressemble a du pseudo-code et a été développé afin de vérifier des algorithmes parallèles et distribués. Malheureusement, $^+$CAL souffre de deux problèmes. Le premier est qu'il est impossible de spécifier librement le niveau d'atomicité car il est souvent imposé par le langage (par exemple dans les branches d'une conditionnelle). Le deuxième problème est que $^+$CAL et TLA$^+$ ne sont outillés que d'un model checker. Or les spécifications qui nous intéressent sont difficilement traitables par un model checker.

5.3 Rely-Guarantee en Isabelle

L'approche «rely-guarantee» a été conçue par Jones [7], voir aussi le livre [3]. Son objectif est de permettre une vérification compositionnelle de programmes parallèles. La méthode étend les triplets de Hoare en rajoutant deux formules : la première caractérisant ce que peuvent faire les autres systèmes (comme notre condition *rely*), la deuxième caractérisant ce que fait le système (*guarantee*). On peut alors décrire dans *guarantee* l'effet atomique de l'algorithme. Mais ceci ne permet pas de vérifier complètement le système car il pourrait exécuter plusieurs fois l'action décrite par le prédicat *guarantee*. On pourrait rajouter des variables «fantômes» mais cela complique la spécification. Il n'est pas prévu non plus de développer conjointement des algorithmes par raffinement.

Une librairie «rely-guarantee» [12,13] pour Isabelle a été développée par Prensa Nieto. Cette formalisation nous permettrait de vérifier nos algorithmes à condition de rajouter les primitives atomiques. C'est ce que nous avons fait en travail préliminaire. Mais la méthode ne supporte pas le raffinement.

6 Conclusion et travail futur

On a présenté une méthode permettant de vérifier la spécification d'une famille d'algorithmes. Cette méthode se veut surtout adaptée à la vérification d'algorithmes sans verrou. Elle est fondée sur la description de plusieurs algorithmes qui sont exécutés par un nombre quelconque de processus et qui opérent sur une mémoire partagée. Nous avons formalisé les concepts d'algorithmes et de systèmes et défini les obligations de preuve pour s'assurer de la cohérence d'une spécification à un certain niveau d'abstraction. Une notion de raffinement d'algorithmes et de systèmes permet le développement conjoint de plusieurs algorithmes. Elle admet notamment qu'une opération d'un certain algorithme puisse être effectuée par un algorithme différent ; cette technique est souvent utilisée dans les algorithmes qui nous intéressent comme nous avons vu dans le cas du RDCSS (section 2.3). Or, ce type de raffinement est souvent difficile à justiier dans des méthodes généralistes comme B ou TLA$^+$.

Il nous reste à valider notre méthode en l'appliquant à des études de cas plus conséquents que les simples exemples présentés ici, notamment au cas de l'algorithme RDCSS puis CAS_n de Harris et al. [5]. Par ailleurs, nous sommes en train de justifier la méthode en la formalisant et en démontrant sa correction dans l'assistant de preuve Isabelle/HOL.

Afin de mettre à la disposition un environnement outillé de preuve et de développement aux concepteurs d'algorithmes sans verrou, nous étudierons l'instanciation de la méthode à une syntaxe concrète. Nous avons observé que, bien que représentant des systèmes à états infinis, les algorithmes n'effectuent pas en général des opérations complexes sur leurs données. C'est pourquoi nous souhaitons concevoir un générateur d'obligations de preuve qui pourra s'interfacer avec des outils de preuve automatiques. Afin de traiter des algorithmes plus complexes, il conviendra de prévoir le remplacement d'une transition par un sous-algorithme dont la correction a déjà été démontrée.

Références

1. Abrial, J.-R. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. Abrial, J.-R., Cansell, D., and Méry, D. Refinement and Reachability in Event_B. In *Formal Specification and Development in Z and B (ZB 2005)* (Guilford, UK, 2005), H. Treharne, S. King, and M. Henson, Eds., vol. 3455 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 222–241.
3. de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., and Zwiers, J. *Concurrency Verification : Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.
4. Gao, H. *Design and verification of lock-free parallel algorithms*. PhD thesis, University of Groningen, Apr. 2005.
5. Harris, T. L., Fraser, K., and Pratt, I. A. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (2002).
6. Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (January 1991), 124–149.
7. Jones, C. B. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems 5*, 4 (Oct. 1983), 596–619.
8. Lamport, L. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.
9. Lamport, L. Checking a multithreaded algorithm with +cal. In *20th Intl. Symp. Distributed Computing (DISC 2006)* (Stockholm, Sweden, 2006), S. Dolev, Ed., vol. 4167 of *Lecture Notes in Computer Science*, pp. 151–163.
10. Nipkow, T., Paulson, L., and Wenzel, M. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. No. 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
11. Parkinson, M., Bornat, R., and O'Hearn, P. Modular verification of a non-blocking stack. *SIGPLAN Not. 42*, 1 (2007), 297–302.

12. Prensa Nieto, L. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL.* PhD thesis, Technische Universität München, 2002.

13. Prensa Nieto, L. The rely-guarantee method in Isabelle/HOL. In *European Symposium on Programming (ESOP 2003)* (Warsaw, Poland, 2003), P. Degano, Ed., vol. 2618 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 348–362.

14. Sundell, H., and Tsigas, P. Lock-free and practical deques using single-word compare-and-swap. Tech. Rep. 2004-02, Computing Science, Chalmers University of Technology, Mar. 2004.

15. Vafeiadis, V., Herlihy, M., Hoare, T., and Shapiro, M. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06 : Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM Press, pp. 129–136.

16. Valois, J. D. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems* (Las Vegas, NV, 1994), pp. 64–69.