Développement prouvé de structures de données sans verrou

Loïc Fejoz

Nancy-Université
Université Henri Poincaré

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

➢ Contact SCD Nancy 1 : theses.sciences@scd.uhp-nancy.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Développement prouvé de structures de données sans verrou

THÈSE

présentée et soutenue publiquement le 26-30 janvier 2008

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy-Université

(spécialité informatique)

par

Loïc Fejoz

Composition du jury

Rapporteurs : Yamine Ait-Ameur
Christoph Weidenbach

Examinateurs : Jean-Paul Bodeveix
Claude Godart
Dominique Méry
Stephan Merz
Viktor Vafeiadis

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Mis en page avec la classe thloria.

Remerciements

Je remercie tout d'abord mon superviseur, Stephan Merz, pour son accompagnement et sa compréhension, ainsi que Pascal Fontaine pour ses nombreuses remarques de relecture. Je remercie également toute l'équipe Mosel pour ces agréables années passées ensemble. Je remercie les rapporteurs, et toutes les personnes que j'ai pu rencontrer à travers cette thèse. Je sais gré à *Microsoft Research* de leur soutien financier grâce leur programme de financement *Microsoft Research PhD Scholarship*. Enfin je suis reconnaissant envers ma femme Jacqueline.

à Erwann,
qui de toute façon aurait été trop jeune pour comprendre.

Table des matières

Chapitre 1

Introduction

Que ce soit pour des applications bureautiques, ou pour des serveurs hautes-performances, tous les logiciels doivent aujourd'hui utiliser au mieux le parallélisme offert par le matériel. Les architectures parallèles sont en effet devenues monnaie courante. Toute la communauté des développeurs est donc maintenant confrontée à l'écriture de programmes concurrents. Malheureusement, la conception de tels programmes est difficile, beaucoup plus difficile que l'écriture de programmes séquentiels. Et pourtant l'écriture, la maintenance et le test de programmes séquentiels bénéficient de plus de 30 ans de recherche en outils et méthodes. Nous recommandons d'ailleurs la lecture du livre de Herlihy et Shavit [31] pour une introduction à la programmation multi-processeur.

Si de plus on considère l'invasion des logiciels embarqués dans notre vie quotidienne, s'assurer de la correction de ceux-ci est plus que nécessaire afin d'assurer leur sûreté de fonctionnement. Mais les bugs concurrents sont difficiles à reproduire. Tout d'abord il est difficile de retrouver les entrelacements d'exécutions des différents processeurs qui ont amené l'erreur de conception à se manifester. Une fois l'erreur de conception trouvée, il est souvent difficile de reproduire des entrelacements précis et donc de rendre automatiques les tests de régression. De plus, pour corriger l'erreur, il faut prendre en compte tous les entrelacements possibles des exécutions. Et effectivement une étude [49] a montré que plus de 68% des bugs concurrents sont dûs à des défauts d'atomicité.

L'adaptation des langages de programmation est une des pistes explorées. Les *Software Transactional Memory* (STM) [74] font partie de cette mouvance. Cette construction permet de rendre atomique une séquence d'instructions. Elle permet plus facilement au programmeur de raisonner sur son programme et donc d'éviter certains bugs. Mais cette construction ne permet pas d'éviter tous les bugs comme l'a aussi montré Lu et al. [49]. Ainsi un autre groupe d'erreurs fréquentes correspond à une violation de l'ordre temporel souhaité. C'est notamment le cas quand un processus initialise une variable qui devrait être utilisée seulement *ensuite* par un autre processus. De toutes façons les STM ne sont pas disponibles dans tous les langages et leur implémentation (correcte) n'est pas triviale. La vérification de programmes concurrents reste donc primordiale.

Cependant, pour vérifier un programme, il faut pouvoir caractériser ce que l'on attend de lui, c'est-à-dire sa spécification. Dans un monde séquentiel, la spécification établit la relation entre les entrées et sorties d'un programme. Dans un monde concurrent, cela ne suffit plus, il faut aussi spécifier comment vont interagir les différentes parties, autrement dit les entrelacements autorisés. C'est pour cela que la notion d'atomicité est importante. L'atomicité indique qu'une séquence d'instructions ne doit pas être interrompue ou corrompue par d'autres. De plus, comme il plus facile de raisonner de manière séquentielle, on désire n'autoriser que les exécutions telles que les effets du programme semblent séquentiels. Un moyen simple d'y parvenir est d'assurer l'exclusion mutuelle.

L'exclusion mutuelle –présentée au chapitre 2– est traditionnellement assurée par des verrous. Malheureusement cela ne permet pas d'utiliser efficacement le parallélisme et apporte aussi son lot d'erreurs. Par exemple, il serait peu efficace d'imposer l'exclusion mutuelle à deux processus partageant une queue (file) d'objets, et dont l'un ne ferait que rajouter des éléments, et l'autre que les enlever. Il faut donc définir précisément le degré de parallélisme que l'on autorise, et c'est tout l'objet du chapitre 3. Dans [13], De Roever *et al.* formalisent plusieurs méthodes de vérification et leurs propriétés. Cet ouvrage

constitue une bonne référence.

Dans cette thèse, nous nous focaliserons sur les algorithmes sans verrou. La principale motivation est que ces algorithmes sont au cœur des programmes parallèles. En effet, on les retrouve notamment dans l'implémentation des STM. Mais aussi, parce que nombre d'idiomes de programmation parallèle font appel à des structures de données sans verrou. Par exemple, si l'on veut faire communiquer deux processus par messages interposés, ils ont besoin d'une boite aux lettres. Celle-ci ne doit pas être bloquante et il est donc préférable de l'implémenter avec une file sans verrou. Il existe de nombreuses bibliothèques de structures de données sans verrou [36, 41, 73, 8, 72, 33, 79, 11, 26] disponibles. Or, peu d'entre elles ont été vérifiées formellement, et encore moins avec des assistants à la preuve.

L'usage d'un assistant à la preuve est intéressant à plus d'un titre. Il permet d'augmenter la confiance de notre raisonnement. Et, ce faisant, il oblige surtout à tout expliciter et donc à ne laisser aucune zone d'ombre. La confiance dans la correction d'un programme est d'autant plus importante que le coût engendré par un bug est dommageable. Or, la sûreté des hommes et des machines dépend de plus en plus de programmes. Cela peut même devenir une question vitale dans certains cas (transport, assistance médicale, etc). Un assistant à la preuve permet de nous rapprocher de la confiance à 100% mais sans l'atteindre. En effet, un assistant à la preuve est souvent constitué d'un petit noyau de code qui ne garantit pas la correction. Mais toute l'attention étant portée sur cette portion de code, on la considère comme sûre.

De plus, comme tout doit être formalisé, cela oblige à entrevoir toutes les possibilités sans en oublier aucune. Contrairement à des tests qui ne permettraient que de vérifier certaines d'entre elles. Par ailleurs, la formalisation peut aussi servir de documentation de référence, notamment sur les hypothèses de fonctionnement.

Les spécifications étant données, on pourrait les vérifier à l'exécution [19] comme certains langages de programmation le font pour le typage. Cette solution est donc un intermédiaire entre efficience et correction. Mais seule la vérification de la correction lors de l'écriture des programmes permettrait d'obtenir l'efficience maximale. Pour cela, il existe plusieurs méthodes dont certaines sont présentées au chapitre 4. Mais notre expérience de ces méthodes a montré qu'elles n'étaient pas forcément adaptées aux algorithmes sans verrou utilisés pour l'implémentation des structures de données. Certaines nous obligent à encoder de nombreux aspects dans leur logique ce qui par la suite ne permet plus d'avoir des preuves automatiques. D'autres, étant peu outillées, ne permettent pas de les exploiter complètement. Sur ces bases, nous avons développé une méthode spécifique.

Cette méthode est détaillée dans le chapitre 5. Elle modélise un ensemble de processus comme une machine à états. Les transitions autorisées sont données par la spécification des algorithmes suivis par chacun des processus. La vérification est basée sur le raffinement. C'est-à-dire que l'on vérifie qu'une machine dite "concrète" simule une machine "abstraite". La machine concrète exécute les algorithmes sans verrou. Les transitions de la machine abstraite sont, quant à elles, données par la définition des types abstraits correspondant.

Les types abstraits sont en fait une définition fonctionnelle des structures de données. Mais si l'on prend le même point de vue qu'en orienté-objet, c'est la structure elle-même qui change. Elle représente donc son propre état. Ainsi les évènements à l'origine des transitions abstraites sont les appels aux opérations (méthodes) de la structure de données. La méthode ainsi définie permet de vérifier que tous les entrelacements possibles sont une exécution correcte et valide. Elle est de plus conçue pour être compositionnelle. La compositionnalité permet de raisonner sur des sous-ensembles du programme et d'assurer tout de même la sûreté de l'ensemble. Elle est aussi automatique dans le sens où la vérification d'un raffinement correct peut être validée par des prouveurs automatiques.

Comme notre méthode est utilisée pour la vérification de programmes, il est nécessaire de la vérifier elle-même. Pour gagner en confiance dans notre méthode, nous avons donc fait sa preuve dans un assistant à la preuve (cf chapitre 6). Dans l'idéal toute la chaîne de développement devrait être prouvée, compilateur inclus [35] ! La conception et la vérification d'une telle méthode est en soit une tâche ardue et l'assistant à la preuve est alors un outil indispensable.

La méthode a été développée avec en tête les algorithmes trouvés dans les librairies précédemment citées. Un algorithme, le RDCSS, a retenu toute notre attention. Il concentre en effet dans un algorithme simple à comprendre plusieurs astuces. Le RDCSS est un algorithme intermédiaire permettant d'implémenter CAS_n avec CAS_1 [28]. Ces opérations sont aussi des briques de bases pour les autres algorithmes.

C'est donc sur cette exemple, non trivial à prouver, que nous avons testé notre méthode (cf chapitre 7).

Chapitre 2

Exclusion Mutuelle

2.1 Introduction

L'exclusion mutuelle est une technique de base dans les structures de données partagées. En effet, c'est elle qui, traditionnellement, nous permet de mettre à jour une structure de données tout en garantissant la correction de l'opération. Le raisonnement est simple : chaque opération, supposée correcte, se faisant l'une après l'autre, il ne peut y avoir d'effet de bord de l'une sur l'autre. L'exclusion mutuelle évite donc toutes opérations concurrentes.

Pour pouvoir raisonner sur un système, il faut pouvoir décrire son état ainsi que les changements de celui-ci. Or raisonner sur les changements, c'est aussi raisonner sur le temps. En effet, il est nécessaire de comparer l'ordre des évènements ainsi que la durée d'intervalles, etc. Un système sera donc considéré comme une machine à état. Il est lui-même composé de processus (*aka* thread) pouvant eux aussi être considérés comme des machines à état.

Les transitions seront appelées *évènements*. Un évènement n'a pas de durée, il apparaît comme étant atomique et instantané. Deux évènements ne peuvent donc se produire en même temps. Dès lors, on peut ordonner les évènements chronologiquement selon leurs apparitions. Nous noterons $e_1 \longrightarrow e_2$ le fait que l'évènement e_1 se soit produit avant e_2. Comme il n'est pas toujours possible de comparer l'ordre d'apparition d'évènements de processus distincts, $\longrightarrow$ définit donc un ordre partiel. On étend cette notation aux intervalles de temps. Soit d_1, d_2, e_1, e_2 des évènements, soit $I_d = [d_1, d_2]$ et $I_e = [e_1, e_2]$ des intervalles de temps délimités par ces évènements, alors $I_d \longrightarrow I_e$ si et seulement si $d_2 \longrightarrow e_1$.

2.2 Section critique

En 1965, Dijkstra [15] définit le problème de l'exclusion mutuelle comme suit. Considérons n processus dont les évènements sont cycliques. A chaque cycle, il existe une *section critique*. C'est-à-dire qu'il existe un intervalle de temps où un seul processus peut voir ses évènements de sa section critique apparaître. Les processus communiquent par l'intermédiaire d'une mémoire partagée. Par ailleurs, la solution doit satisfaire les conditions suivantes :

- La solution doit être symétrique. Tous les processus exécutent donc le même algorithme.
- Aucune hypothèse ne doit être faite sur la vitesse d'exécution relative des processus, pas même que la vitesse d'un processus est constante dans le temps.
- Un processus s'arrêtant en dehors de la section critique ne doit pas pouvoir bloquer les autres.
- Si plusieurs processus peuvent entrer en section critique, la décision de qui doit entrer en premier ne peut être repoussée indéfiniment.

Au passage, notons que le problème existe aussi si les processus communiquent par l'intermédiaire de messages. En modélisant explicitement les envois et réceptions de messages avec des boites aux lettres partagées, on peut ramener le cas réparti à de la mémoire partagée. La propriété correspondant à l'exécution en solitaire du bloc de code correspondant à la section critique est appelée *exclusion mutuelle*. Dans un problème à deux processus a et b, si l'on note CS_x^n l'intervalle correspondant à la n-ième exécution de

la section critique du processus x, il faut donc que a soit en section critique avant b, noté ($CS_a^j \longrightarrow CS_b^k$, soit l'inverse. La propriété se formalise ainsi :

$$\forall j, k.(CS_a^j \longrightarrow CS_b^k) \vee (CS_b^k \longrightarrow CS_a^j)$$

La question d'algorithmes assurant cette propriété a longtemps été de peu d'intérêt pratique du fait de l'architecture même des ordinateurs. Sur une machine monoprocesseur il suffit en effet d'interdire les interruptions logicielles et matérielles pour assurer l'exclusion mutuelle. Mais avec l'apparition d'ordinateurs multiprocesseurs, ceci est devenu un vrai problème. De plus, le problème a été étendu par la mise en réseau d'ordinateurs. La communication ne se faisant plus alors par une mémoire partagée mais par l'envoi de messages.

```
int temp = x; /* Début de la zone de danger */
x = temp + Cst_pid; /* Fin de la zone de danger */
```

FIG. 2.1 – Incrémentation de la variable globale x

Pour la suite, nous supposerons que la section critique cherche à incrémenter une variable globale x par une constante Cst_{pid} propre à chaque processus (identifié par son pid). L'incrémentation se fait en deux évènements comme indiquée par le pseudo-programme de la figure 2.1. Notons $write_a^j(temp = x)$ le j-ème évènement où le processus a affecte la valeur de x à $temp$. Imaginons que cette suite d'évènements soit observée lors des tests :

$$write_a^j(temp_a = x) \longrightarrow write_b^k(temp_b = x) \longrightarrow write_a^j(x = temp_a + Cst_a) \longrightarrow write_b^k(x = temp_b + Cst_b)$$

Cette exécution n'est pas conforme à l'intention du programmeur. On doit donc restreindre les exécutions possibles. Pour cela, une des solutions est l'utilisation de verrou.

```
lock(aGlobalLock); /* Début de la section critique, représentée par l'intervalle de temps CS_a */
int temp = x;
x = temp + Cst_pid;
unlock(aGlobalLock); /* Fin de la section critique */
```

FIG. 2.2 – Utilisation de verrou

Comme indiqué par le programme de la figure 2.2, on associe un verrou à chaque section critique. Dans l'exemple, il n'y a qu'une seule section critique et donc qu'un seul verrou global *aGlobalLock*. Chaque processus appelle la méthode *lock* du verrou avant d'entrer dans la section critique afin d'*acquérir* le verrou, et *unlock* en sortant afin de le *libérer*. Une bonne implémentation de verrou assure les trois propriétés suivantes :

- L'*exclusion mutuelle* de la section critique est réalisée, *ie* deux processus différents ne peuvent detenir le verrou en même temps.
- *absence de blocage* : Sur tous les processus appelant *lock*, certains réussiront à acquérir le verrou. Si un processus n'arrive pas à acquérir le verrou cela signifie que d'autres l'acquièrent une infinité de fois.
- *absence de famine* : Chaque processus essayant d'acquérir le verrou finira par y parvenir. Chaque appel à *lock* finira.

Notez que l'absence de famine implique l'absence de blocage.

2.3 Sémaphore

Une implémentation classique de *lock* et *unlock* utilise les méthodes P et V des sémaphores. Un sémaphore est une variable ou un type de données abstrait comptant le nombre de ressources libres.

```
void P(Semaphore s) { /* Acquire resource. Aka lock. */
  wait until s > 0 { s−−; }
  /* must be atomic once s > 0 is detected */
}

void V(Semaphore s) { /* Release resource. Aka unlock. */
  s++; /* Must be atomic */
}

void Init(Semaphore s, int v) {
  s = v;
}
```

FIG. 2.3 – Pseudo-code d'implémentation des sémaphores

Deux opérations modifient les sémaphores. P est l'opération qui permet d'acquérir une ressource. Elle bloque tant qu'une ressource ne s'est pas libérée puis décrémente le compteur. V libère une ressource. Elle indique donc que l'on n'utilise plus la ressource. La figure 2.3 présente le pseudo-code des opérations sur les sémaphores. Évidemment, on peut aussi initialiser un sémaphore au nombre de ressources disponibles initialement. S'il n'y a qu'une seule ressource disponible alors le sémaphore est aussi appelé *mutex*.

Les méthodes P et V sont aussi connues sous les termes *acquire-release* et *wait-signal*. Le second nom est représentatif d'une implémentation des sémaphores. En effet, l'attente dans la fonction P correspond à un réveil du processus. Ce réveil est provoqué par l'incrémentation effectuée dans la fonction V. Une implémentation possible peut donc utiliser une file d'attente des processus. La fonction V signalant alors au processus dans la queue le changement de la valeur de s. Mais rien ne sert de réveiller tous les processus puisqu'un seul pourra atomiquement décrémenter s. Les implémentations garantissent généralement que les processus seront réveillés dans l'ordre dans lequel ils ont été suspendus.

Ceci garantit l'équité. Pour l'exclusion mutuelle, l'équité est la propriété qui garantit que chaque processus a autant de chance d'accéder à la section critique. Autrement dit, il n'y a pas de discrimination entre processus. Les processus agissant de manière concurrente, on élargit cette notion à toutes les actions de chacun des processus. L'équité a donc un lien avec la *vivacité* (*liveness*).

On se focalise souvent sur la *correction*, sous-entendu, pour les programmes qui terminent. Mais que peut-on dire des programmes qui ne terminent pas ? Peuvent-ils tous terminer ? Certains seulement ? En fait, la question de la vivacité est souvent posée indépendamment de la correction car elle dépend de la plate-forme sur laquelle on fait fonctionner les algorithmes. En rajoutant des hypothèses sur l'*équité* des processus, on peut alors répondre à ces questions. On définit les propriétés suivantes pour différencier les cas :

wait-free : tous les processus progressent même s'il y a du délai ;

lock-free : certains processus progressent ;

obstruction-free : les processus exécutés de manière isolée progressent.

Ainsi, si l'on ne rajoute pas des hypothèses d'équité, on ne peut pas garantir que chacun des processus utilisant des sémaphores finira. Il est d'usage d'annoter les actions. Ainsi la notion d'*équité faible* d'une action A signifie que s'il est continuellement (sans interruption) possible d'effectuer une action A alors A finira par être effectuée. C'est la propriété qui est utilisée lorsque l'on veut modéliser le fait qu'un processeur ne peut rester à ne rien faire s'il y a une instruction à exécuter. L'*équité forte* est souvent utilisée sur les conditions des boucles des algorithmes. Elle est définie comme : si A est infiniment souvent demandée, alors A sera obtenue (infiniment souvent). Autrement dit plus on essaie, plus on a de chance d'y parvenir.

Dans la pratique, ces annotations sont justifiées soit par la connaissance de la machine et de son ordonnanceur, soit par des calculs de probabilités, soit par la connaissance de la contention, soit par hypothèse de non présence de pannes. Par exemple, un processeur en panne peut être simulé comme

un processeur n'exécutant pas la prochaine instruction. Certaines des méthodes présentées au chapitre 4 permettent d'aborder ces questions. Néanmoins, par la suite, seule la question de la sûreté sera relevée.

2.4 Solution pour 2 processus

```
/**
 * Block until the caller can enter the critical section.
 */
void lock(FischerLock lock) {
  do {
    while (lock.turn != −1) {};
    lock.turn = pid;
    delay();
  } while (lock.turn != pid);
}

/**
 * Called when the caller leaves the critical section.
 */
void unlock(FischerLock lock) {
  lock.turn = −1;
}
```

FIG. 2.4 – Algorithme de Fischer

Une implémentation des plus simple des méthodes *lock* et *unlock* suggérée par Michael Fischer est donnée dans la figure 2.4. Intuitivement cet algorithme correspond à l'attitude suivante :

1. attendre que personne ne veuille le tour ;
2. dire alors que je le veux ;
3. attendre la réaction des autres pendant un certain temps ;
4. si toujours personne d'autre ne le veux, le prendre, sinon recommencer

Le champ *turn* indique donc le numéro de processus dont c'est le tour. -1 indique que personne n'a pris le tour. Toute l'astuce réside dans la méthode *delay*. Elle doit durer suffisamment longtemps pour qu'un autre processus ait le temps de sortir de la boucle d'attente et d'affecter son *pid* à *turn*. Cet algorithme bien que ne répondant pas à la condition sur les hypothèses de vitesse d'exécution requises par Dijkstra (voir section 2.2), est intéressant comme point de départ. Une première observation est que l'algorithme utilise l'*attente active*. C'est-à-dire qu'il utilise une boucle pour surveiller le changement de valeur d'une variable. Cette technique, bien que très utilisée, n'est pas forcément recommandable car elle empêche la mise en veille ou la planification d'autres tâches au processeur. Une autre solution utiliserait des signaux comme les sémaphores de la section précédente. Le choix d'implémentation dépend aussi du nombre de processeurs.

L'algorithme de Peterson pour deux processus ($pid \in \{0, 1\}$), présenté à la figure 2.5 utilise deux champs. On retrouve le champs *turn* indiquant à qui est le prochain tour, et un deuxième champ *flag* qui indique si oui ou non un processus veut entrer en section critique. Acquérir le verrou consiste donc simplement à mettre à jour son entrée dans *flag* et attendre soit son tour $turn = pid$, soit que l'autre processus ne désire pas entrer en section critique $flag[1 - pid] = 0$. Voir la trace de la figure 2.6. L'algorithme de Peterson vérifie bien les conditions précédemment citées (cf section 2.2), autrement dit il vérifie notamment le théorème suivant.

Théorème 1 *Deux processus p_0 et p_1 exécutant l'algorithme de Peterson ne peuvent être en section critique simultanément.*

```
/**
 * Block until the caller can enter the critical section.
 */
void lock(PetersonLock lock) {
  lock.flag[pid] = 1;
  lock.turn = 1−pid;
  while(lock.flag[1−pid] && lock.turn = (1−pid)) {};
}

/**
 * Called when the caller leaves the critical section.
 */
void unlock(PetersonLock lock) {
  lock.flag[pid] = 0;
}
```

FIG. 2.5 – Algorithme de Peterson pour deux processus.

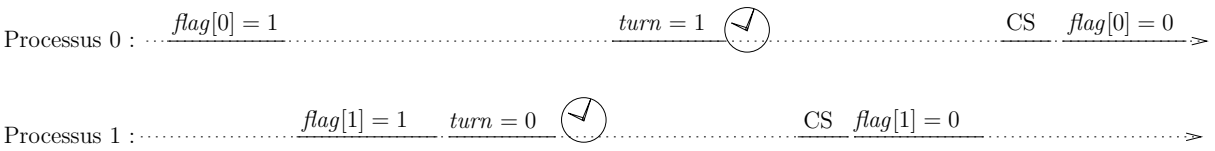


FIG. 2.6 – Exemple de trace de 2 processus utilisant l'algorithme de Peterson. L'horloge correspond à l'attente effectuée dans la boucle.

Preuve. La preuve se fait par contraposition. Supposons p_0 et p_1 simultanément en section critique, respectivement CS_0^j et CS_1^k. Supposons de plus que p_0 soit entré le premier en section critique. Cette condition ne change pas la généralité de la preuve. De par le respect de l'ordre d'exécution de l'algorithme, on obtient :

$$(\mathit{flag}[0] = 1)_0^j \longrightarrow (\mathit{turn} = 1)_0^j \longrightarrow (\mathit{while}\ldots)_0^j \longrightarrow CS_0^j$$

Puisque p_0 est sorti de la boucle, à cet instant, soit $\mathit{flag}[1] = 0$, soit $\mathit{turn} = 0$.

Dans un premier temps, supposons que $\mathit{flag}[1] = 0$. p_1 a dû exécuté :

$$(\mathit{flag}[1] = 1)_1^k \longrightarrow (\mathit{turn} = 0)_1^k \longrightarrow (\mathit{while}\ldots)_1^k$$

Mais comme p_0 est en section critique, $\mathit{flag}[0] = 1$ et donc p_1 n'a pas pu sortir de la boucle et donc entrer en section critique. L'exclusion mutuelle est donc vérifiée dans ce cas-ci.

Maintenant, supposons $\mathit{turn} = 0$. Puisque p_1 est en section critique, $\mathit{flag}[1] = 1$. De plus, puisque $\mathit{turn} = 0$, p_1 a exécuté la seconde ligne après p_0. :

$$(\mathit{turn} = 1)_0^j \longrightarrow (\mathit{turn} = 0)_1^k$$

Mais comme $(\mathit{turn} = 0)_1^k \longrightarrow (\mathit{while}\ldots)_1^k$, les conditions n'ont pas pu être validées et donc p_1 ne peut être en section critique. p_0 est donc bien le seul en section critique.

p_0 et p_1 ne peuvent donc être ensemble en section critique. □

2.5 Généralisation

Dans la section précédente, il n'a été question que de deux processus. Cette simplification facilite la compréhension. Mais à l'origine ces algorithmes peuvent fonctionner avec un nombre n quelconque de processus. Les algorithmes sont donc paramétrés par n. Leur vérification est alors généralement faite

par l'utilisation d'un invariant. Une assertion I est un invariant si chaque action du système considéré conserve celle-ci. Ainsi pour prouver une propriété P, il faudra trouver un invariant I tel que P soit une conséquence de I et tel que I soit une conséquence des conditions initiales. Les propriétés vérifiables par cette technique sont appelées propriétés d'invariances, et sont donc des propriétés de sûreté. Elles permettent d'affirmer que rien de mauvais ne peut arriver. Il est souhaitable d'automatiser la preuve des invariants. Différentes techniques existent pour cela [23]. On y gagne en confidence et cela rend la tâche de vérification plus facile d'accès en simplifiant le processus de vérification.

Le cadre de cette thèse se situe clairement dans un cadre paramétré, le nombre de processus y est quelconque. De plus, la technique des invariants sera la principale technique de vérification. Par ailleurs, nous resterons dans un cadre avec mémoire partagée. Il existe néanmoins des algorithmes d'exclusion mutuelle spécifiques quand la communication inter-processus se fait par échanges de messages [70, 51]. Le chapitre suivant va nous permettre de préciser ce qu'est une structure de données et quels sont les propriétés désirées.

Chapitre 3

Structures de données partagées

Le précédent chapitre a permis de se familiariser avec les concepts de verrou et d'exécutions parallèles. Par ailleurs, nous avons commencé à voir les notions de formalisation, de correction et de progression. Dans ce chapitre, nous préciserons la notion de correction. La correction d'une implémentation donnée est souvent basée sur l'équivalence observationnelle avec un programme séquentiel de référence, l'équivalence étant plus ou moins stricte selon les applications. En particulier, nous introduirons la *consistance séquentielle* et la *linéarisabilité*. Dans un deuxième temps, nous détaillerons les différentes architectures matérielles disponibles. Ceci nous permettra de classifier les types de modèles mémoires existants, ainsi que différentes primitives essentielles à l'implémentation d'algorithmes sans verrou.

3.1 Concurrence et correction

Dans le chapitre 2, nous avons abordé la vérification de l'exclusion mutuelle de quelques algorithmes. Dans ce chapitre, nous nous intéressons à la correction. Que signifie qu'un programme est correct ? Correct par rapport à quoi ? Étant donné une exécution, un programmeur peut facilement dire si celle-ci est valide ou non. Il existe donc implicitement une spécification. Toute la difficulté se concentre donc dans l'expression de cette spécification. La plus simple est probablement de partir d'une spécification séquentielle. Une deuxième question se pose alors : quel est le lien entre ce programme séquentiel et le programme en question ? En effet, l'appel à une méthode n'est pas atomique, il dure un certains temps. Comment ordonner alors les appels ? Doit-on ordonner en fonction de l'ordre d'appel ? Faire en sorte que l'effet atomique se trouve quelque part entre le début et la fin de l'appel ? Quelle assurance a-t'on sur l'ordre de deux appels consécutifs ? C'est à toutes ces questions que nous allons répondre dans ce chapitre. Mais le principe clef à retenir est qu'il est plus simple de raisonner sur des exécutions séquentielles.

3.2 Définition séquentielle des types abstraits

Prenons l'exemple d'une file d'attente d'une cafétéria. Les évènements suivants peuvent survenir :
- vérifier s'il y a encore quelqu'un dans la file
- ajouter un client dans la file
- servir la personne en tête
- créer une nouvelle file
- voir qui est en tête de file

Les opérations correspondantes du type abstrait "file d'attente" sont donc :

isEmpty vérifier que la file est vide ;

enqueue ajouter un élément dans la file ;

dequeue enlever le premier élément de la file ;

new créer une nouvelle file ;

head regarder le premier élément de la file.

Un type abstrait [27] est la définition logique d'une sorte, avec ses opérations et les axiomes correspondants. Soit E le type des éléments des files de type $Queue\langle E\rangle$, les opérations ont les types suivants :

- $isEmpty\colon Queue\langle E\rangle \to \mathbb{B}$
- $enqueue\colon Queue\langle E\rangle \to E \to Queue\langle E\rangle$
- $dequeue\colon Queue\langle E\rangle \nrightarrow Queue\langle E\rangle$
- $new\colon \to Queue\langle E\rangle$
- $head\colon Queue\langle E\rangle \nrightarrow E$

On peut alors définir naturellement un certain nombre d'axiomes concernant les files. Par exemple, une file nouvellement crée est vide $isEmpty(new()) = \top$. De même, qu'une file dans laquelle on vient d'insérer un élément n'est pas vide $isEmpty(enqueue(q, e)) = \bot$. Par contre les opérations *dequeue* et *head* ne sont définies que si la liste n'est pas vide. On a donc besoin de définir des pré-conditions. On notera $pre(head(q)) \triangleq \neg isEmpty(q)$ et $pre(dequeue(q)) \triangleq \neg isEmpty(q)$ le fait que les opérations *dequeue* et *head* ne peuvent être appliquées que sur des files non vides. Une axiomation des files d'attentes pourrait donc être : $\forall q\colon Queue\langle E\rangle, e\colon E$.

- $isEmpty(new()) \triangleq \top$
- $isEmpty(enqueue(q, e)) \triangleq \bot$
- $head(enqueue(q, e)) \triangleq$ IF $isEmpty(q)$ THEN e ELSE $head(q)$
- $dequeue(enqueue(q, e)) \triangleq$ IF $isEmpty(q)$ THEN q ELSE $enqueue(dequeue(q), e)$
- $pre(head(q)) \triangleq \neg isEmpty(q)$
- $pre(dequeue(q)) \triangleq \neg isEmpty(q)$

Une présentation plus complète est disponible dans [84]. Cette définition abstraite correspond donc à une axiomatisation du type abstrait "file d'attente". Mais si l'on considère maintenant les exécutions où les transitions correspondent aux opérations et les états sont caractérisés par la file, on a alors une définition séquentielle. Cette définition est quasi-identique à la définition des *Application Program Interface* (API) d'une file en langage orienté objet. Les pré- et post-conditions permettent de définir séquentiellement une structure de données.

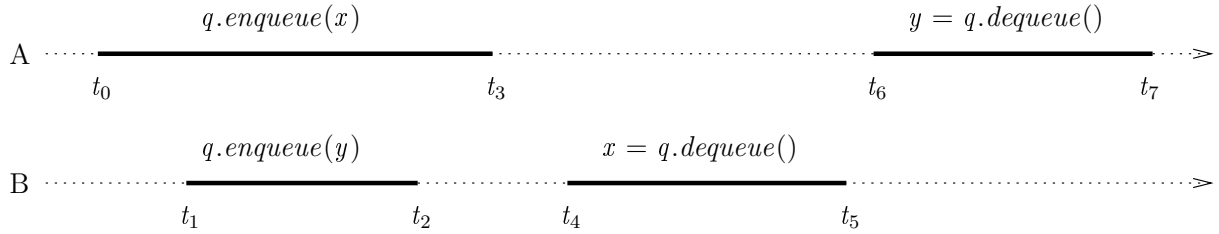


FIG. 3.1 – H_1, une trace qui pourrait être observée lors d'une séance de tests avec 2 processus A et B. Cette trace est intuitivement acceptable si q est supposée vide avant cette observation. Les intervalles sont délimités par les appels et les retours t_i des opérations sur la file q. $y = q.dequeue()$ signifie que l'élément dépilé de q par cette opération est y.

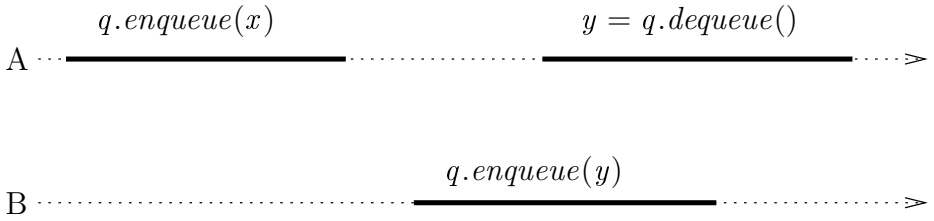


FIG. 3.2 – H_2 une trace non acceptable si q est supposée vide avant : x devrait être dépilé avant y.

Malheureusement, dans un monde parallèle, cette définition n'est pas suffisante. Supposons une file q partagée par plusieurs processus, les exécutions des opérations peuvent alors se chevaucher. On ne peut plus alors parler d'ordre entre ses appels. Les figures 3.1, 3.2, 3.3, 3.4 montrent différentes traces où deux

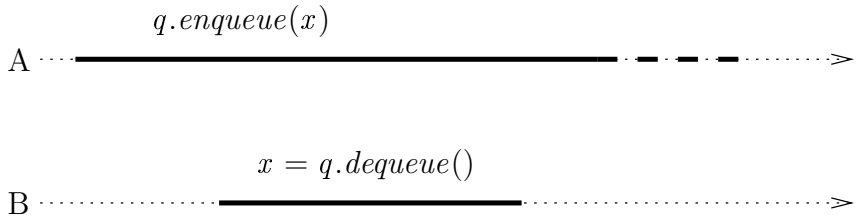


FIG. 3.3 – H_3 une trace acceptable : l'empilement a eu lieu bien que l'opération ne soit pas terminée.

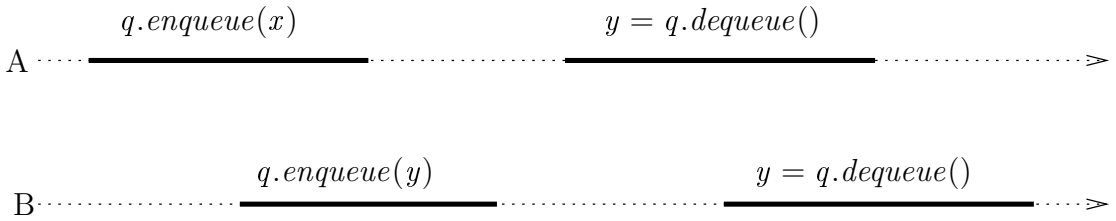


FIG. 3.4 – H_4 une trace non acceptable : y ne peut être dépilé 2 fois.

processus manipulent la file q. On suppose l'axe du temps partant vers la droite. Chaque opération dure un intervalle de temps indiqué en dessous de l'opération. Les intervalles qui se chevauchent indiquent donc des opérations concurrentes.

Les types abstraits sont intéressants pour formaliser les opérations. En pratique, la structure partagée encapsule l'état et les méthodes permettent de changer celui-ci. Par la suite, on va donc utiliser la notation orientée-objet. Ainsi $x = q.dequeue()$ signifie que x a été dépilé de q. C'est l'équivalent de $x = head(q) \wedge q' = dequeue(q)$. De même, $q.enqueue(x)$ correspond à $q' = enqueue(q, x)$

La trace H_1 de la figure 3.1 est clairement acceptable. Elle correspond à notre intuition sur le fonctionnement d'une queue concurrente. Cette trace montre que les deux processus empilent de manière concurrente x et y. Puis le processus 2 dépile x, et enfin le processus 1 dépile y. Intuitivement, les opération d'empilement étant concurrentes, on imagine bien qu'une des deux opérations a eu lieu avant l'autre. En l'occurence , c'est l'empilement de x. Cet ordre est donc acceptable.

La trace H_2 de la figure 3.2 n'est pas intuitivement acceptable. En effet, il est clair que x a été empilé avant y. y ne peut donc pas être dépilé avant. H_3 de la figure 3.3 est acceptable bien que l'opération d'empilement ne soit pas terminée. Enfin H_4 de la figure 3.4 est clairement inaceptable étant donné que y est dépilé deux fois. Seule *l'intention de la sémantique* de la structure de données en question nous permet d'affirmer si une trace est acceptable ou non.

3.3 Caractérisation d'une trace

Contrairement au chapitre 2, dans lequel une opération était atomique, l'appel à une opération est caractérisé par l'opération appelée, le moment de l'appel ainsi que le moment du retour. Une exécution ou *trace* caractérise l'historique des événements. Herlihy et Shavit [31] posent les définitions suivantes :

Définition 2 *Une* trace *est une liste finie d'évènements, un évènement pouvant être soit l'*invocation *d'une méthode, soit le* retour *de celle-ci.*

Définition 3 *Un* appel *à une méthode dans la trace H est une paire formée par deux évènements. Le premier évènement correspond à l'invocation, le deuxième au retour de la même méthode, sur le même objet, par le même processus et la plus proche du premier évènement.*

Définition 4 *Une invocation est* pendante *(dans la trace H) s'il n'y a pas d'évènement de retour correspondant.*

Définition 5 *Une* extension *de H est une trace à laquelle plusieurs évènements de réponses ont été rajoutés aux invocations pendantes.*

Définition 6 *Une sous-trace de H est une trace étant une sous-liste des évènements de H. complete(H) est la sous-trace contenant tous les appels et uniquement eux. On rejete donc toutes les invocations sans retours et les retours sans invocations.*

Définition 7 *Une trace est* séquentielle *si*
- *le premier évènement est une invocation,*
- *chaque invocation est directement suivie du retour sauf peut-être le dernier évènement.*

Définition 8 *Une sous-trace de H d'un processus A est la sous-trace, notée $H|A$, contenant tous les evènements du processus A. On définit similairement la sous-trace d'un objet $H|x$.*

Définition 9 *Deux traces H et S sont dites* équivalentes *si, et seulement si, pour tous processus A, $H|A = S|A$.*

Définition 10 *Une trace est valide ou* bien formée *si, et seulement si, pour tout processus A, $H|A$ est séquentielle.*

3.4 Consistance séquentielle

La consistance séquentielle, ainsi que d'autres critères comme la sérialisabilité, sont des notions issues de la communauté des bases de données [81]. Ces notions ont ensuite évolué dans le cadre des systèmes asynchrones et distribués [55]. L'idée de base est de ramener le raisonnement sur des programmes parallèles au cas séquentiel. Pour cela il faut s'assurer que n'importe quelle trace du système réel a les mêmes effets qu'une trace séquentielle abstraites. Les deux sections suivantes discutent donc des relations permises ou non entre une trace abstraite et une trace concrète.

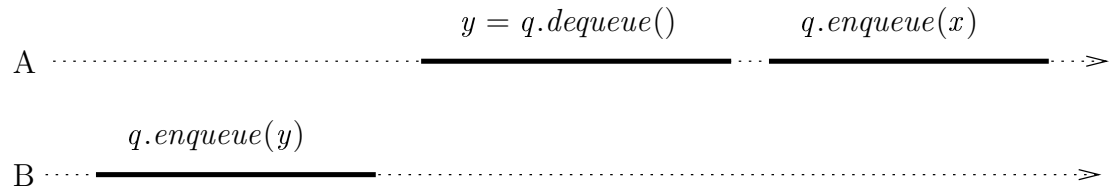


FIG. 3.5 – Trace abstraite expliquant la trace H_2 : les 2 opérations du processus A sont interverties.

Dans la trace H_2 de la figure 3.2, un processus empile x puis un autre y. Plus tard, le premier processus dépile y. A priori cette trace n'est pas acceptable puisque l'on devrait retrouver x. La figure 3.5 donne une explication de ce qui peut s'être passé. L'ordre des opérations du processus A est inversé. En supposant que l'ordre initial soit donné par un algorithme, ce serait surprenant. C'est ce que suggère la consistance séquentielle [42] : les effets des opérations doivent suivre le même ordre que celui donné par l'algorithme. Un deuxième principe est que les effets des opérations doivent apparaître comme s'ils s'effectuaient les uns après les autres. Ces deux principes définissent la consistance séquentielle.

Définition 11 *Une trace valide H est dite séquentiellement consistante si, et seulement si, il existe une extension H' de H et une trace séquentielle $\overline{H}$ telle que complete(H') est équivalente à $\overline{H}$*

$\overline{H}$ étant une trace séquentielle, elle formalise le deuxième principe. Le premier principe est formalisé dans l'équivalence. Supposons que le processus A exécute un algorithme composé de deux commandes m_0 et m_1 en séquence. Notons $m_0 \longrightarrow_{source} m_1$, l'ordre indiqué par l'algorithme. Alors, pour le processus A, $m_0 \longrightarrow_{source} m_1$ implique $m_0 \longrightarrow_{H|A} m_1$ et donc l'équivalence l'assurera dans $\overline{H}$. De plus on compare avec une extension de H pour simuler qu'une invocation pendante peut déjà avoir produit des effets.

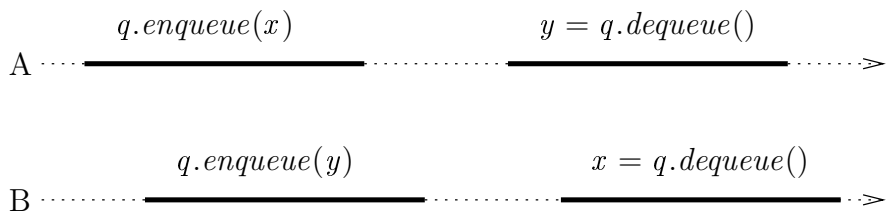


FIG. 3.6 – H_6 une trace acceptable avec plusieurs ordres valides

Ainsi la trace H_6 de la figure 3.6 est acceptable bien qu'elle ait deux ordres possibles consistant avec la trace. En effet, x peut avoir été empilé en premier, ou alors y. L'ordre de l'algorithme est bien respecté dans les deux cas, de même pour la sémantique de la queue. L'exécution est donc bien séquentiellement consistante.

Remarquons que, dans les architectures parallèles modernes (présentées dans la section 3.6), la consistance séquentielle n'est pas assurée. En effet, les effets des lectures et des écritures ne sont pas propagés immédiatement mais seulement mis en cache [62, 37, 3]. Il en résulte qu'il est nécessaire de demander explicitement cette propagation. Les instructions demandant cette propagation sont appelées *memory barriers* ou *fences*. Ce choix a été guidé par des soucis d'efficacité et de rapidité.

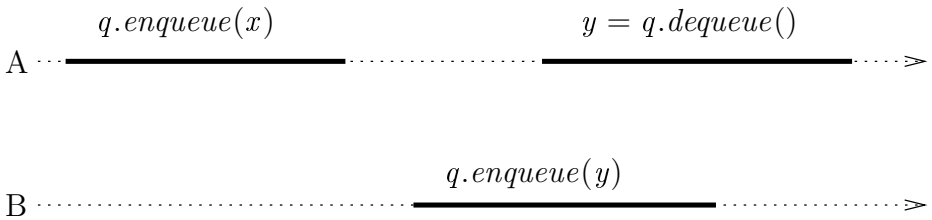


FIG. 3.7 – H_2 une trace séquentiellement consistante bien que non intuitive

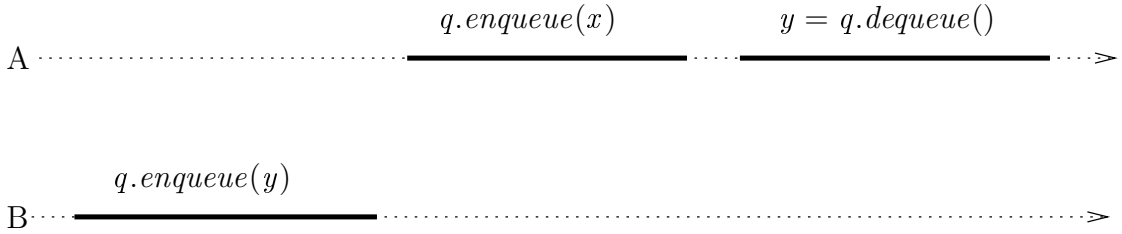


FIG. 3.8 – Une trace expliquant pourquoi H_2 est séquentiellement consistante.

Pour finir, notons que la trace H_2 en figures 3.2 et 3.7 est bien séquentiellement consistante. Nous avions pourtant dit qu'elle n'était pas acceptable et nous avions définis la consistance séquentielle pour cela. La consistance séquentielle n'est donc pas intuitive puisque l'appel à l'empilement de x se termine avant celui de y. Mais comme ces deux appels ne sont pas liés par l'ordre du programme, on peut réordonner les deux appels comme le montre la figure 3.8. De plus la consistance séquentielle n'est pas une méthode compositionnelle, en plus de ne pas simplifier la compréhension des traces. Un exemple de telle composition est sur la figure 3.9.

Dans cet exemple, la sous-trace $H_7|p$ de H_7 contenant uniquement les appels concernant la queue p est séquentiellement consistante. En effet, une trace $\overline{H_7|p}$ expliquant les effets commence par effectuer le *enqueue* sur p par B, puis les opérations faites par A sur p. En supposant que p soit initialement vide, on empile d'abord y puis x, et enfin on dépile y. Pour q, ce serait d'abord mise de x par A dans la queue, puis de y, et enfin on dépile x. Mais lorsque l'on compose les deux, on ne peut pas appliquer ces deux

explications puisqu'elles imposeraient d'inverser l'ordre des opérations effectuées par B. La composition n'a plus alors d'explication et n'est donc pas séquentiellement consistante.

Nous allons donc définir une autre notion, la linéarisabilité.

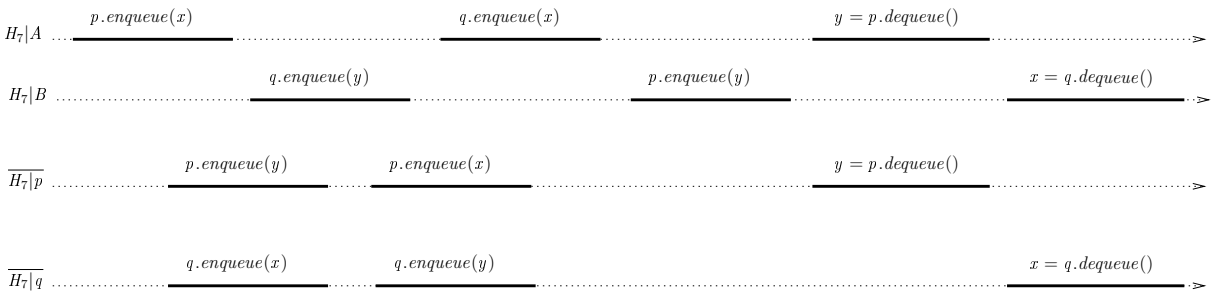


FIG. 3.9 – Deux traces séquentiellement consistantes $H_7|p$ et $H_7|q$ ne se composent pas en une trace H_7 séquentiellement consistante. Les traces $\overline{H_7|p}$ et $\overline{H_7|q}$ expliquent pourquoi $H_7|p$ et $H_7|q$ sont séquentiellement consistantes. Notez que l'inversion des empilements est autorisée puisqu'ils ne sont pas liés par le code source.

3.5 Linéarisabilité

Comme nous avons pu le voir, la consistance séquentielle n'est pas une notion suffisamment "forte" pour être intuitive et n'est pas non plus compositionnelle. Donc au lieu de dire que les effets des opérations doivent suivre l'ordre donné par l'algorithme, nous allons dire que l'effet d'une opération doit apparaitre comme atomique et avoir lieu entre l'invocation et le retour de celle-ci. Ainsi la trace H_2 de la figure 3.7 n'est pas linéarisable puisque les deux opérations d'empilement n'ont pas d'intervalle commun. La linéarisabilité (*aka* consistance atomique) fait que l'ordre temporel des opérations est respecté.

Définition 12 *Une trace valide H est dite linéarisable si, et seulement si, elle a une extension H' et s'il existe une trace séquentielle $\overline{H}$ telles que :*
- *complete(H') est équivalent à $\overline{H}$,*
- *si un appel $m_0 = (i_o, r_o)$ précède un autre appel $m_1 = (i_1, r_1)$ dans H alors il en est de même dans $\overline{H}$.*

La deuxième condition impose de garder l'ordre originel. Par contre si deux appels se chevauchent, alors il n'y a pas d'ordre imposé. Par ailleurs il est clair que la linéarisabilité implique la consistance séquentielle. L'inverse n'est pas vrai comme on l'a vu pour la trace H_2.

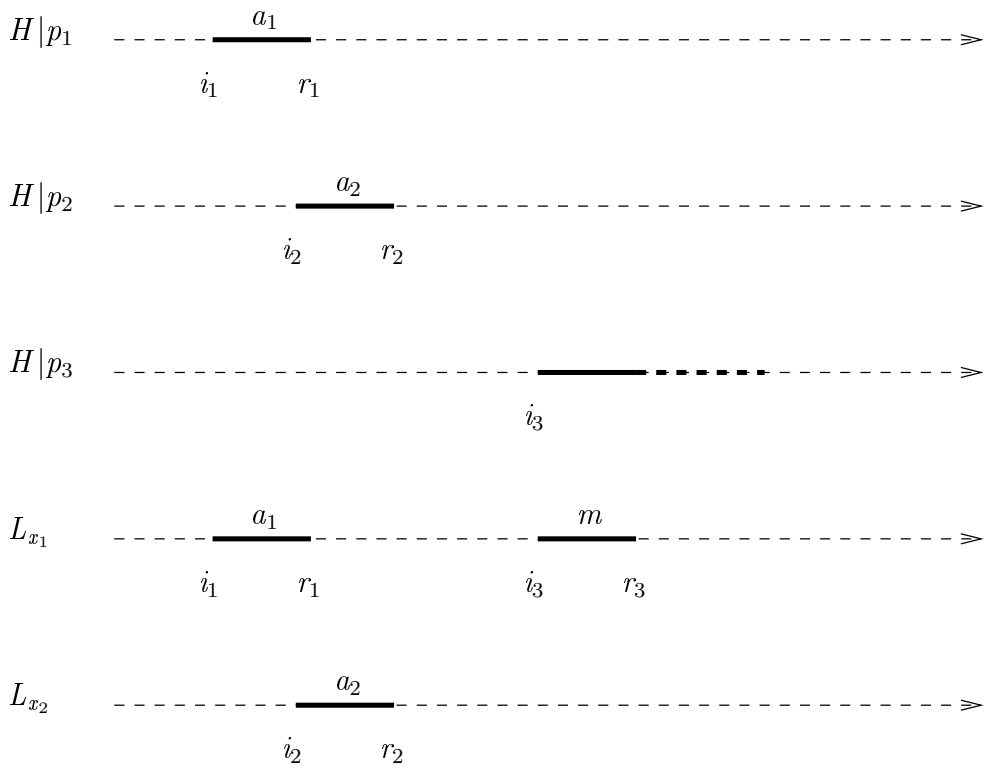
La propriété de linéarisabilité est généralement démontrée par l'existence de *points de linéarisabilité*. Un point indique le moment de l'algorithme où l'effet a lieu. Pour un algorithme à base de verrous, il s'agit généralement de la section critique. Pour les autres cas, il existe généralement un ou plusieurs points de linéarisabilité potentiels. Par exemple, dans l'implémentation du dépilement d'une file d'attente, il peut s'agir du moment où le pointeur vers le premier élément est mis à jour. Certains points de linéarisabilité peuvent dépendre de l'historique ou d'évènements futurs ce qui complique les preuves.

La linéarisabilité est compositionnelle. C'est-à-dire que la composition de deux structures de données linéarisables l'est aussi :

Théorème 13 *Une trace H est linéarisable si, et seulement si, pour tout objet x, $H|x$ est linéarisable.*

Preuve (idée). Pour chaque objet x, prenons une linéarisation L_x de $H|x$. Soit R_x les réponses aux invocations pendantes rajoutées dans L_x. Soit H' la trace construite en ajoutant à H tous les réponses des R_x. Les éléments de cette preuve sont illustrés par la figure 3.10. Le raisonnement se fait par induction sur le nombre d'appel dans H'.

Si H' contient un seul appel, alors évidemment H est linéarisable. Si H' contient moins de k appels $(k > 1)$, considérons, pour chaque objet x, le dernier appel de $H'|x$. Un de ces appels, noté m, est



$H = [i_1, i_2, r_1, r_2, i_3]$

$H|x_1 = [i_1, r_1, i_3]$ $L_{x_1} = [i_1, r_1, i_3, r_3]$

$H|x_2 = [i_2, r_2] = L_{x_2}$

$R_{x_1} = [r_3]$ $R_{x_2} = [\,]$

$H' = [i_1, i_2, r_1, r_2, i_3, r_3]$

$G' = [i_1, i_2, r_1, r_2]$

$S' = [i_1, r_2, i_2, r_2]$

FIG. 3.10 – Illustration des éléments du théorème 13.

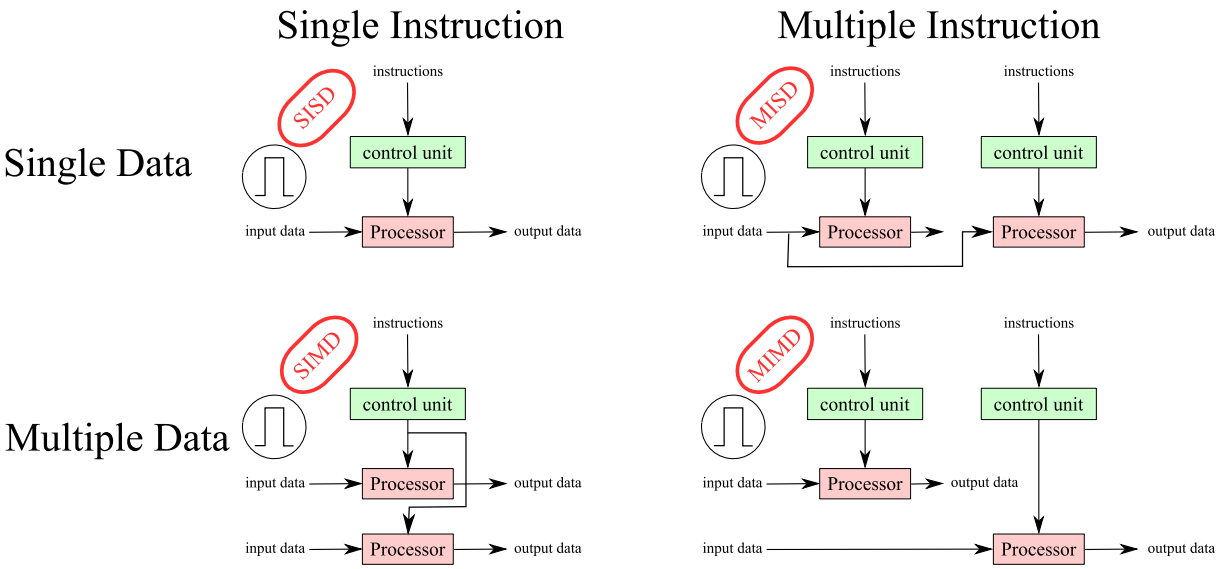


FIG. 3.11 – Taxonomie de FLYNN.

maximal en regard de l'ordre $\longrightarrow_H$. C'est-à-dire qu'il n'y a pas de m' tel que $m \longrightarrow_H m'$. Soit G' la trace définie comme H' auquel on a soustrait m : $G' \triangleq H' - \{m\}$. Comme m est maximal, H' est équivalente à $G' \cdot m$. Par l'hypothèse d'induction G' est linéarisable en S'. H' et H sont alors linéarisables en $S' \cdot m$. La réciproque est triviale. □

La consistance séquentielle et la linéarisabilité [32] sont donc deux notions importantes car simplifiant le raisonnement sur les programmes. Mais elles sont limitées par l'architecture des machines. Les deux sections suivantes vont donc s'attacher à décrire les architectures existantes et leurs capacités.

3.6 Modèles de mémoire

Pour comprendre complètement un programme concurrent, il faut comprendre l'architecture des machines. En effet, il est possible d'analyser un programme selon plusieurs niveaux d'abstraction. Le but de cette section est donc de justifier l'utilisation du modèle de mémoire partagée afin d'analyser la correction. Mais elle montrera aussi pourquoi ce ne serait pas suffisant pour vérifier l'efficacité (au sens vitesse d'exécution).

FLYNN donna la première taxonomie de l'architecture des ordinateurs dans [22]. Il différencie simplement les architectures en fonction du nombre d'entrées de données et d'instructions. Tous les flux sont synchrones. La figure 3.11 synthétise les quatre catégories possibles :

Single Instruction, Single Data c'est le cas de l'ordinateur uni-processeur classique ;

Single Instruction, Multiple Data c'est le cas où une instruction est à effectuer sur plusieurs données, par exemple dans les cartes graphiques (GPU) ;

Multiple Instructions, Single Data architecture peu courante effectuant plusieurs instructions sur une même donnée ;

Multiple Instructions, Multiple Data ce cas recouvre toutes les machines dites "parallèles".

Cette taxonomie permet de décomposer sur le type de parallélisme (data/instruction) permis ou non. On peut la compléter en indiquant si les flux de données et d'instructions sont synchronisés ou non :

Single Program, Multiple Data c'est un MIMD fonctionnant comme un SIMD mais sans la synchronisation. C'est-à-dire que chaque processus exécute le même programme mais chacun à son rythme ;

Multiple Program, Multiple Data c'est du parallélisme sans contrainte donc un MIMD sans synchronisation.

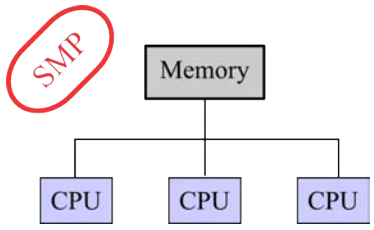


FIG. 3.12 – Symmetric Multiprocessing (SMP).

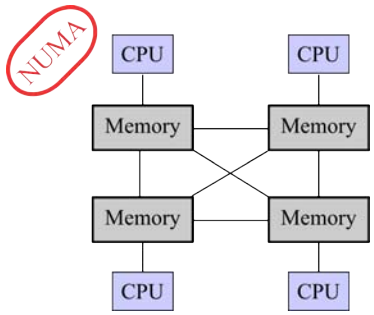


FIG. 3.13 – Non-Uniform Memory Access (NUMA).

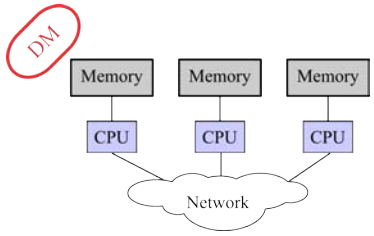


FIG. 3.14 – Distributed memory (DM).

La taxonomie de Flynn n'indique pas d'où viennent les instructions et les données. Cette lacune est importante dans notre cas pour l'étude des systèmes. Elle influe en effet sur la modélisation. Cette thèse se place clairement dans le cas du MPMD –le MIMD étant le cas particulier synchronisé–. Il s'agit donc de savoir comment est organisée la mémoire. La vision la plus simple est la vision *Symmetric Multiprocessing* (SMP). Comme représenté sur la figure 3.12, il s'agit de plusieurs *Central Processing Unit* (CPU), un CPU étant constitué d'une unité de contrôle et d'un processeur, accédant à une mémoire unique. Cela a été l'idée première lors de la conception de machine parallèle. Mais pour des raisons matérielles et de performance, les CPUs possèdent un cache mémoire. On a donc plus ou moins réparti la mémoire aux CPUs [3, 69]. On se retrouve donc dans le cas de la figure 3.13. La mémoire composite peut alors être cohérente ou non en fonction de la gestion du cache. A l'extrême on trouve une mémoire virtuelle physiquement répartie sur plusieurs machines relié par un bus ou un réseau. C'est le cas représenté sur la figure 3.14. Dans les deux derniers cas, les performances diffèrent en fonction de la localisation de la partie accédée. Pour une étude de la correction, il est important de connaître si la mémoire est cohérente, et pour une étude des performances, il faut connaître la répartition.

Au vu de ces différentes architectures, nous avons retenu l'organisation classique de CPU possédant une mémoire locale, et accédant tous à une mémoire globale (MPMD+NUMA ou MPMD+DM). Chaque processus suit un programme déterminé à son rythme. L'exécution d'une instruction est supposée atomique. Par ailleurs nous ne supposerons rien sur l'ordonnancement des processus. Une exécution du système sera donc un entrelacement des exécutions de chacun des processus (*interleaving semantic*).

3.7 Primitives

Pour pouvoir profiter pleinement du parallélisme, il faut limiter les synchronisations inter-processus. Nous avons vu dans le chapitre précédent comment synchroniser ceux-ci par l'exclusion mutuelle. Mais cette technique fait perdre beaucoup de parallélisme en n'autorisant pas la concurrence de certaines opérations, et ce quelque soit le niveau de granularité choisi. C'est pourquoi les processeurs fournissent des primitives de plus haut niveau permettant la synchronisation. Parmi ces primitives, on trouve par exemple *Fetch-And-Add* (FAA), *Compare-And-Swap* (CAS), *Load-Link* (LL)/*Store-Conditionnal* (SC), et *Test-And-Set* (TAS). Ce sont ces primitives qui permettent de construire des algorithmes sans verrou et donc les structures de données qui nous intéressent. Certains algorithmes partagent temporellement l'accès à une ressource, assurant l'exclusion mutuelle par la gestion des délais d'exécutions. Herlihy a montré dans [30] qu'il existe une hiérarchie de ces primitives dans le sens où on peut construire certaines primitives avec d'autres. Ainsi on peut construire TAS avec CAS. Malheureusement CAS est aussi plus difficile à implémenter matériellement et a donc un coût d'utilisation plus fort.

Cette hiérarchie (*aka wait-free hierarchy*) est basée sur le problème du consensus. Le consensus consiste pour n processus à se mettre d'accord sur une valeur commune. Pour être consistant, les processus doivent décider de la même valeur. Pour être valide, la valeur choisie doit être une des valeurs initialement proposées par l'un des processus. Les processus suivent un protocole (ou algorithme) utilisant certaines primitives. On forme la hiérarchie en calculant, pour un ensemble de primitives données, $max(n)$ tel que le problème soit résolu. Ce nombre est appelé *consensus number*. La hiérarchie a donc une infinité d'échelons. Voici un extrait de cette hiérarchie :

1	*read*, *write*
2	FAA
$2m$	$write_m$
∞	CAS, LL/SC, *swap*

```
word_t fetchAndAdd(word_t* addr) {
  word_t value = *addr;
  (*addr)++;
  return value;
}
```

Fig. 3.15 – Définition séquentielle de FAA

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

FIG. 3.16 – Définition séquentielle de CAS_1

read et *write* sont les opérations classiques de lecture et écriture. $write_m$ est l'opération qui atomiquement modifie m adresses mémoire. *swap* interchange atomiquement les valeurs de 2 adresses mémoires. Cette hiérarchie considérée comme "non robuste" peut être affinée [7, 38, 75, 71]. Elle est non robuste car certaines combinaisons de primitives leur donnent un pouvoir plus grand [48]. Par ailleurs cette hiérarchie ne considère que des aspects théoriques mais pas forcément d'efficacité ou de consommation mémoire [20].

```
word_t fetchAndAddCas1(word_t* addr) {
  word_t value;
  word_t t,old;
  do {
    value = *addr;
    t = value+1;
    old = CAS1(addr, value, t);
  } while (old != value);
  return value;
}
```

FIG. 3.17 – Implémentation de FAA à partir de CAS_1

```
word_t fetchAndAddLLSC(word_t* addr) {
  word_t value;
  word_t t,old;
  do {
    value = ll(addr);
    t = value+1;
  } while (!sc(addr, t));
  return value;
}
```

FIG. 3.18 – Implémentation de FAA à partir de LL/SC

Après avoir vu la classification, voici une description succincte de plusieurs primitives. La définition séquentielle de FAA est donnée en figure 3.15. Notons que dans certaines implémentations, l'incrément est donné en paramètre. Comme la hiérarchie l'indique on peut aussi l'implémenter avec CAS_1 et LL/SC. Les implémentations sont données dans les figure 3.17 et 3.18.

CAS est en fait une famille d'opérations dont souvent seul CAS_1 est fourni par le processeur (instruction CMPXCHG pour les Intel Pentium™). CAS_1 est défini comme étant équivalent à l'exécution atomique du programme de la figure 3.16. Elle compare la valeur pointée par le premier argument avec le deuxième argument. S'il y a égalité, elle affecte à la valeur référencée par le premier pointeur la valeur du troisième argument sinon elle la laisse inchangée. CAS_1 comme d'autres primitives a un coût important du fait qu'elle impose une barrière (*fence*) aux contrôleurs de cache mémoire.

L'opération CAS_n peut être implémentée avec CAS_1 comme le font Harris et al [28]. Comme on le verra dans les chapitres suivants, l'implémentation utilise plusieurs astuces. Ça complexité est telle qu'on ne peut en vérifier avec le model-checking qu'une instance simplifiée et réduite à 3 processeurs. De plus, la justification de la réduction n'est pas triviale. Une preuve aurait l'avantage de justifier formellement les arguments avancés.

LL et SC sont deux opérations utilisées conjointement (implémentées sous les noms LDL_L/STL_C, LDQ_L/STQ_C, LWARX/STWCX, et LDREX/STREX selon les architectures). LL, aussi appelée Load-And-Reserve, permet de lire la valeur d'un pointeur mémoire. SC permet de modifier cet emplacement. S'il y a eu une modification entre temps, SC échouera, et ce même si la valeur stockée est identique. Ce comportement évite donc le problème ABA. C'est le problème qui survient avec CAS_1, quand on lit une valeur a, puis on calcule une valeur en fonction pendant que la mémoire est changée en b, puis en a et qu'enfin on vérifie que la valeur est toujours a. Si on trouve a, cela devrait vouloir dire que la valeur n'a pas été modifiée, ce qui n'est pas forcément le cas. Par exemple, pour une liste chaînée, si le noeud de tête est identique, la liste n'a pas été modifiée. Or, comme la mémoire est finie, il se peut que l'on ait réutilisé ce noeud introduisant alors une *race condition*. Dans la pratique, on peut réduire la probabilité d'apparition du problème en faisant attention à l'ordre des allocations mémoires.

Chapitre 4

Modèles formels

Dans ce chapitre, nous allons passer en revue l'état de l'art en matière de méthodes formelles pour la preuve d'algorithmes concurrents. D'autres méthodes auraient pu être abordées, entre autres la réduction de Lipton ou l'inférence de types pour l'atomicité [46, 21, 45, 16], mais ne le seront pas, étant trop restreintes dans leurs applications ou bien étant reprises dans les méthodes citées. Après une courte introduction à chacune des techniques ou idées, nous verrons comment appliquer celles-ci à un petit exemple. Cet exemple sera aussi traité par notre technique dans le chapitre 7. Celui-ci consiste à remplacer l'affectation $x = f(x)$ par un algorithme utilisant la primitive CAS_1 présentée dans la section 3.7. f est considérée comme une fonction sans effet de bord et qui n'est pas fournie atomiquement par le processeur. Cet algorithme est donné dans la figure 4.1. Cet algorithme, bien que simple, est intéressant car c'est un schéma d'algorithme très classique. Ainsi en posant $f(x) \triangleq x + 1$, on retrouve l'implémentation de FAA donnée au chapitre précédent.

```
void cas1Assign(word_t *x) {
  word_t r, c, v;
  do {
    r = *x;
    /* The following calculation is not atomic
       but it is side-effect free */
    v = f(r);
    /* invariant : v == f(r) */
    c = CAS1(x, r, v);
  } while (c != r);
}
```

FIG. 4.1 – Affectation $x = f(x)$ avec CAS_1

Que signifie remplacer un algorithme par un autre ? Il faut voir cela comme on remplacerait un composant électronique par un autre. C'est-à-dire que, considérés comme des boites noires, il y a une équivalence observationnelle entre les deux. Autrement dit, observé de l'extérieur, on ne peut distinguer l'un de l'autre à l'usage. En adaptant la définition du principe de substitution de Liskov [47], nous pourrions dire que si $q(Sys_1)$ est une propriété démontrable pour tout système Sys_1 utilisant l'algorithme $algo_1$, alors $q(Sys_2)$ doit l'être pour tout système Sys_2 tel que Sys_2 est le système obtenu en remplaçant $algo_1$ par $algo_2$ dans Sys_1.

4.1 Assume-guarantee

4.1.1 Introduction

La méthode *assume-guarantee* a été introduite par Jones et al. [39] comme une extension de la logique de Hoare. Cette dernière est un système formel fournissant un ensemble de règles permettant de raisonner à propos de la correction d'un programme impératif. La logique a été initialement décrite dans [34]. Ce système est basé sur les *triplets de* Hoare. Un triplet décrit l'effet d'un programme sur l'état du système. Ainsi un triplet $\{\phi\}\mathrm{P}\{\psi\}$ indique que toute exécution (se terminant) du programme P, commençant dans un état où ϕ est vérifiée, amène à un état où ψ l'est. ϕ est appelée pré-condition, ψ post-condition. ϕ et ψ sont deux assertions dans une théorie appropriée aux données manipulées par le programme. Cette théorie pourrait, par exemple, être la logique des prédicats étendue par la théorie de l'arithmétique sur les entiers. $\{x = 42\}x{+}{+}\{x = 43\}$ est un exemple de triplet valide.

La logique est constituée d'axiomes et de règles d'inférence. Les axiomes sont les triplets indiquant l'effet des commandes de base. Les règles permettent de composer ces commandes en un programme. Voici quelques règles :

$$\{\phi[x/E]\}x := E\{\phi\} \text{ Affectation}$$

$$\frac{\phi \Rightarrow \phi_1 \qquad \{\phi_1\}\mathrm{P}\{\psi_1\} \qquad \psi_1 \Rightarrow \psi}{\{\phi\}\mathrm{P}\{\psi\}} \text{ Conséquence}$$

$$\frac{\{\phi\}\mathrm{P}_1\{\chi\} \qquad \{\chi\}\mathrm{P}_2\{\psi\}}{\{\phi\}\mathrm{P}_1;\mathrm{P}_2\{\psi\}} \text{ Séquence}$$

La logique de Hoare n'est pas très adaptée au raisonnement sur les programmes parallèles. En effet, il est facile de raisonner sur les programmes $a \mathrel{+}= 2$ et $a \mathrel{+}= 3$. Mais on ne peut plus raisonner sur $a{+}{=}2 \parallel a{+}{=}3$. Les deux programmes interagissent l'un sur l'autre au travers de la variable a. Pour pouvoir raisonner sur de tels programmes, il faut aussi modéliser l'environnement. L'environnement pourrait être modélisé par ce qu'il fait. Mais dans les techniques présentées ci-après, on le caractérise plutôt par ce qu'on lui autorise de faire. Ainsi le triplet $\{a = 0\}a \mathrel{+}= 2\{a = 2\}$ est valide dans un environnement ne pouvant pas modifier la variable a ($a' = a$). Pour faciliter la composition, on a aussi besoin d'une formule caractérisant ce que peut faire le programme, ici $a' = a + 2$. Ainsi peut-on composer le raisonnement avec le programme $b \mathrel{+}= 2$, puisqu'il ne change pas a. Le triplet $\{a = 0\}a \mathrel{+}= 2 \parallel b \mathrel{+}= 2\{a = 2\}$ est toujours valide. Par contre, pour composer le programme $a \mathrel{+}= 2$ avec lui-même il faudrait alors vérifier que $a' = a + 2 \implies a' = a$. Bien évidemment ceci n'est pas valide, on ne peut donc rien en déduire sur la composition dans cet exemple.

Les travaux de Owicki-Gries [63] et de Assumption-Commitment [56] sont l'origine de ces idées [88], et ont aboutis à la création des principes du *Assume/Guarantee*. On étend les triplets de Hoare par deux assertions sur ce que garantit le programme et ce qu'il suppose. Ainsi, dans $\langle r, g\rangle\{\phi\}\mathrm{P}\{\psi\}$, r (*rely* ou *assume*) décrit ce que suppose le programme P sur l'environnement et g (*guarantee*) ce qu'il garantit. Ces formules caractérisant les effets d'un programme, elles sont généralement écrites dans la logique des prédicats à deux états, x' représentant la valeur de la variable x après l'effet. ϕ et ψ sont les pré- et post-conditions comme dans les triplets de Hoare.

Soit P un programme atomique caractérisé par la formule à deux états p,

$$\frac{\phi \wedge r \Rightarrow \phi' \qquad \dfrac{\phi \wedge p \Rightarrow \psi'}{\{\phi\}\mathrm{P}\{\psi\}} \qquad \psi \wedge r \Rightarrow \psi' \qquad p \Rightarrow g}{\langle r, g\rangle\{\phi\}\mathrm{P}\{\psi\}}$$

Fig. 4.2 – Passage des triplets de Hoare aux conditions de *Assume/Guarantee*.

Prenons par exemple le triplet de Hoare $\{a = 0\}a \mathrel{+}= 2\{a = 2\}$ celui-ci est trivial de par la règle de l'affectation. Mais dans un environnement concurrent, pour qu'il reste valide, il ne faut pas que les autres processus (donc les autres programmes) modifient a. De plus, le programme $a \mathrel{+}= 2$ ne modifie pas les

$$\frac{\langle r_1, g_1\rangle\{\phi\}\mathrm{P}_1\{\chi\} \qquad \langle r_2, g_2\rangle\{\chi\}\mathrm{P}_2\{\psi\}}{\langle r_1 \wedge r_2, g_1 \vee g_2\rangle\{\phi\}\mathrm{P}_1;\mathrm{P}_2\{\psi\}}\ \textsc{Séquence}$$

$$\frac{\langle r_1, g_1\rangle\{\phi_1\}\mathrm{P}_1\{\psi_1\} \qquad \langle r_2, g_2\rangle\{\phi_2\}\mathrm{P}_2\{\psi_2\} \qquad g_2 \Rightarrow r_1 \qquad g_1 \Rightarrow r_2}{\langle r_1 \wedge r_2, g_1 \vee g_2\rangle\{\phi_1 \wedge \phi_2\}\mathrm{P}_1 \| \mathrm{P}_2\{\psi_1 \wedge \psi_2\}}\ \textsc{Parallèle}$$

$$\frac{\phi \Rightarrow \phi_1 \qquad r \Rightarrow r_1 \qquad \langle r_1, g_1\rangle\{\phi_1\}\mathrm{P}\{\psi_1\} \qquad g_1 \Rightarrow g \qquad \psi_1 \Rightarrow \psi}{\langle r, g\rangle\{\phi\}\mathrm{P}\{\psi\}}\ \textsc{Conséquence}$$

FIG. 4.3 – Règles principales de la méthode *assume-guarantee*.

variables b et c. Il garantit donc $b' = b \wedge c' = c$. Nous en déduisons donc :

$$\langle a' = a, b' = b \wedge c' = c\rangle\{a = 0\}a \mathrel{+}= 2\{a = 2\}$$

De manière générale, soit P un programme atomique caractérisé par la formule à deux états p tel que $\phi \wedge p \Rightarrow \psi'$ alors $\{\phi\}\mathrm{P}\{\psi\}$ est un triplet de HOARE valide. Pour pouvoir passer au quintuplet du *Assume/Guarantee*, il faut que la pré- et post-condition soient stables sous la condition de supposition du programme. En effet, les triplets de HOARE ne permettent de raisonner que du passage d'un état à un autre. Il ne faut donc pas que les autres processus puissent modifier la caractérisation de cet état, d'où cette condition. Il faut aussi que le programme vérifie bien la condition de garantie. Les conditions formelles sont données en figure 4.2. Les règles de la méthode se déduisent alors facilement. Voir la figure 4.3 pour les règles principales.

Si l'on compose maintenant l'exemple précèdent avec le programme $b \mathrel{+}= 3$, on peut en déduire :

$$\frac{\langle a' = a, b' = b \wedge c' = c\rangle\{a = 0\}a \mathrel{+}= 2\{a = 2\} \qquad \langle b' = b, a' = a \wedge c' = c\rangle\{b = 0\}b \mathrel{+}= 3\{b = 3\}}{\langle a' = a \wedge b' = b, c' = c\rangle\{a = 0 \wedge b = 0\}a \mathrel{+}= 2 \| b \mathrel{+}= 3\{a = 2 \wedge b = 3\}}$$

Que peut-on dire du programme $a \mathrel{+}= 2 \| a \mathrel{+}= 3$? Peut-on en déduire le quintuplet suivant ?

$$\langle ?, ?\rangle\{a = 0\}a \mathrel{+}= 2 \| a \mathrel{+}= 3\{a = 5\}$$

La méthode est incomplète si l'on n'utilise pas les variables fantômes et/ou de prophéties. L'exemple ci-après montre que certains triplets corrects ne pourraient être prouvés. Ces variables sont des variables auxiliaires servant à décrire l'exécution d'une action ou encore des valeurs futures mais qui n'existe pas dans le programme réel. Sans elles, la méthode n'est pas complète [1]. Pour pouvoir prouver cet exemple, il est nécessaire de faire appel à deux variables fantômes, une d_i décrivant l'incrémentation faite par le processus lui-même, et une autre d_j $(i \neq j)$ décrivant l'incrément effectué par les autres processus. Il convient donc de toujours vérifier que $a = d_i + d_j$. On utilise donc le programme qui incrémente d_i en même temps que a. On obtient le triplet de HOARE suivant :

$$\{a = d_i + d_j \wedge d_i = 0\}a \mathrel{+}= v \| d_i = v\{a = d_i + d_j \wedge d_i = v\}$$

D'après la règle de la figure 4.2, on obtient

$$\begin{array}{lll} \langle & d_i' = d_i \wedge (a = d_i + d_j \Rightarrow a' = d_i' + d_j') & , \\ & d_j' = d_j \wedge (a = d_i + d_j \Rightarrow a' = d_i' + d_j') & \rangle \\ \{ & a = d_i + d_j \wedge d_i = 0 & \} \\ & a \mathrel{+}= v \| d_i = v & \\ \{ & a = d_i + d_j \wedge d_i = v & \} \end{array}$$

En utilisant la règle de composition et de conséquence, on obtient alors :

$$\begin{array}{llll}
\langle & d_0' = d_0 \wedge (a = d_0 + d_1 \implies a' = d_0' + d_1') & , & \\
& d_1' = d_1 \wedge (a = d_0 + d_1 \implies a' = d_0' + d_1') & \rangle & \\
\{ & a = d_0 + d_1 \wedge d_0 = 0 & \} & \\
& a \mathrel{+}= 2 \| d_0 = 2 & & \\
\{ & a = d_0 + d_1 \wedge d_0 = 2 & \} &
\end{array}
\quad
\begin{array}{llll}
\langle & d_1' = d_1 \wedge (a = d_0 + d_1 \implies a' = d_0' + d_1') & , & \\
& d_0' = d_0 \wedge (a = d_0 + d_1 \implies a' = d_0' + d_1') & \rangle & \\
\{ & a = d_0 + d_1 \wedge d_1 = 0 & \} & \\
& a \mathrel{+}= 3 \| d_1 = 3 & & \\
\{ & a = d_0 + d_1 \wedge d_1 = 3 & \} &
\end{array}$$

$$\begin{array}{lll}
\langle & d_0' = d_0 \wedge d_1' = d_1 \wedge (a = d_0 + d_1 \implies a' = d_0' + d_1') & , \\
& a = d_0 + d_1 \implies a' = d_0' + d_1' & \rangle \\
\{ & a = d_0 + d_1 \wedge d_0 = 0 \wedge d_1 = 0 & \} \\
& a \mathrel{+}= 3 \| d_0 = 2 \| a \mathrel{+}= 2 \| d_1 = 3 & \\
\{ & a = d_0 + d_1 \wedge d_0 = 2 \wedge d_1 = 3 & \}
\end{array}$$

Avec les principes du raffinement exposés dans la section 4.3, on pourra en déduire

$$\langle a' = a, \top \rangle \{a = 0\} a \mathrel{+}= 3 \| a \mathrel{+}= 2 \{a = 5\}$$

La méthode est donc compositionnelle et permet de raisonner sur une partie d'un système. La méthode a été formalisée [67] dans l'assistant à la preuve Isabelle/HOL [61, 66]. Nous allons utiliser cet outil pour la preuve de l'affectation utilisant CAS_1.

4.1.2 Exemple

Tout d'abord nous pouvons facilement montrer que

$$\langle \top, x' = f(x) \vee x' = x \rangle \{\top\} x = f(x) \{\top\}$$

est un quintuplet valide. Ce quintuplet atteste que chaque pas intermédiaire de ce programme affecte $f(x)$ à x ou alors laisse x inchangé. Il n'indique néanmoins pas combien de fois cette affectation est réalisée, ce qui est nécessaire pour assurer la sûreté de l'algorithme. Pour faire une preuve complète, il faudrait utiliser la technique des variables dites fantômes et prouver le quintuplet suivant :

$$\langle c' = c, x' = f(x) \vee x' = x \rangle \{c = 0\} x = f(x) \parallel c{+}{+} \{c = 1\}$$

Nous allons seulement prouver que l'algorithme de la figure 4.1 vérifie le premier quintuplet car cet exemple est suffisant pour comprendre l'utilisation de la méthode.

La première étape a été de modifier le langage impératif afin de lui rajouter l'opération CAS_1. En réalité nous avons juste modifié la syntaxe et implémenté BCAS_1 qui renvoie un booléen indiquant si l'affectation a eu lieu ou non. La preuve complète en Isabelle est disponible dans l'annexe A. Par ailleurs, le concept de variables locales n'existant pas, on encode cet aspect dans l'assertion "*rely*" par le fait que les autres processus ne doivent pas les modifier.

La preuve en elle-même n'est pas très compliquée une fois que l'on "trouve" l'invariant local $v = f(r)$. En effet, on utilise la règle de séquencement avec

$$\langle r' = r \wedge v' = v \wedge c' = c, x' = x \vee x' = f(x) \rangle \{\top\} v = f(r) \{v = f(r)\}$$

et

$$\langle r' = r \wedge v' = v \wedge c' = c, x' = x \vee x' = f(x) \rangle \{v = f(r)\} c = \mathrm{CAS}_1(x, r, v) \{\top\}$$

Néanmoins, dans l'assistant à la preuve Isabelle, les preuves ne sont pas complètement automatiques à cause de l'encodage de la méthode utilisé. Par ailleurs, nous n'avons pas démontré que l'affectation n'est effectuée qu'une seule fois.

4.2 Logique de séparation

La logique de séparation [64, 83] est une logique dédiée aux raisonnement sur les programmes utilisant des pointeurs et la pile. Une formule est donc valide en regard d'une pile. La pile est alors une variable implicite des formules. La logique étend la logique propositionnelle avec :

- *emp* qui est satisfait par la pile vide (*empty*) ;
- $x \mapsto y$ satisfait par la pile dont l'unique pointeur x a pour valeur y ;
- $P \star Q$ satisfait par les piles pouvant être disjointes en deux parties, la première satisfaisant P, la deuxième Q.

L'opérateur ne $\star$ définit que des piles simples sans *aliasing*. La formule $x_1 \mapsto y_1 \star \ldots \star x_k \mapsto y_k \star true$ implique donc que tous les x_i sont différents deux à deux.

La logique de séparation est ensuite utilisée comme langage d'assertion soit dans les triplets de Hoare, soit dans la méthode *assume-guarantee*. Mais elle permet un raisonnement local grâce à des règles telles que :

$$\frac{\{P\}\mathrm{C}\{Q\}}{\{P \star R\}\mathrm{C}\{Q \star R\}}\ \textsc{Frame-Rule}$$

Cette règle correspond à l'intuition qu'un programme fonctionnant avec une pile vérifiant P peut fonctionner avec une pile plus grande laissant une partie de celle-ci inchangée. L'opérateur $\star$ permet ainsi facilement de raisonner sur des structures de données comme les listes chaînées, les arbres... à condition de connaître où la mémoire est partagée par plusieurs pointeurs.

Par ailleurs, la logique de séparation peut être appliquée aux algorithmes parallèles. La logique encode alors la propriété d'une partie de la mémoire par un programme. On peut alors composer deux programmes s'ils agissent sur deux parties distinctes de la pile :

$$\frac{\{P_1\}\mathrm{C}_1\{Q_1\} \qquad \{P_2\}\mathrm{C}_2\{Q_2\}}{\{P_1 \star P_2\}\mathrm{C}_1 \| \mathrm{C}_2\{Q_1 \star Q_2\}}\ \textsc{Parallèle}$$

Smallfoot [6] est un prouveur automatique de la logique de séparation. Une version modifiée Smallfoot-tRG a notamment été utilisé pour prouver l'exemple RDCSS [82] que l'on présente dans la section 7.4. Concernant notre petit exemple, il se prouverait de la même façon qu'en *Assume-Guarantee* car il n'y a pas à proprement parler de séparation. On n'utilise donc pas l'opérateur $\star$.

4.3 Raffinement

Le raffinement est la transformation d'un programme de haut niveau d'abstraction en un autre plus concret. Généralement le niveau le plus bas (le plus concret) est directement exécutable par la machine cible. Le raffinement peut se faire en plusieurs étapes ce qui simplifie à la fois la compréhension et les preuves. En effet, avec les méthodes actuelles, et la spécification et le programme final sont décrits dans le même langage. Le raffinement sert donc à la fois à lever un certain indéterminisme mais aussi à introduire petit à petit les détails d'implémentations.

Une spécification de haut-niveau ne doit décrire que ce qui peut être observé d'un point de vue extérieur [1]. La spécification formalise alors une boite noire. Elle ne décrit que ce qui est nécessaire. Il y a donc de nombreuses transitions indéterministes ou non définies. Le raffinement permet de documenter ces choix et de vérifier que la boite noire fonctionne toujours correctement. Si on décrit un composant électronique, on pourrait donc remplacer un composant par un autre. On est assuré du bon fonctionnement car les évènements et les états visibles de l'extérieur sont identiques.

D'un point de vue formel, le raffinement est une implication : la sémantique du programme concret impliquant celle du programme abstrait. Néanmoins le raffinement n'est utile que si l'implication se fait modulo un certain *collage*. En effet, il existe deux types de raffinement. Le premier, le raffinement de données *data refinement* sert à convertir un modèle de données abstrait en une structure implémentable. On l'utilise, par exemple, pour remplacer un ensemble par une liste chaînée. Le deuxième est le raffinement d'opération afin de les rendre implémentable. La post-condition est alors renforcée et la pré-condition

affaiblie. De plus il sert aussi à supprimer l'indéterminisme du programme abstrait. L'opération inverse du raffinement est appelée *abstraction*.

Ainsi l'opération $x' \in \{1,2,3\}$ est raffinée par $x' \in \{1,3\}$, elle-même raffinée par $x' = 1$. Dans cet exemple, l'*invariant de collage* —qui définit la relation entre les données abstraites et concrètes— est évidemment l'égalité. Par ailleurs, on passe d'une opération indéterministe à une autre déterministe. Si l'on raffine $s' = s \cup \{1\}$ par $q' = enqueue(q,1)$, l'invariant de collage fait alors le lien entre les éléments de s et les éléments de q. Un tel invariant pourrait être $s = \{e | \exists n \in \mathbb{N}. e = dequeue^n(q)\}$.

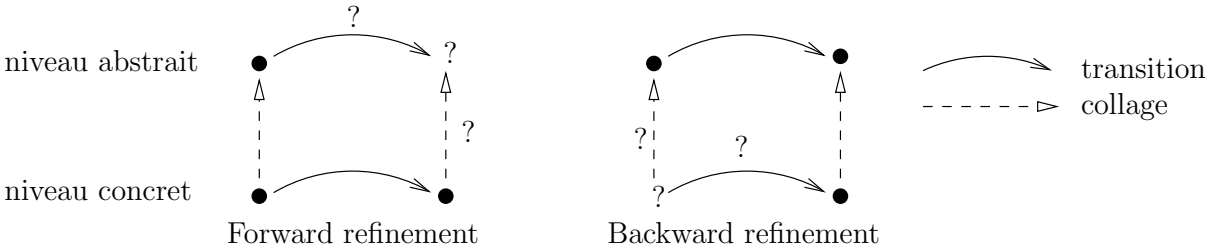


Fig. 4.4 – Deux types de simulation

Il existe différentes façons de définir le raffinement. L'idée principale est de dire que tout ce qui est permis par la spécification abstraite l'est par la spécification concrète. On peut notamment le formaliser par le *refinement calculus* [4, 57]. Une autre approche se fait via les systèmes de transition. Le raffinement est défini par rapport à la simulation d'un système par un autre [50]. Plus précisément on trouve deux sortes de simulation comme précisé sur la figure 4.4. Dans les deux cas, il existe deux états, un abstrait l'autre concret, reliés par l'invariant de collage. Dans la simulation en avant *forward simulation*, pour chaque pas concret, il faut démontrer qu'il existe un pas abstrait tel que l'état abstrait suivant vérifie l'invariant de collage après. Dans la simulation en arrière, c'est le même principe mais pour chaque pas abstrait amenant à ces états, il faut retrouver le pas concret arrivant à l'état cible. Le raffinement en arrière $\sqsubseteq_B$ ou en avant $\sqsubseteq_F$ définissent un pré-ordre. On dit alors que le programme $\underline{\mathrm{P}}$ raffine $\overline{\mathrm{P}}$, noté $\underline{\mathrm{P}} \sqsubseteq \overline{\mathrm{P}}$ si, et seulement si, $\exists \mathrm{P}. \underline{\mathrm{P}} \sqsubseteq_B \mathrm{P} \sqsubseteq_F \overline{\mathrm{P}}$.

```
typedef struct {
} Roulette1;

int run(Roulette1 &r) {
  // randint returns a random int number between 0 and 36.
  int v = randint(0,36);
  return v;
}
```

Fig. 4.5 – Une roulette renvoyant des résultats aléatoires à la demande.

L'utilisation des deux simulations est nécessaire. Par exemple, les algorithmes présentés en figure 4.5 et 4.6 sont deux implémentations d'une roulette. Du point de vue observationnel, les deux roulettes sont identiques. En voyant fonctionner une roulette, on ne peut pas dire quelle implémentation elle utilise. Ces deux algorithmes sont équivalents. On devrait donc pouvoir prouver que l'un raffine l'autre et réciproquement. Or ce n'est pas le cas si l'on n'utilise que la simulation en avant. Avec celle-ci, on ne peut que démontrer que la roulette 2 raffine la roulette 1. La simulation en avant est donc incomplète [29, 25, 14, 17]. On pourrait dire de même pour la simulation en arrière. Seul donc la combinaison de ces deux simulations permet de définir le raffinement. Dans la pratique, la simulation en avant permet de prouver de nombreux cas. Toutes les méthodes n'utilisent donc pas la combinaison des deux simulations. Les deux méthodes présentées ci-après sont basées sur le raffinement. La simulation a aussi été utilisée pour montrer directement la linéarisabilité comme dans [85].

```
typedef struct {
  int nextValue;
} Roulette2;

Roulette2* newRoulette2() {
  Roulette2* r = (Roulette2*)malloc(size_of(Roulette2));
  r->nextValue = randint(0,36);
  return r;
}

int run(Roulette2 &r) {
  // randint returns a random int number between 0 and 36.
  int old = r.nextValue;
  r.nextValue = randint(0,36);
  return old;
}
```

FIG. 4.6 – Une roulette prévoyant le prochain résultat (aléatoire) à chaque lancement. Un point de vue extérieur ne permet pas de la distinguer de la roulette 1.

4.4 B

B [2] est une méthode formelle destinée à écrire des programmes à partir de spécifications formelles. Descendante de Z [76], elle se base sur la notion de machine abstraite. La méthode couvre l'écriture de la spécification, le raffinement successif de celle-ci, jusqu'à la génération de code. Bien entendu chaque raffinement génère un certain nombre d'*obligations de preuves* à vérifier.

La méthode B a évoluée en plusieurs langages chacun lié à un outil :

B classique définit dans le "B book" [2] et supporté par l'atelier B ;

B événementiel substituant la notion d'action par la notion d'évènement, supporté par *Click'n Prove* - B4free [10] ;

B# qui étend le B événementiel par des éléments de Z, supporté par Rodin.

Le langage décrivant les machines est basé sur les substitutions généralisées. Une substitution décrit un changement d'état. Les substitutions de base du B classique sont :

substitution neutre SKIP qui aboutit au même état ;

substitution simple ou multiple $x, y := E, F$, ou encore $x := E \| y := F$, qui affectent simultanéement les valeurs des expressions E et F à x et y ;

substitution devient tel que $x :\in S$ qui affecte à x une valeur indéterminée de l'ensemble S.

On peut composer des substitutions avec les opérateurs suivants :

séquencement $G; H$ qui applique successivement G puis H ;

pré-condition $P|G$ qui assure que l'état suivant sera correct uniquement si G est appliquée dans un état où la pré-condition P est vérifiée ;

garde $P \Rightarrow G$, qui ne peut être appliquée que dans un état où P est satisfaite ;

choix borné $G \square H$ qui applique de manière indéterministe G ou H ;

choix non borné $@z.G$ qui applique G pour une valeur non déterminée de z.

La correction s'effectue par la technique d'un invariant global écrit dans la théorie des ensembles. Ainsi chaque évènement doit assurer la validité de l'invariant après son action. Les obligations de preuves sont donc évidemment que l'état initial vérifie l'invariant, et que tous les évènements font passer d'un état vérifiant l'invariant à un autre le vérifiant également. Un modèle B est donc composé de, au moins, trois parties : invariant, initialisation, évènements.

L'exemple de la spécification de l'affectation $x = f(x)$ servira de fil conducteur à la découverte de B. L'exemple complet avec le raffinement utilisant CAS_1 est présenté en Annexe B.

Un modèle B commence toujours par le mot clef **MODEL** suivi du nom du modèle. On déclare ensuite les ensembles, les variables et les constantes. Un ensemble peut être énuméré comme c'est le cas ici pour l'ensemble $PLACE$.

MODEL
$AbstractSpec$
SETS
$VAL; PLACE = \{PA1, PA2\}; PROCESS$
VARIABLES
x, pp
CONSTANTS
f

Les propriétés ou axiomes à propos des constantes et des ensembles sont ensuite définis. Ici, la propriété concerne le typage.

PROPERTIES
$f \in VAL \rightarrow VAL \wedge$
$\forall v.(v \in VAL)$

Il y a un invariant unique par modèle B. C'est donc l'élément central de celui-ci. Au niveau abstrait, il est souvent simple et concerne, là encore, principalement le typage. Dans le cas présent, on dit que la variable x est de type VAL. Elle sera globale à tous les processus. pp est de type $PROCESS \rightarrow PLACE$. C'est une des façons d'encoder le fait qu'une variable est locale à un processus. De plus, elle est de type $PLACE$. Elle correspond donc au compteur ordinal (*program counter*). $PLACE$ est donc l'ensemble des prédicats de place qui vont nous servir à décrire l'avancement de l'algorithme.

INVARIANT
$x \in VAL \wedge$
$pp \in PROCESS \rightarrow PLACE$

On décrit ensuite l'état initial puis les évènements. Il y a deux principaux évènements dans notre modèle. Le premier *rely* définit l'action que peuvent faire les autres processus sur l'environnement (constitué ici uniquement de la variable x). En l'occurrence, on substitue à x une valeur v telle que $v \in VAL$. Le deuxième correspond à l'affectation en question. On substitue donc à la fois $f(x)$ à x, et $pp(p)$ passe de $PA1$ à $PA2$.

INITIALISATION
$x :\in VAL \parallel$
$pp :\in PROCESS \rightarrow PLACE$
EVENTS
$rely =$ **ANY** v **WHERE**
$v \in VAL$
THEN
$x := v$
END ;
$assign =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp(p) = PA1$
THEN
$x := f(x) \parallel$
$pp(p) := PA2$
END ;

Un modèle est donc la définition d'une machine à transition d'état écrit dans un langage unique décrivant à la fois les transitions et l'invariant. Sur un tel modèle, l'outil, par exemple Click'n prove, génère alors trois obligations qui sont toutes vérifiées automatiquement. En effet, on doit vérifier que l'état initial implique l'invariant, que l'évènement *rely* conserve l'invariant, de même pour *assign*.

Dans un deuxième temps, on définit un raffinement, *ie* une autre machine qui va raffiner le premier modèle. Dans ce modèle-ci, on encode l'algorithme de la figure 4.1 de manière tout à fait classique. Le modèle complet se trouve en Annexe B.2. Le seul élément nouveau se trouve dans l'invariant. En effet, l'invariant va aussi contenir l'invariant de collage (cf section4.3). Dans cet exemple, on utilise principalement une relation entre les prédicats de place concrets et abstraits. De plus, on utilise le prédicat de place pour encoder l'invariant local $v = f(r)$ dont nous avions déjà eu besoin dans la section 4.1.2. Il devient $pp2(p) = P3 \Rightarrow v(p) = f(r(p))$, c'est à dire que quand un processus p est dans la place $P3$, alors $v(p) = f(r(p))$.

La méthode rend explicite les points de linéarisabilité. En effet, les évènements sont nommés et on ne peut en définir de nouveau. On connaît donc clairement où ils sont.

L'outil génère 43 obligations de preuves dont 23 sont à faire manuellement. Nombre d'entre elles sont simples, mais deux sont un peu plus délicates. En effet, il faut donner deux théorèmes au prouveur (disponible en commentaire dans le code de l'Annexe B.2). Pourtant il les prouve automatiquement. La difficulté est double ici. Tout d'abord l'écriture et surtout la lecture des modèles sont difficiles. Les invariants locaux sont tous regroupés dans l'invariant global et chaque variable locale devient une fonction. Cette contrainte n'en est pas une théorique mais seulement pratique. Par contre, elle a un impact directe sur le nombre de preuves automatiques du prouveur de l'outil.

4.5 $^+$CAL et TLA$^+$

TLA$^+$ est un langage de spécification inventé par Lamport. Il est basé sur la Logique Temporelle des Actions (*Temporal Logic af Actions*) (TLA) [43]. TLA est un formalisme logique qui permet à la fois de décrire un système de transitions mais aussi des formules logiques temporelles. TLA$^+$ a été spécialement inventé afin d'étudier formellement les systèmes réactifs et concurrents. Lamport désirait un langage :

- contenant un nombre minimal de concepts pouvant décrire et analyser des systèmes ;
- représentant la composition de système, ainsi que l'implémentation ;
- basé au maximum sur des concepts mathématiques simples.

Ce formalisme unique permet ainsi de comparer à la fois un système à ses propriétés mais aussi les systèmes entre eux. Le raffinement n'est donc en TLA$^+$ qu'une simple implication logique. TLA$^+$ n'est actuellement équipé que d'un *model checker* appelé TLC [54]. TLC ne peut vérifier que des systèmes à espaces finis ce qui en limite grandement l'usage dans notre cas. Néanmoins il peut être d'un grand recours dans la phase initiale de création d'un algorithme. Des travaux existent afin de pouvoir vérifier des formules TLA dans l'assistant à la preuve Isabelle [61].

Une spécification TLA$^+$ a pour forme générale $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$. La formule *Init* représente un ensemble d'états initiaux. Le tuple *vars* contient toutes les variables du système. La formule à deux états *Next* représente toutes les transitions possibles du systèmes. Enfin L est une formule temporelle indiquant l'équité des actions (que nous ne décrirons pas ici). Dans une version simplifiée de l'exemple, on pourrait définir :

$Next \triangleq Assign \vee Rely$
$Assign \triangleq x' = f(x)$
$Rely \triangleq x' \in Nat$

x' représente la valeur de x après la transition. La vérification du bon "typage" de x reviendrait donc à vérifier le théorème suivant :

THEOREM
$Spec \implies \Box(x \in Nat)$

Cet exemple illustre bien le fait qu'un programme est représenté par une formule temporelle comme une autre. Logiquement, c'est comme si la spécification devait être un modèle pour la propriété à vérifier. La vérification de la consistance séquentielle s'écrit donc facilement puisqu'il suffit de montrer que la

spécification concrète implique la spécification abstraite. La linéarisabilité peut aussi se faire relativement facilement. Le raffinement est donc très naturel à écrire en TLA$^+$ car il n'y a pas de points de linéarisation à donner, contrairement à B.

Les modèles TLA$^+$ s'éloignant de l'écriture de pseudo-code, $^+$CAL a été inventé [44]. $^+$CAL est donc un langage dédié à la vérification d'algorithmes, notamment distribués. Un algorithme $^+$CAL est ensuite traduit en TLA$^+$ par l'outil du même nom.

```
–algorithm Assign variables x ;

process ProcAssign ∈ 1 .. M begin
   OP : x := f(x)
 end process

process ProcRely ∈ M .. N begin
   OPR : with val ∈ Nat do
        x := val
      end with ;
 end process

end algorithm
```

FIG. 4.7 – Algorithme $x = f(x)$ en $^+$CAL.

```
–algorithm casAssign variables x ;

macro BCAS1(x, o, n, r) begin
 if (x = o) then
   x := n ;
   r := TRUE ;
 else r := FALSE ;
 end if
end macro

procedure CasAssign() variables b, v, fv ; begin
A :    b := FALSE ; A1 :    while ¬b do
B :      v := x ;
C :      fv := f(v) ;
D :      BCAS1(x, v, fv, b) ; end while
end procedure

process ProcAssign ∈ 1 .. M begin
   OP : call CasAssign() ;
 end process

process ProcRely ∈ M .. N begin
   OPR : with val ∈ Nat do
        x := val
      end with ;
 end process

end algorithm
```

FIG. 4.8 – L'implémentation utilisant CAS_1 en $^+$CAL.

La figure 4.7 montre la description de N processus pouvant soit modifier x de manière quelconque, soit affecter $f(x)$ à x. Cet algorithme va nous servir de spécification pour la vérification de l'implémentation utilisant CAS_1 de la figure 4.8. Les labels OP, A, B, C, D servent à indiquer les points d'entrelacements autorisées. Autrement dit, ils ont le même rôle que les évènements en B. La traduction complète des deux algorithmes se trouve en Annexe C. La traduction de $^+$CAL en TLA$^+$ suit les même schémas que ceux présentés pour la méthode B. Ainsi l'on va représenter explicitement les processus. La ligne $^+$CAL contenant le mot-clef *process* définit ainsi l'ensemble des processus en TLA$^+$: $ProcSet \triangleq (1 \,..\, N)$. Cet

ensemble va servir de domaine à toutes les fonctions qui vont représenter les variables locales. Ainsi dans le raffinement les variables locales b, v, fv vont être représentées par des variables de type fonction de $ProcSet \to Nat$. Chaque label va être traduit par un opérateur TLA$^+$. Et comme il correspond à l'évènement d'un processus, le processus qui effectue le pas est donné en paramètre. Bien entendu, comme en B, le compteur ordinal pc est géré explicitement, tout comme la pile des appels de procédures. De plus, il faut aussi obligatoirement indiquer quelles sont les variables qui ne sont pas modifiées. Ainsi,
A : $b :=$ FALSE ;
devient :

$$\begin{aligned} A(self) \triangleq\ & \wedge pc[self] = \text{“A”} \\ & \wedge b' = [b \text{ EXCEPT } ![self] = \text{FALSE}] \\ & \wedge pc' = [pc \text{ EXCEPT } ![self] = \text{“A1”}] \\ & \wedge \text{UNCHANGED } \langle x,\ stack,\ v,\ fv \rangle \end{aligned}$$

La procédure *CasAssign* peut alors être représentée comme la disjonction des opérateurs A, B, C, D. Un pas du système est alors $Next \triangleq \exists\, self \in ProcSet : CasAssign(self) \vee \ldots$. Les deux spécifications étant définies, on peut alors exprimer que le second système raffine le premier. Pour cela, on étend une des spécifications et l'on importe la première. Le théorème affirmant le raffinement n'est alors qu'une implication. La vérification se ferait donc sur le module suivant :

MODULE *CasAssignRefine*

The *CasAssign* model is a refinement of the *Assign* one.

EXTENDS *CasAssign*

$Atomic \triangleq$ INSTANCE *Assign*
$AtomicSpec \triangleq Atomic!Spec$

THEOREM
$Spec \implies AtomicSpec$

L'ensemble $^+$CAL-TLA$^+$ serait donc un bon candidat pour la vérification d'algorithmes sans verrou. Mais le manque d'outils restreint son utilisation à la conception. En effet, la vérification du raffinement précédent n'est pas possible avec TLC. L'espace d'état n'est pas fini, pire, il y a une paramétrisation sur la fonction f. Par ailleurs, même si des prouveurs existaient, l'automatisation des preuves TLA$^+$ semble peut probable sur des exemples réalistes. Prouver que $Init_1 \wedge \Box[Next_1]_{vars_1} \implies Init_2 \wedge \Box[Next_2]_{vars_2}$, revient déjà à prouver $Init_1 \implies Init_2$, mais aussi $[Next_1]_{vars_1} \implies [Next_2]_{vars_2}$. Autant la première formule serait facile à prouver, autant la deuxième est difficile. Cette deuxième formule nécessiterait l'utilisation d'un invariant tout comme en B. Mais contrairement à B, les évènements ne sont pas discrétisés. La formule *Next* est vue comme un tout et les outils n'ont pas accès à chacun des évènements. Bien sûr, par discipline, on peut se forcer à l'écrire de la sorte.

Chapitre 5

Modélisation d'algorithmes sans verrou

Nous allons maintenant introduire notre méthode pour faire la preuve d'algorithmes sans verrou. Comme on a pu le voir dans le chapitre précédent et ainsi que l'énumère Jones dans [40], il faut une méthode qui soit compositionnelle, qui permette de vérifier la linéarisabilité, qui permette de raisonner séquentiellement et qui laisse libre du niveau de granularité choisi. Notre méthode va donc tâcher d'extraire l'essence des méthodes du chapitre précédent et de les combiner afin de faciliter son utilisation par un développeur. Il faut donc que les preuves soient au maximum automatiques.

Plus particulièrement, notre méthode sera basée sur le raffinement. C'est un moyen très simple, de non seulement faciliter la compréhension d'un système de par l'introduction pas à pas de détails, mais d'assurer aussi la linéarisabilité à condition que les opérations abstraites soient atomiques. Par ailleurs, la vérification doit se faire pour un nombre quelconque de processus.

Dans ce chapitre, nous allons en même temps récapituler les besoins et les formaliser. Puis nous exposerons les obligations de preuves que doivent vérifier de tels systèmes afin d'être valides garantissant les propriétés désirées. Enfin nous exposerons les preuves de correction de cette formalisation dans le chapitre suivant. Le chapitre 7 contiendra le traitement des exemples.

5.1 Introduction

Comme défini à la section 3.6, nous considérerons ici le modèle où chaque processus possède une mémoire locale et peut accéder à une mémoire partagée. Par ailleurs, chaque processus évolue à son rythme ; les vitesses d'exécutions sont indépendantes. Néanmoins nous allons considérer comme immédiatement visible les lectures et écritures en mémoire partagée. C'est-à-dire que, du point de vue système, les exécutions de chacun des processus sont entrelacées (*interleaving semantics*). Par contre, nous ne supposerons rien sur l'ordonnancement des processus. Un processus peut donc tout à fait exécuter plusieurs actions à la suite.

Les actions qu'effectue un processus sont définies par un algorithme. Un algorithme est la définition impérative d'une opération. Le degré d'atomicité d'une opération n'est pas fixe mais dépend du niveau d'abstraction choisi. Ainsi le modélisateur peut décider de laisser transparaître des étapes intermédiaires ou non. Par la suite, chaque opération sera modélisée par un algorithme. L'appel d'une opération est simulé par le fait qu'un processus commence à exécuter l'algorithme correspondant. La sémantique d'une opération est propre à chaque type de données (Voir section 3.2). Mais ce qui importe est qu'elles doivent toutes travailler de concert. D'où la nécessité de les développer en même temps. Par ailleurs, il nous faut maintenant décrire ce que fait une action. Nous adoptons le point de vue suivant : une action est une relation entre l'état du système avant et son état après avoir effectué l'action. L'état d'après est représenté par des variables primées, et l'état avant par des variables simples. Ainsi l'affectation $x := y+1$ est représentée par $x' = y+1 \wedge y' = y$ dans un environnement où x et y sont les seules variables. On peut

aussi exprimer des transitions plus complexes qui ne seraient pas forcément exécutables atomiquement par un processeur. Il en va ainsi de la double affectation $\langle x, y\rangle := \langle y+2, x+3\rangle$ qui s'écrit $x' = y+2 \wedge y' = x+3$.

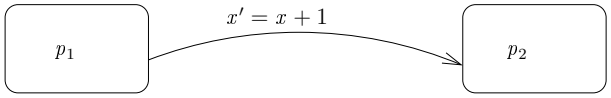


FIG. 5.1 – Modélisation d'un algorithme *Inc* simple incrémentant x

Pour représenter un algorithme, nous allons nous baser sur son flot de contrôle. Chaque noeud du graphe du flot de contrôle est appelé place. Tout comme un processeur, un processus possède un compteur ordinal *pc* indiquant l'unique place à laquelle il est. Ainsi l'action de l'algorithme constitué d'une unique affectation $x := x + 1$ en passant de la place p_1 à p_2 est $pc = p_1 \wedge x' = x + 1 \wedge pc' = p_2$. Dans la suite nous réduirons souvent l'action à son effet sur les variables autre que le compteur ordinal. Les actions sur le compteur ordinal étant exprimées directement par un graphe orienté comme sur la figure 5.1.

Par ailleurs, il existe des variables locales à un processus, d'autres sont globales. Cette distinction reflète l'architecture étudiée. L'action précédente exécutée par le processus $Proc_i$ sera ainsi notée $x' = x+1$ en supposant x global (sinon ce serait $x'_i = x_i + 1$). Une variable particulière est *pid*. Le *pid* correspond au numéro i de processus. C'est une constante unique à chaque processus. Un des cas d'utilisation du *pid* est l'identification du processus étant à l'origine d'une action. L'exemple de la section 7.3 l'utilise à cette fin. Le *pid* peut aussi servir à différencier un processus maître des autres.

5.2 Formalisation

Dans la section précédente, nous avons eu un court aperçu des notions utiles à la description des systèmes étudiés. Nous allons maintenant formaliser ces notions afin de pouvoir faire la preuve des algorithmes.

Comme illustré par la figure 5.2, un système consiste en un nombre indéterminé n de processus $Proc_1, \ldots, Proc_n$. Chacun de ces processus exécute un algorithme particulier parmi $Algo_1, \ldots, Algo_m$. On retrouve la même organisation en ce qui concerne les états. Le système est composé d'une mémoire partagée, aussi appelé environnement global *genv*. Notons $genv : \mathcal{G}$ le fait que *genv* soit de type $\mathcal{G}$. La nature précise des types sera expliquée au paragraphe suivant. Chaque processus i exécutant un algorithme $\alpha(i)$ a un état caractérisé par sa mémoire locale $lenv_i : \mathcal{L}_{\alpha(i)}(i)$ ainsi que son compteur ordinal $pc : \mathcal{P}$ (*program counter* en anglais). La mémoire globale *genv* est accessible par tous les processus *a contrario* des mémoires locales.

Notre formalisation n'impose pas de structures particulières sur la définition des mémoires globales et locales. Une définition classique de la RAM comme étant une fonction des naturels vers les entiers $\mathcal{G} \triangleq \mathbb{N} \to \mathbb{Z}$ conviendrait tout à fait. Néanmoins, par la suite, nous les supposerons de type $\mathcal{G} \triangleq GV \to \mathcal{V}$ avec GV un ensemble de variables globales et un ensemble $\mathcal{V}$ de valeurs. Techniquement nous désirons que les variables locales à un processus aient un nom unique. L'environnement local dépend donc à la fois de l'algorithme $\alpha(j)$ exécuté par le processus j et du processus. L'environnement local de l'algorithme a sera donc de type $\mathcal{L}_a \triangleq LV_a \to \mathbb{N} \to \mathcal{V}$. Nous noterons $PV_{\alpha(j)}(j) = GV \cup LV_{\alpha(j)}(j)$ l'ensemble des variables auxquelles un processus j peut accéder.

Enfin nous définissons $\mathcal{SF}(V)$ l'ensemble des formules caractérisant un état (*State Formula*). Cet ensemble est constitué des formules de la logique du premier ordre étendue par la théorie de l'arithmétique sur les entiers, dont les variables libres sont dans V. Par exemple, $\mathrm{SF}(\{a\}) \triangleq \{a, \neg a\}$ en supposant que a soit de type booléen. De manière similaire, $\mathcal{TF}(V)$ dénote l'ensemble des formules de transitions. Elles peuvent contenir les variables libres v et v' pour toutes variables $v \in V$. En effet, les formules de transitions sont interprétées sur une paire d'états et nous utilisons donc ici la notation classique où les variables non primées réfèrent au premier état, et les primées au deuxième. Ainsi les formules $\neg a \wedge a' = \neg a$ et $even(b) \wedge b' = b + 1$ appartiennent à $\mathcal{TF}(\{a, b\})$. Pour une formule d'état P, P' dénote la formule de transition obtenue depuis P en remplaçant toutes les variables libres par leur version primée.

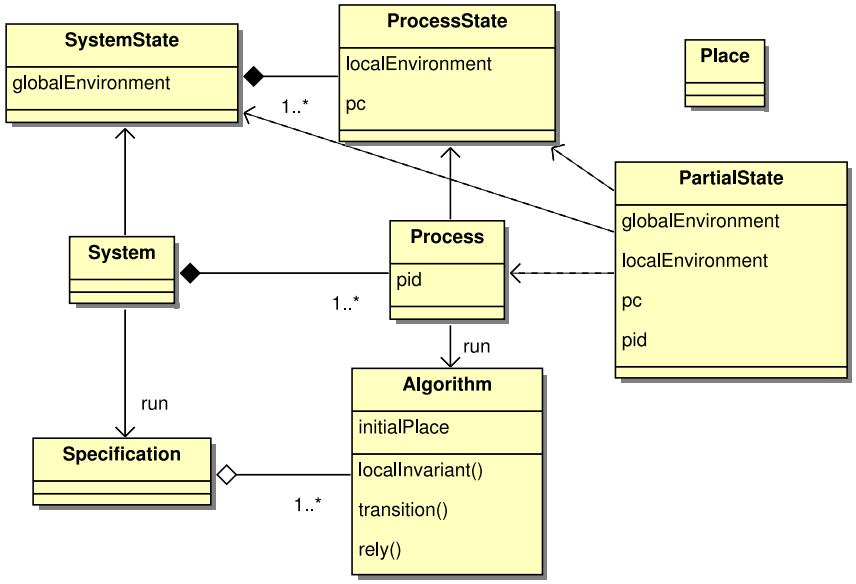


FIG. 5.2 – Modélisation d'un système sans verrou.

Définition 14 *Un algorithme* $A = (Q, q_0, \delta, I, R)$ *agissant sur un ensemble de variables* $PV_A(i) = GV \cup LV_A(i)$ *est défini par :*

- *un ensemble fini de* places $Q : 2^{\mathcal{P}}$*, avec une place initiale* $q_0 \in Q$*,*
- *une relation de transition* $\delta : Q \to Q \to \mathbb{N} \to \mathcal{TF}(PV)$ *associant une formule de transition* $\delta(q, q', i) \in \mathcal{TF}[PV(i)]$ *pour toutes paires de places* q *et* q'*, et avec* i *un numéro de processus,*
- *une fonction d'état définissant un* invariant local $I : Q \to \mathbb{N} \to \mathcal{SF}(PV)$ *tel que pour toute place* q *et tous processus* i*,* $I(q, i) \in \mathcal{SF}(PV(i))$
- *pour chaque place* q*, une fonction de transition* rely $R : Q \to \mathbb{N} \to \mathcal{TF}(PV)$ *telle que* $R(q, i) \in \mathcal{TF}(PV(i))$

L'algorithme A *est bien formé si*

$$I(q,i) \wedge \delta(q,q',i) \implies I(q',i)' \qquad et \qquad I(q,i) \wedge R(q,i) \wedge LV_A(i)' = LV_A(i) \implies I(q,i)'$$

sont valides pour toutes les places $q, q' \in Q$ *et tous les processus* i*.*

La relation δ exprime donc le flot de contrôle de l'algorithme mais aussi les changements d'états autorisés. Un algorithme est ainsi simplement représenté comme un système de transitions annoté. L'annotation de l'invariant local $I(q, i)$ permet de caractériser la vue partielle qu'a un processus i en place q.

Dans l'algorithme *Inc* de la figure 5.1, on peut supposer que x est une variable de type entier positif ou nul. Si x peut être nulle en p_1, ce n'est plus le cas en p_2. Ces propriétés sont encodées dans les invariants. Le *rely* doivent donc être adapté afin de respecter les invariants. Nous pourrions donc encoder cet algorithme *Inc* comme suit :

$$\begin{array}{ll}
GV_{Inc} \triangleq & \{x\} \\
LV_{Inc} \triangleq & \{i \mapsto \varnothing\} \\
Inc \triangleq & (Q_{Inc}, p_1, \delta_{Inc}, I_{Inc}, R_{Inc}) \\
Q_{Inc} \triangleq & \{p_1, p_2\} \\
\delta_{Inc}(p_1, p_2, i) \triangleq & x' = x + 1 \\
\delta_{Inc}(p_1, p_1, i) = \delta_{Inc}(p_2, p_2, i) \triangleq \bot & \\
I_{Inc}(p_1, i) \triangleq & x \in \mathbb{N} \\
I_{Inc}(p2, i) \triangleq & x \in \mathbb{N}^+ \\
R_{Inc}(p_1, i) \triangleq & x' \in \mathbb{N} \\
R_{Inc}(p_2, i) \triangleq & x' \in \mathbb{N}^+
\end{array}$$

Les obligations de preuves vérifiant que l'algorithme est bien formé sont données ci-après. La première permet d'assurer l'invariant sous l'unique transition. On suppose l'invariant vrai avant en p_1 et on doit assurer l'invariant après en p_2. Les deux autres permettent d'assurer l'invariant lorsque d'autres processus font un pas, à condition qu'ils respectent le *rely*. On remarque aussi que la logique des formules n'est pas imposée par notre modélisation. Elle est choisie en fonction des invariants, des transitions voire de l'encodage de la mémoire. Ceci fait que ces preuves sont en pratique faciles à prouver comme on le verra dans le chapitre 7.

$$\begin{array}{ll} x \in \mathbb{N} \wedge x' = x + 1 \implies x' \in \mathbb{N}^+ & I_{Inc}(p_1, i) \wedge \delta_{Inc}(p_1, p_2, i) \implies I_{Inc}(p_2, i)' \\ x \in \mathbb{N} \wedge x' \in \mathbb{N} \implies x' \in \mathbb{N} & I_{Inc}(p_1, i) \wedge R_{Inc}(p_1, i) \implies I_{Inc}(p_1, i)' \\ x \in \mathbb{N}^+ \wedge x' \in \mathbb{N}^+ \implies x' \in \mathbb{N}^+ & I_{Inc}(p_2, i) \wedge R_{Inc}(p_2, i) \implies I_{Inc}(p_2, i)' \end{array}$$

La formule de transition R, dite de *rely*, est inspirée de la méthode *assume-guarantee* (cf chapitre 4.1). Dans la méthode *assume-guarantee*, elle est globale et sert à exprimer avec quel environnement on peut composer la spécification. Ici, elle est locale à chaque place. Cela permet de garder l'aspect local du raisonnement. Ainsi au final, pour pouvoir composer avec une autre spécification, il faut que chaque transition de celle-ci vérifie tous les *rely* de celle-ci. Par exemple, on peut composer la spécification précédente avec l'algorithme effectuant l'unique transition :

$$x \in \mathbb{N}^+ \wedge x' = 2 * x$$

En effet, cette transition vérifie bien tous les *rely* (celui de p_1 et de p_2) :

$$\begin{array}{lll} x \in \mathbb{N}^+ \wedge x' = 2 * x & \implies & x' \in \mathbb{N} \\ x \in \mathbb{N}^+ \wedge x' = 2 * x & \implies & x' \in \mathbb{N}^+ \end{array}$$

Par ailleurs, on va aussi se servir de cette relation pour la composition "interne", c'est-à-dire composer deux processus qui exécutent un même algorithme ou une même spécification. Ainsi chacune des transitions de chacun des algorithmes doit respecter le rely R de chacune des places de tous les algorithmes. Rappelons que, comme on a pu le voir dans la section 3.2, il est souvent nécessaire de co-développer plusieurs algorithmes. Dès lors on définit une spécification comme étant un ensemble d'algorithmes.

Définition 15 *Soit GV un ensemble de variables globales et $\forall i \in \mathbb{N}. LV_1(i), \ldots, LV_m(i)$ des ensembles de variables locales disjointes de GV, et deux à deux disjointes. On définit $PV_a(i) = GV \cup LV_a(i)$. Une spécification est un ensemble d'algorithmes $Spec = \{Alg_1, \ldots, Alg_m\}$ ou chaque $Alg_a = (Q_a, q_{0,a}, \delta_a, I_a, R_a)$ est un algorithme sur l'ensemble des variables PV_a. La spécification Spec est bien formée si*

$$I_a(q_i, i) \wedge I_b(q_j, j) \wedge \delta_b(q_j, q'_j, j) \wedge LV_a(i)' = LV_a(i) \implies R_a(q_i, i)$$

pour tout $a, b \in \{1, \ldots, m\}$, $q_i \in Q_a$, $q_j, q'_j \in Q_b$, et i et j des identifiants distincts de processus.

Dans l'exemple de l'incrémentation, il n'y a qu'un algorithme et qu'une seule transition. On doit donc vérifier que cette transition vérifie bien les deux formules de *rely*. L'obligation de preuve assurant que la transition $\delta(p_1, p_2, i)$ vérifie le *rely* de p_1 est :

$$\begin{array}{l} \wedge\ x \in \mathbb{N} \\ \wedge\ x \in \mathbb{N} \\ \wedge\ x' = x + 1 \\ \implies \\ x' \in \mathbb{N} \end{array}$$

Si x était local, il faudrait vérifier que deux processus i et j distinct $i \neq j$ utilisant cet algorithme peuvent se composer, d'où :

$$\begin{array}{l} \wedge\ x_i \in \mathbb{N} \\ \wedge\ x_j \in \mathbb{N} \\ \wedge\ x'_j = x_j + 1 \\ \wedge\ x'_i = x_i \\ \implies \\ x'_i \in \mathbb{N} \end{array}$$

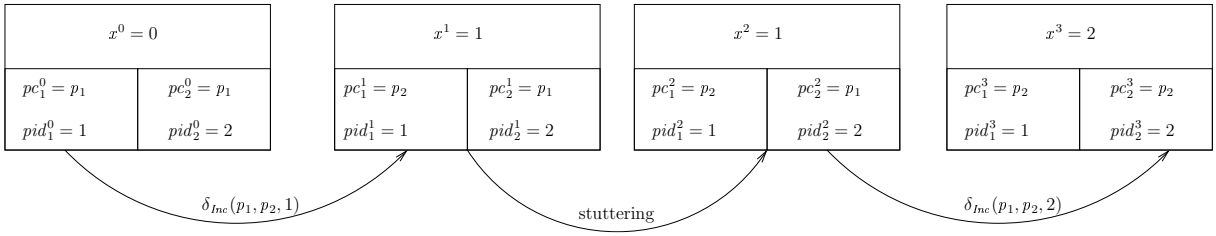


FIG. 5.3 – exécution d'un système composé de 2 processus exécutant l'algorithme d'incrémentation.

Nous n'avons défini jusqu'à maintenant que la partie statique d'un système sans verrou. En effet, les algorithmes ne définissent que le code que chaque processus exécute. A l'exécution, un système est composé de plusieurs processus chacun exécutant un algorithme (qui peut être le même). Ainsi la figure 5.3 représente une exécution d'un système à deux processus chacun effectuant le même algorithme d'incrémentation. Un état du système est donc décomposé en trois parties. Une première partie correspond à l'état global. Les deux autres parties correspondent aux états locaux des deux processus. Les variables locales apparaissent ainsi clairement, de même que les variables globales. Cette figure fait apparaître artificiellement les pseudo-variables locales pid_i et pc_i afin d'illustrer la nécessité de faire des copies des variables locales par processus. La modélisation présentée jusqu'ici ne rend pas explicite ces pseudo-variables mais elles le sont dans la modélisation en Isabelle dont nous parlerons plus loin. La pseudo-variable *pid* représente un numéro de processus, alors que le *pc* représente le compteur ordinal (*program counter*) et donc la place active. Les deux transitions de l'exécution illustrée sont formalisées par l'unique transition de l'algorithme *Inc*. Le *stuttering*, ou bégaiement, est autorisé dans une exécution comme illustré par la transition de l'état 1 à 2 dans la figure 5.3. Le *stuttering* nous sera utile lors du raffinement.

Définition 16 *Un système Sys exécutant une spécification Spec consiste en n processus* $Proc_1, \ldots, Proc_n$ *où chaque processus* $Proc_i$ *exécute l'algorithme* $\alpha(i) \in Spec$. *L'état du système est donné par le tuple* $s = (genv, (pc_1, lenv_1), \ldots, (pc_n, lenv_n))$ *où genv évalue les variables globales de GV,* $pc_i \in Q_{\alpha(i)}$ *est le compteur ordinal du processus,* $Proc_i$, *et* $lenv_i$ *évalue une copie des variables locales* LV_i *du processus* $Proc_i$. *Enfin* pid_i *est considéré comme une constante valant i. Une exécution du système est donc une* ω*-séquence* $\sigma = s^0 s^1 \ldots$ *d'états du système telle que*

- *les places de tous les processus en* s^0 *sont les places initiales de l'algorithme correspondant, et les environnements partiels* $penv_i$ *sont des états initiaux satisfaisant l'invariant local correspondant à la place initiale.*

$$pc_i^0 = q_{\alpha(i),0} \wedge penv_i^0 \models I_{\alpha(i)}(pc_i^0, i)$$

  pour tout $i \in 1 \ldots n$, *et où* $penv_i^k$ *est l'état partiel donné par* $(genv^k, lenv_i^k)$.
- *chaque transition* (s^k, s^{k+1}) *correspond soit à la transition* $\delta_{\alpha(i)}$ *d'un processus* $Proc_i$, *laissant inchangé les autres états locaux,*

$$\begin{array}{ll} \wedge & (penv_i^k, penv_i^{k+1}) \models \delta_{\alpha(i)}(pc_i^k, pc_i^{k+1}, i) \\ \wedge & \forall j \in 1 \ldots n.\ j \neq i \implies lenv_j^{k+1} = lenv_j^k \wedge pc_j^{k+1} = pc_j^k \end{array}$$

  pour un unique $i \in 1..n$, *soit à un* stuttering, ie $s^{k+1} = s^k$.

Nous supposons une mémoire séquentiellement consistante. C'est pourquoi nous supposons un entrelacement des transitions. Par ailleurs nous acceptons le *stuttering* afin de simplifier le raffinement défini dans la section suivante. Par contre, nous n'autorisons pas la création et la destruction de processus dynamiquement. Si la création est une part important de la preuve alors il est toujours possible de faire appel aux variables fantômes, comme définies à la Sect. 4.1.1. Ce n'est donc pas une véritable limitation. Enfin il faut remarquer que les systèmes exécutant une spécification bien formée respectent leur invariant tout au long de leur exécution :

Théorème 17 *Soit Sys un système correspondant à une spécification bien formée Spec, et* $\sigma = s^0 s^1 \ldots$ *une exécution de Sys. Soit* $k \in \mathbb{N}$ *et* $s^k = (genv^k, (pc_1{}^k, lenv_1{}^k), \ldots, (pc_n{}^k, lenv_n{}^k))$. *Alors pour tout* $i \in \{1, \ldots, n\}$:

$$penv_i{}^k \models I_{\alpha(i)}(pc_i{}^k)$$

où $penv_i{}^k$ *est l'état partiel donné par* $(genv^k, lenv_i{}^k)$ *et* $I_{\alpha(i)}$ *est l'invariant de l'algorithme exécuté par le processus* $Proc_i$.

Preuve. La preuve se fait par induction sur k. Pour $k = 0$, par la définition 16 d'un système, on obtient bien :

$$penv_i{}^0 \models I_{\alpha(i)}(pc_i{}^0)$$

Supposons $k > 0$ et la propriété valide en k. Si $s^{k+1} = s^k$ et donc (s^k, s^{k+1}) est un *stuttering*, la propriété est alors vraie en $k+1$. Si (s^k, s^{k+1}) est une transition du processus j, et on peut alors utiliser les définitions. 14 et 15. Dans la suite de la preuve, les variables non primées correspondant aux variables de l'état k, celles primées à l'état $k+1$.

Puisque c'est j qui effectue la transition et que la spécification est bien formée (définition 15), tous les autres processus $i \neq j$ ont leur rely vérifié :

$$I_{\alpha(i)}(pc_i, i) \wedge I_{\alpha(j)}(pc_j, j) \wedge \delta_{\alpha(j)}(pc_j, pc'_j, j) \wedge LV_{\alpha(i)}(i)' = LV_{\alpha(i)}(i) \implies R_{\alpha(i)}(pc_i, i)$$

Or nous savons aussi que l'invariant est stable sous l'action du rely (de par la définition 14), d'où :

$$I_{\alpha(i)}(pc_i, i) \wedge R_{\alpha(i)}(pc_i, i) \implies I_{\alpha(i)}(pc_i, i)'$$

De plus, l'algorithme $\alpha(j)$ étant bien formé (cf définition 14), on sait qu'une transition effectuée dans un état vérifiant l'invariant nous fait arrivé dans un autre état vérifiant lui aussi l'invariant :

$$I_{\alpha(j)}(pc_j, j) \wedge \delta_{\alpha(j)}(pc_j, pc'_j, j) \implies I_{\alpha(j)}(pc'_j, j)'$$

Seul le compteur ordinal du processus j ayant changé, l'invariant est bien vérifié pour tous les processus $i \neq j$ et pour j. □

L'encodage de cette formalisation en Isabelle/HOL se trouve en Annexe D. Celui-ci présente des différences subtiles par rapport à la formalisation. Tout d'abord, comme expliqué dans l'introduction de cette section, le type des places, de la mémoire locale et globale sont des paramètres de la formalisation. L'encodage est donc plus proche du système de la figure 5.2. Nous avons aussi défini explicitement ce qu'est un état partiel, c'est-à-dire l'état partiel du système vu par un processus :

```
record ('place,'genv,'lenv)PartialState =
  genv :: "'genv"
  lenv :: "'lenv"
  pc :: "'place"
  pid :: "nat"
  algo_index :: "nat"
```

La référence à un algorithme se fait par son index dans la liste des algorithmes d'une spécification comme le montre le type du champ `algo_index` correspondant à la fonction α utilisée jusqu'à présent. La relation de transition d'un algorithme est différente et est définie comme une fonction de type

`('place,'genv,'lenv)PartialState ⇒ ('place,'genv,'lenv)PartialState ⇒ bool`

Cet encodage démontre bien que la modélisation est indépendante de la logique utilisée pour décrire les invariants, transitions, etc. Il a par contre l'inconvénient d'introduire certains artefacts. Les pseudo-variables *pc* et *pid* apparaissent explicitement. De plus, il duplique certaines informations comme, par exemple, le *pid*et l'environnement global. Nous avons donc dû introduire la fonction `coherent_state`. Cette fonction assure que les environnements globaux de deux états partiels sont identiques puisque ce sont deux vues partielles d'un même état du système. Il assure aussi que si les deux processus ont le même *pid* alors ils ne sont qu'un et donc les environnements locaux aussi (idem pour le *pc*). Les obligations de preuves sont renforcés afin de ne pas autoriser les transitions à modifier le *pid*. Malgré tout, cet encodage

permet de garder des signatures de fonctions simples et compréhensibles. Les autres encodages testés noient le lecteur sous des détails techniques.

Maintenant que nous pouvons décrire une spécification, ainsi que l'état d'un système (et donc d'un processus), le chapitre suivant décrit comment établir des liens de raffinement entre spécifications.

Chapitre 6

Preuve de correction

6.1 Raffinement

Les définitions de la section précédente nous permettent de spécifier un algorithme ainsi que l'exécution d'un système. Dans cette section, nous nous attellerons à prouver la correction de ceux-ci. À cette fin nous utiliserons le raffinement (cf section 4.3). En effet, la propriété qui nous intéresse étant la linéarisabilité, il est intéressant de partir d'une spécification atomique (et donc séquentielle) des opérations des structures de données puis de la raffiner. En montrant le raffinement, on assure la linéarisabilité de fait. Le raffinement se fait modulo une abstraction des données, ce qui nous permettra d'ajouter des détails au fur et à mesure du développement.

Par la suite $\overline{x}$ (resp. $\underline{x}$) dénotera la variable x de la spécification abstraite (resp. concrète). Il en sera de même pour les environnements et autres états. Afin de motiver la définition d'un raffinement valide, nous allons nous baser sur un exemple simple. Nous commencerons ainsi par modéliser cet exemple selon les principes du précédent chapitre. Nous montrerons alors quelles sont les propriétés nécessaires pour démontrer le raffinement. Cet exemple nous permettra ainsi d'introduire la définition formelle et surtout de l'illustrer simplement. Des exemples plus conséquents seront présentés dans le chapitre 7 suivant.

```
/* IncAB (abstrait) */
global x;
local a,b;
x += a + b;
```

```
/* IncAB (concret) */
global x;
local a,b,v;
v = a + b;
x += v;
```

L'utilisation de variables temporaires est très commune en programmation. Ainsi il est intuitif que, d'une certaine façon, le deuxième programme ($\underline{IncAB}$) raffine le premier ($\overline{IncAB}$)[1] C'est-à-dire que d'un point de vue extérieur, on observe exactement la même chose. Ils sont indiscernables l'un de l'autre quelque soit l'environnement avec lequel ils interagissent.

En effet, de l'extérieur, on ne peut modifier a et b puisque ce sont des variables locales. On ne peut donc qu'observer le même fonctionnement. Si a et b n'étaient pas locales, il suffirait de modifier une des deux variables entre les exécutions des deux commandes de l'algorithme de droite pour observer un comportement différent.

De plus, le point de linéarisabilité est bien sûr l'affectation à x. C'est en effet la seule commande modifiant l'unique variable globale x. Autrement dit, pour toute exécution concrète, on doit trouver une trace abstraite telle que l'effet de l'affectation ait lieu entre le début et la fin de l'exécution de la deuxième affectation concrète, comme illustré sur la figure 6.1. Sur la figure, les instructions ont une durée. En effet, on peut considérer le fait de rentrer dans une place précédent une instruction comme le début de l'appel d'une opération, et le fait de sortir d'une place suivant l'instruction comme la fin de l'appel. Les transitions internes sont bien quant à elles atomiques. La linéarisabilité est bien assurée

[1] Les deux programmes étant équivalents, la réciproque est aussi valide.

puisque la transition (atomique) abstraite est effectuée au même instant que la transition concrète. Elle même a eu lieu entre le début de l'appel et le retour.

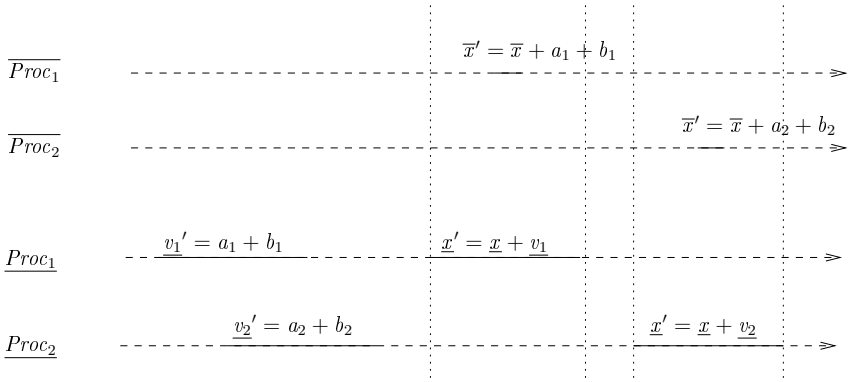


FIG. 6.1 – Exécutions linéarisables utilisant des variables temporaires.

L'algorithme abstrait $\overline{IncAB}$ correspond à l'exécution atomique, c'est-à-dire celle n'utilisant pas de variables temporaires. On l'encode donc comme indiqué sur la figure 6.2 et conformément au chapitre précédent. On a ici procédé au renommage des variables au processus i. Ainsi a_i est la variable locale au processus i. L'algorithme concret $\underline{IncAB}$ serait lui encodé comme indiqué sur la figure 6.3.

$$
\begin{array}{ll}
\overline{GV} \triangleq & \{\overline{x}\} \\
\overline{LV_{IncAB},_i} \triangleq & \{\overline{a_i}, \overline{b_i}\} \\
\overline{IncAB} \triangleq & (Q_{\overline{IncAB}}, \overline{p_1}, \delta_{\overline{IncAB}}, I_{\overline{IncAB}}, R_{\overline{IncAB}}) \\
Q_{\overline{IncAB}} \triangleq & \{\overline{p_1}, \overline{p_2}\} \\
\delta_{\overline{IncAB}}(\overline{p_1}, \overline{p_2}, i) \triangleq & \overline{x}' = \overline{x} + \overline{a_i} + \overline{b_i} \wedge \overline{a_i}' = \overline{a_i} \wedge \overline{b_i}' = \overline{b_i} \\
I_{\overline{IncAB}}(\overline{p_1}, i) \triangleq & \overline{x} \in \mathbb{N} \wedge \overline{a} \in \mathbb{N} \wedge \overline{b} \in \mathbb{N} \\
I_{\overline{IncAB}}(\overline{p_2}, i) \triangleq & \overline{x} \in \mathbb{N} \wedge \overline{a} \in \mathbb{N} \wedge \overline{b} \in \mathbb{N} \\
R_{\overline{IncAB}}(\overline{p_1}, i) \triangleq & \overline{x}' \in \mathbb{N} \\
R_{\overline{IncAB}}(\overline{p_2}, i) \triangleq & \overline{x}' \in \mathbb{N}
\end{array}
$$

FIG. 6.2 – Définition complète de l'algorithme $\overline{IncAB}$.

Comme on cherche à montrer l'interchangeabilité des deux programmes $\overline{IncAB}$ et $\underline{IncAB}$, on ne raffine pas les données. Les invariants de collage assurent donc tous $\overline{x} = \underline{x} \wedge \overline{a_i} = \underline{a_i} \wedge \overline{b_i} = \underline{b_i}$. L'encodage du raffinement et des algorithmes est résumé par la figure 6.5. Les flèches pointillées indiquent une partie de l'invariant de collage mais aussi les points de linéarisabilité. La définition de l'invariant de collage est donnée en figure 6.4.

La modélisation de ce raffinement doit nous permettre de reconstituer une trace abstraite pour toute trace concrète. Nous allons utiliser à cette fin la simulation en avant vue à la section 4.3. Par exemple, étant donnée la trace concrète au bas de la figure 6.6, on doit pouvoir recréer la trace abstraite (en haut) de la même figure. Sur cette figure, on représente un système concret à deux processus. Celui-ci effectue deux pas. On montre aussi la relation avec un système abstrait lui-aussi à deux processus et la trace abstraite de ce système qui colle bien avec la trace concrète. Si on arrive à reconstruire une trace abstraite pour *toutes* traces concrètes alors le raffinement est assuré.

Supposons donc une trace concrète quelconque, nous allons maintenant définir les propriétés nécessaires à la reconstruction d'une trace abstraite. Pour cela, il nous faut un système concret $\underline{Sys}$, une spécification concrète $\underline{Spec} = \{\underline{Alg}_1, \ldots, \underline{Alg}_m\}$, une spécification abstraite $\overline{Spec} = \{\overline{Alg}_1, \ldots, \overline{Alg}_m\}$, tel que $\underline{Spec}$ soit un raffinement de $\overline{Spec}$, et une exécution valide du système concret $\underline{\sigma} = \underline{s}^0 \underline{s}^1 \ldots$. Notez

$$
\begin{array}{ll}
\underline{GV} \triangleq & \{\underline{x}\} \\
LV_{\underline{IncAB}}(i) \triangleq & \{\underline{a_i}, \underline{b_i}, \underline{v_i}\} \\
\underline{IncAB} \triangleq & (Q_{\underline{IncAB}}, \underline{p_1}, \delta_{\underline{IncAB}}, I_{\underline{IncAB}}, R_{\underline{IncAB}}) \\
Q_{\underline{IncAB}} \triangleq & \{\underline{p_1}, \underline{p_2}, \underline{p_3}\} \\
\delta_{\underline{IncAB}}(\underline{p_1}, \underline{p_2}, i) \triangleq & \underline{v_i}' = \underline{a_i} + \underline{b_i} \wedge \underline{a_i}' = \underline{a_i} \wedge \underline{b_i}' = \underline{b_i} \wedge \underline{x}' = \underline{x} \\
\delta_{\underline{IncAB}}(\underline{p_2}, \underline{p_3}, i) \triangleq & \underline{x}' = \underline{v_i} \wedge \underline{a_i}' = \underline{a_i} \wedge \underline{b_i}' = \underline{b_i} \wedge \underline{v_i}' = \underline{v_i} \\
I_{\underline{IncAB}}(\underline{p_1}, i) \triangleq & \underline{x} \in \mathbb{N} \wedge \underline{a_i} \in \mathbb{N} \wedge \underline{b_i} \in \mathbb{N} \\
I_{\underline{IncAB}}(\underline{p_2}, i) \triangleq & \underline{x} \in \mathbb{N} \wedge \underline{a_i} \in \mathbb{N} \wedge \underline{b_i} \in \mathbb{N} \wedge \underline{v_i} = \underline{a_i} + \underline{b_i} \\
I_{\underline{IncAB}}(\underline{p_3}, i) \triangleq & \underline{x} \in \mathbb{N} \wedge \underline{a} \in \mathbb{N} \wedge \underline{b} \in \mathbb{N} \\
R_{\underline{IncAB}}(\underline{p_1}, i) \triangleq & \underline{x}' \in \mathbb{N} \\
R_{\underline{IncAB}}(\underline{p_2}, i) \triangleq & \underline{x}' \in \mathbb{N} \\
R_{\underline{IncAB}}(\underline{p_3}, i) \triangleq & \underline{x}' \in \mathbb{N}
\end{array}
$$

FIG. 6.3 – Définition complète de l'algorithme $\underline{IncAB}$.

$$
\begin{array}{ll}
G_{IncAB}(\underline{p_1}, \overline{p_1}, i) \triangleq & \overline{x} = \underline{x} \wedge \overline{a_i} = \underline{a_i} \wedge \overline{b_i} = \underline{b_i} \\
G_{IncAB}(\underline{p_2}, \overline{p_1}, i) \triangleq & \overline{x} = \underline{x} \wedge \overline{a_i} = \underline{a_i} \wedge \overline{b_i} = \underline{b_i} \\
G_{IncAB}(\underline{p_3}, \overline{p_2}, i) \triangleq & \overline{x} = \underline{x} \wedge \overline{a_i} = \underline{a_i} \wedge \overline{b_i} = \underline{b_i} \\
\text{Sinon } G_{IncAB}(p, q) \triangleq & \bot
\end{array}
$$

FIG. 6.4 – Définition de l'invariant de collage G_{IncAB}.

que les deux spécifications doivent avoir le même nombre d'algorithmes puisqu'elles doivent être toutes les deux une implémentation d'une même structure de données. Afin de reconstituer une trace abstraite $\overline{\sigma} = \overline{s}^0\overline{s}^1\ldots$, il faut d'abord trouver un état initial abstrait $\overline{s}^0$. De plus, les états initiaux doivent bien évidemment vérifier l'invariant de collage. Dans notre exemple simple, il suffit de prendre les mêmes valeurs que les variables concrètes. Ici l'invariant de la place $\overline{p_1}$ est le même que celui de $\underline{p_1}$, il est donc évidemment respecté.

Maintenant, nous nous retrouvons exactement dans le cas général de la figure 4.4 et instancié sur un cas à trois processus par la figure 6.7 pour la simulation en avant. Nous avons en effet le processus 1 qui effectue un pas entre les états $(\underline{s}^0, \underline{s}^1)$ et de plus nous avons aussi l'état initial abstrait $\overline{s}^0$. Les conditions de validité doivent donc être suffisantes pour reconstruire $\overline{s}^1$ tel que le pas entre $(\overline{s}^0, \overline{s}^1)$ soit un *stuttering* ou bien le pas d'un unique processus. De plus il faut aussi que $(\underline{s}^1, \overline{s}^1)$ vérifie les invariants de collage de chacun des processus.

Dans notre exemple *IncAB* et pour le pas concret $(\underline{p_1}, \underline{p_2})$, il faut que les processus qui sont en $(\underline{p_1}, \overline{p_1})$, $(\underline{p_2}, \overline{p_1})$ ou en $(\underline{p_3}, \overline{p_3})$ voient leur invariant de collage toujours vérifié. De plus, comme $\underline{p_2}$ ne raffine que $\overline{p_1}$, le pas abstrait ne peut être qu'un *stuttering*. Spécifiquement au pas $(\underline{p_1}, \underline{p_2})$ de l'exemple, et au minimum, il faut donc que la formule suivante soit valide :

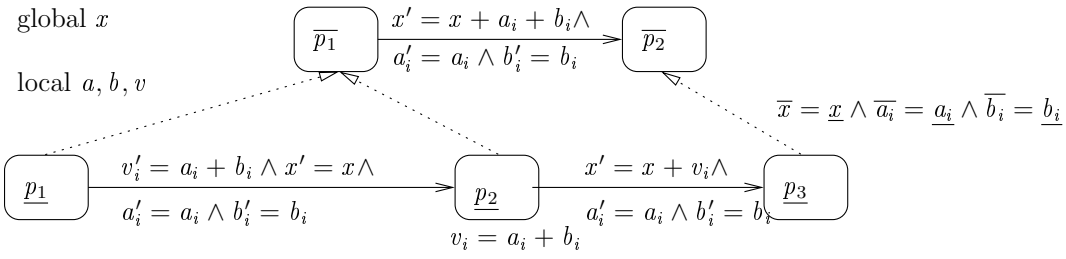


FIG. 6.5 – Illustration de l'usage des variables temporaires par le raffinement.

$\forall \underline{x}\ \underline{x}'\ \overline{x}.$		Quelques soient les états des
$\forall \underline{a_i}\ \underline{b_i}\ \underline{v_i}\ \underline{a_i}'\ \underline{b_i}'\ \underline{v_i}'\ \overline{a_i}\ \overline{b_i}.$		processus abstrait et concret i,
	$\wedge\ I_{\underline{IncAB}}(\underline{p_1}, i)$	tel que le concret effectue le pas $(\underline{p_1},\underline{p_2})$
	$\wedge\ \delta_{\underline{IncAB}}(\underline{p_1}, \underline{p_2}, i)$	de l'algorithme 1,
	$\wedge\ I_{\overline{IncAB}}(\overline{p_1}, i)$	tel que l'invariant et
	$\wedge\ G_{IncAB}(\underline{p_1}, \overline{p_1}, i)$	l'inv. de collage 1 avec le processus abstrait i,
$\Longrightarrow\ \exists \overline{x}'\ \overline{a_i}'\ \overline{b_i}'.($		Il existe un état abstrait suivant
	$\wedge\ \overline{x}' = \overline{x}$	tel que le pas abstrait soit, dans l'exemple,
	$\wedge\ \overline{a_i}' = \overline{a_i} \wedge \overline{b_i}' = \overline{b_i} \wedge ($	un *stuttering* du processus i
$\bigwedge_{(\underline{m},\overline{m})\ \in\ \{(\underline{IncAB},\overline{IncAB})\}}$		quelque soit les algorithmes que...
$\forall j \neq i.$		des autres processus j abstrait et concret
$\forall LV_{\underline{m}}(j) LV_{\overline{m}}(j).$		et leurs états
$\forall \underline{q_j}\ \overline{q_j}.$		et quelques soient leurs places,
	$\wedge\ I_{\underline{m}}(\underline{q_j}, j) \wedge I_{\overline{m}}(\overline{q_j}, j)$	tels que leurs invariants soient vérifiés,
	$\wedge\ G_m(\underline{q_j}, \overline{q_j}, j)$	tels qu'ils "collent" initialement
	$\wedge\ LV_{\underline{m}}(j)' = LV_{\underline{m}}(j) \wedge LV'_{\overline{m}}(j) = LV_{\overline{m}}(j)$	mais tels qu'ils ne font pas de pas
$\Longrightarrow$		alors
	$G_m(\underline{q_j}, \overline{q_j}, j)'))$	ils doivent toujours coller après

En remplaçant les définitions et en simplifiant un peu, on obtient pour le pas $(\underline{p_1}, \underline{p_2})$ de l'exemple :

$\forall x\ a_i\ b_i\ \underline{v_i}.$		
	$x \in \mathbb{N} \wedge a_i \in \mathbb{N} \wedge b_i \in \mathbb{N} \wedge$	L'invariant $I_{\underline{IncAB},i}(\underline{p_1})$, $I_{\overline{\underline{IncAB}},i}(\overline{p_1})$ sont satisfaits ;
	$v' = a_i + b_i \wedge x' = x \wedge \ldots \wedge$	Le pas concret $\delta_{IncAB,i}(\underline{p_1}, \underline{p_2})$ est effectué ;
	$\ldots$	L'invariant de collage G_i est vérifié ;
$\Longrightarrow\ (\exists x'\ a'_i\ b'_i\ \underline{v_i}'.$		On cherche un état abstrait suivant tel que
	$x' = x \wedge a'_i = a_i \wedge b'_i = b_i \wedge$	Il y ait un *stuttering* abstrait
$(\forall a_j\ b_j\ \underline{v_j}\ \underline{q_j}\ \overline{q_j}.$		Quelque soit les autres processus tel que
	$x \in \mathbb{N} \wedge a_j \in \mathbb{N} \wedge b_j \in \mathbb{N} \wedge$	Leurs invariants $I_{\underline{IncAB},j}(\underline{q_j})$, $I_{\overline{\underline{IncAB}},j}(\overline{q_j})$ sont satisfaits ;
	$G_j(\underline{q_j}, \overline{q_j})$	Ils collent ;
	$a'_j = a_j \wedge b'_j = b_j \wedge \underline{v_j}' = \underline{v_j}$	Ils font un *stuttering* ;
$\Longrightarrow$	$G_j(\underline{q_j}, \overline{q_j})'))$	ils doivent toujours coller après.

Remarquons que l'ensemble Q des places étant toujours fini et fixe, on peut donc remplacer les quantificateurs sur les places $\underline{q_j}, \overline{q_j}$ par une conjonction sur les couples $(\underline{p_1}, \overline{p_1}), (\underline{p_2}, \overline{p_1}), (\underline{p_3}, \overline{p_2})$. Pour bien comprendre cette obligation de preuve, voyons ce qu'il se passe pour la transition $(\underline{p_2}, \underline{p_3})$. Le pas abstrait est forcément $(\overline{p_1}, \overline{p_2})$ puisque $\underline{p_3}$ ne raffine que $\overline{p_2}$.

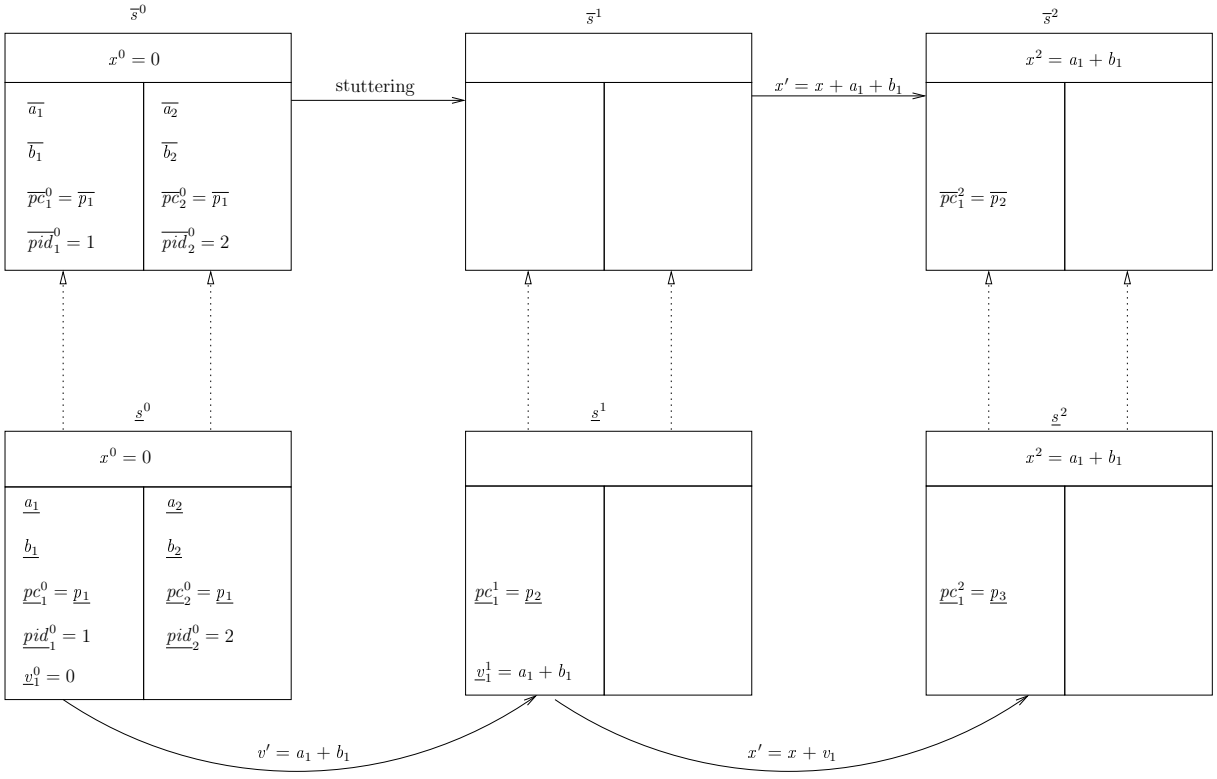


FIG. 6.6 – Exécution concrète raffinant une autre trace abstraite. Seuls les éléments essentiels et/ou changeants sont indiqués.

$\forall \underline{x}\ \underline{x}'\ \overline{x}.$		Quelques soient les états des
$\forall \underline{a_i}\ \underline{b_i}\ \underline{v_i}\ \underline{a_i}'\ \underline{b_i}'\ \underline{v_i}'\ \overline{a_i}\ \overline{b_i}.$		processus abstrait et concret i,
	$\wedge\ I_{\underline{IncAB}}(\underline{p_2}, i)$	tel que le concret effectue le pas $(\underline{p_2},\underline{p_3})$,
	$\wedge\ \delta_{\underline{IncAB}}(\underline{p_2}, \underline{p_3}, i)$	de l'algorithme 1
	$\wedge\ I_{\overline{IncAB}}(\overline{p_2}, i)$	tel qu'il y ait l'invariant et
	$\wedge\ G_{IncAB}(\underline{p_2}, \overline{p_1}, i)$	inv. de collage 1 avec le processus abstrait i,
$\implies \exists \overline{x}'\ \overline{a_i}'\ \overline{b_i}'.($		il existe un état abstrait suivant
	$\delta_{\overline{IncAB}}(\overline{p_1}, \overline{p_2}, i) \wedge ($	tel que le pas abstrait soit $(\overline{p_1}, \overline{p_2})$ par i
$\bigwedge_{(\underline{m},\overline{m})\ \in\ \{(\underline{IncAB},\overline{IncAB})\}}$		quelque soit les algorithmes que...
$\forall j \neq i.$		des autres processus j abstrait et concret
$\forall LV_{\underline{m}}(j) LV_{\overline{m}}(j).$		et leurs états
$\forall \underline{q_j}\ \overline{q_j}.$		et quelques soient leurs places,
	$\wedge\ I_{\underline{m}}(\underline{q_j}, j) \wedge I_{\overline{m}}(\overline{q_j}, j)$	tels que leurs invariants soient vérifiés,
	$\wedge\ G_m(\underline{q_j}, \overline{q_j}, j)$	tels qu'ils "collent" initialement
	$\wedge\ LV_{\underline{m}}(j)' = LV_{\underline{m}}(j)$	mais tels qu'ils ne font pas de pas
	$\wedge\ LV_{\overline{m}}(j)' = LV_{\overline{m}}(j)$	
$\implies$		alors
	$G_m(\underline{q_j}, \overline{q_j}, j)'))$	ils doivent toujours coller après

Dans l'exemple, on connaît évidemment quel est le pas abstrait et de plus il est unique. Mais de manière générale, ce n'est pas le cas. De plus, ce n'est pas forcément l'abstraction du processus faisant le pas concret qui fera le pas abstrait. On doit donc généraliser cette idée. En particulier, les invariants de collage de l'état suivant s'écrivent différemment pour les processus i et j selon que $i = j$ ou non. Si

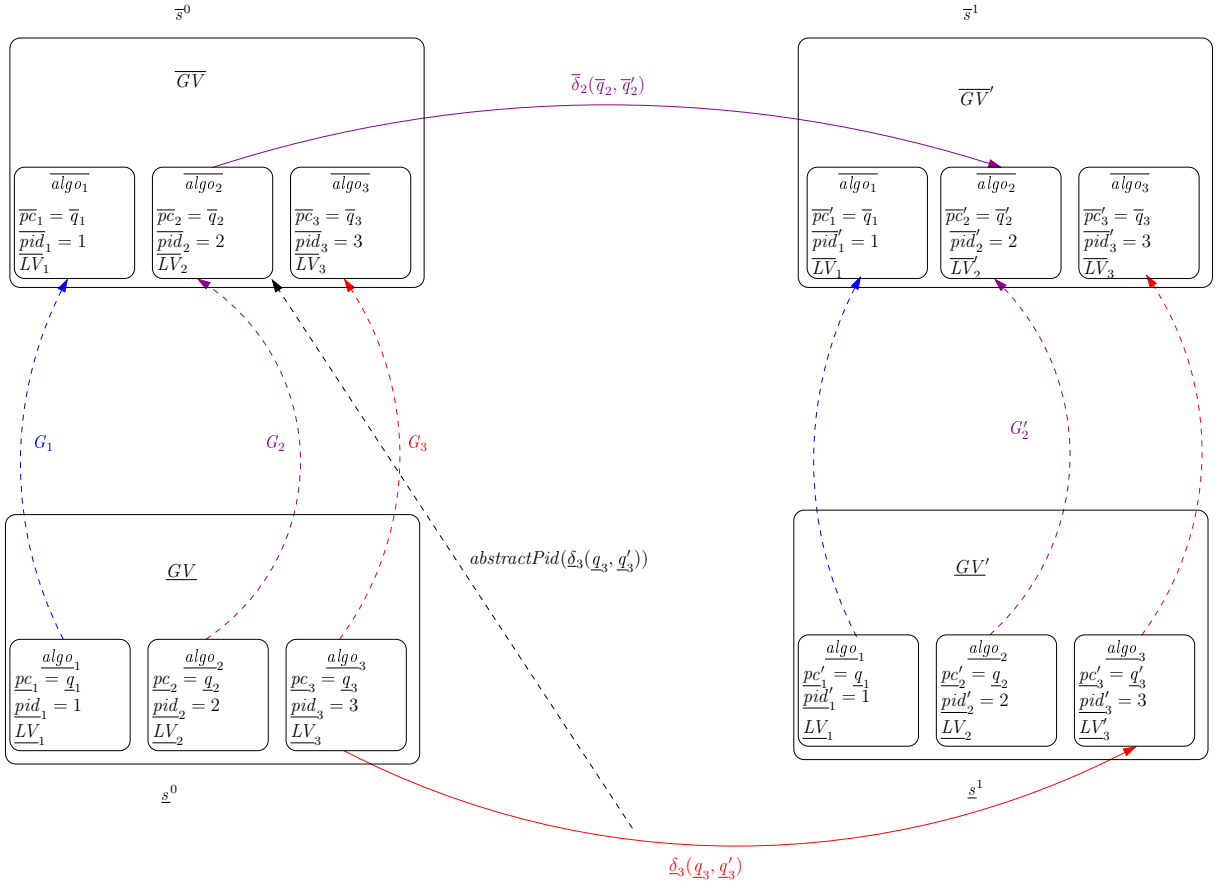


FIG. 6.7 – Une instanciation de l'obligation de preuve principale du raffinement. Il s'agit de trouver $\overline{q}'_2$, $\overline{GV}'$ et $\overline{LV}'_2$.

$i = j$ alors il n'y a qu'un seul invariant de collage à vérifier et celui-ci se fait sur les états abstraits et concrets suivant. Sinon il faut en vérifier deux. Le premier concerne l'état concret après de i, et l'état abstrait partiel constitué du nouvel environnement global (changé par la transition effectué par j) et l'état local d'avant de i. Le deuxième concerne l'état concret partiel constitué du nouvel environnement global (changé par la transition concrète de i) et l'environnement local d'avant de j, et l'état partiel abstrait de j. Bien sûr, pour les processus autres que i et j, il faut toujours que leurs invariants de collage soient vérifiés.

Ce sont ces obligations qui devront être créées par le générateur d'obligations de preuve. Ces preuves peuvent être relativement facilement automatisées. En effet, bien qu'en théorie nous sommes dans un sous-ensemble non décidable, en pratique —comme le montrent les exemples du chapitre suivant— les transitions et le *stuttering* dirigent les prouveurs dans la recherche d'une instanciation valide. Remarquons aussi que ces obligations de preuves évitent les encodages difficiles qui sont obtenus dans les autres méthodes du chapitre 4.

D'un point de vue théorique, n'importe quel processus abstrait pourrait effectuer le pas. Néanmoins dans la pratique, on connaît le processus abstrait effectuant le pas. L'exemple des descripteurs dans la section 7.3 illustre ce point. On renforce donc la troisième condition en explicitant le *pid* du processus abstrait en fonction de la transition concrète effectuée. La fonction *abstractPid* remplit ce rôle. Une illustration de cette condition est donnée en figure 6.7. Il faut aussi rajouter une quatrième condition qui assure que le résultat est un *pid* valide, autrement dit qu'on ne peut forger de *pid*. La suite de la formalisation est donc paramétrée par un prédicat *validPid* indiquant si un *pid* est valide. Ainsi dans la notation basé sur les entiers utilisée jusqu'ici, il dépend de la taille du système, *ie* du nombre de processus.

Dans un système à n processus, on le définit par $\lambda pid.0 < pid \leq n$. La définition complète est donc :

Définition 18 *Soit deux spécifications* $\underline{Spec} = \{\underline{Alg}_1, \ldots, \underline{Alg}_m\}$ *et* $\overline{Spec} = \{\overline{Alg}_1, \ldots, \overline{Alg}_m\}$, *soit* $G = (G_1, \ldots, G_m)$ *des invariants de collages, alors* $\underline{Spec}$ *raffine* $\overline{Spec}$ *modulo* G *si les conditions suivantes sont vérifiées :*

1. $\overline{Spec}$ *et* $\underline{Spec}$ *sont bien formées.*
2. *Pour toute place concrète initiale il existe une place abstraite vérifiant l'invariant de collage :*

$$\left(\bigwedge_{a=1}^{m} \underline{I}_a(\underline{q}_{0,a}, a)\right) \implies \exists \overline{GV}\ LV_{\overline{1}}(a) \ldots LV_{\overline{m}}(a).\ \bigwedge_{a=1}^{m} G_a(\underline{q}_{0,a}, \overline{q}_{0,a}, a) \wedge \overline{I}_a(\overline{q}_{0,a}, a)$$

3. *Toute action concrète correspond à une action abstraite d'une opération, ou bien alors à un* stuttering *:*

$$\begin{array}{ll}
\forall i\ j\ a\ b\ LV_{\underline{a}}(i) LV_{\underline{b}}(j) LV_{\overline{a}}(i) LV_{\overline{b}}(j). & \wedge\ a = \alpha(i) \\
& \wedge\ b = \alpha(j) \\
& \wedge\ validPid(i) \\
& \wedge\ j = abstractPid(\underline{q_i}, \underline{q_i}', i) \\
& \wedge\ \delta_{\underline{a}}(\underline{q_i}, \underline{q_i}', i) \\
& \wedge\ I_{\underline{a}}(\underline{q_i}, i) \\
& \wedge\ G_a(\underline{q_i}, \overline{q_i}, i) \\
& \wedge\ G_b(\underline{q_j}, \overline{q_j}, j) \\
& \wedge\ (i \neq j \implies LV_{\underline{b}}(j)' = LV_{\underline{b}}(j)) \\
\implies & \exists \overline{GV}'\ LV_{\overline{a}}(i)'\ LV_{\overline{b}}(j').\ \bigvee_{\overline{q_j}' \in Q_{\overline{b}}} .(\\
& \wedge\ \vee\ \delta_{\overline{b}}(\overline{q_j}, \overline{q_j}', j) \\
& \quad \vee\ \wedge\ \overline{q_j}' = \overline{q_j} \\
& \qquad \wedge\ \overline{GV}' = \overline{GV} \\
& \qquad \wedge\ LV_{\overline{b}}(j)' = LV_{\overline{b}}(j) \\
& \wedge\ \text{IF}\ j = i \\
& \quad \text{THEN}\ G_a(\underline{q_i}, \overline{q_j}', i)' \\
& \quad \text{ELSE}\ \wedge\ G_a(\underline{q_i}', \overline{q_i}, i)' \\
& \qquad \wedge\ G_b(\underline{q_j}, \overline{q_j}', j)' \\
& \qquad \wedge\ LV_{\overline{a}}(i)' = LV_{\overline{a}}(i) \\
& \wedge\ \bigwedge_{c=1}^{m}\ k \notin \{i, j\} LV_{\underline{c}}(k)\ LV_{\overline{c}}(k).\ \bigwedge_{\underline{q_k} \in Q_{\underline{c}}} .\ \bigwedge_{\overline{q_k} \in Q_{\overline{c}}} . \\
& \quad \wedge\ c = \alpha(k) \\
& \quad \wedge\ LV_{\underline{c}}(k)' = LV_{\underline{c}}(k) \\
& \quad \wedge\ LV_{\overline{c}}(k)' = LV_{\overline{c}}(k) \\
& \quad \wedge\ G_c(\underline{q_k}, \overline{q_k}, k) \\
& \quad \wedge\ I_{\underline{c}}(\underline{q_k}, k) \\
& \quad \wedge\ I_{\overline{c}}(\overline{q_k}, k) \\
& \implies\ \wedge\ G_c(\underline{q_k}, \overline{q_k}, k)' \\
& \qquad \wedge\ I_{\overline{c}}(\overline{q_k}, k)')
\end{array}$$

   doit être valide pour toutes les places $\underline{q_i}$, $\underline{q_i}'$ *de l'algorithme* $\underline{a}$, *les places* $\overline{q_i}$ *de* $\overline{a}$, *et toutes les places* $\underline{q_j}$ *et* $\overline{q_j}$ *de* $\underline{b}$ *et* $\overline{b}$.
4. *Toute action concrète effectuée dans un environnement où le pid est valide donne un pid abstrait valide :*

$$\begin{array}{ll}
\forall i\ \bigwedge_{a=1}^{m}\ \underline{q_a}\ \underline{q_a}'. & \wedge\ validPid(i) \\
& \wedge\ \underline{\delta}_a(\underline{q}_a, \underline{q}_a', i) \\
& \wedge\ \underline{I}(\underline{q}_a) \\
\implies & validPid(abstractPid_a(\underline{q}_a, \underline{q}_a', i))
\end{array}$$

Ce sont ces conditions qui ont été formalisées en Isabelle/HOL par la fonction `valid_refinement` (cf Annexe D). Reste donc à démontrer que cette définition permet bien de démontrer la linéarisabilité et qu'elle est pratique. Le premier point sera l'objet de la section suivante, le second du chapitre 7.

6.2 Correction

Tout d'abord il est évident qu'une spécification devrait se raffiner elle-même. Le théorème suivant vérifie cet état de fait. Pour cela, on définit l'invariant de collage comme étant l'identité et
$abstractPid\colon Q \to Q \to \mathrm{SF}[PV]$ la fonction qui renvoie une expression qui s'évalue comme le *pid* du processus effectuant la transition concrète.

Théorème 19 *Soit* $Spec \triangleq \{Alg_1, \ldots, Alg_m\}$ *une spécification bien formée,*
Soit $idTransPid_i \triangleq \lambda q, q'.i$,
Soit $g_i \triangleq \lambda \underline{q}_i\, \overline{q}_i.\overline{PV}_i = \underline{PV}_i$
Soit $G \triangleq (g_1, \ldots g_m)$,
alors Spec raffine Spec sous l'invariant de collage G et le mapping *de pid idTransPid.*

Preuve. La première condition est évidente puisque *Spec* est bien formée. La deuxième condition est-elle aussi validée aisément par l'identité. La quatrième condition se récrit en $\bigwedge_{i=1}^{m} validPid(\underline{pid}_i) \implies validPid(\underline{pid}_i)$ qui est évidemment valide.

La troisième condition consiste donc à vérifier que pour chaque transition concrète, il existe une transition abstraite ou un *stuttering*. Il est là aussi évident que ce sera la même transition mais abstraite qui convient. Autrement dit, il suffit d'instancier les valeurs suivantes : $\overline{GV}' = \underline{GV}'$, $\overline{LV}'_i = \underline{LV}'_i$ et $\overline{q}'_i = \underline{q}'_i$. Il est alors aisé de démontrer que la disjonction sera valide pour $j = i$. □

Notre intuition est donc bien vérifiée dans cette formalisation. Reste donc à voir maintenant le cas général. Nous avions en fait en tête la simulation en avant comme dit plus haut. C'est-à-dire qu'il faut démontrer que pour chaque exécution d'un système $\underline{Sys}$ correspondant à une spécification concrète $\underline{Spec}$, il existe une exécution d'un système $\overline{Sys}$ correspondant à la spécification abstraite $\overline{Spec}$, tel que $\underline{Spec}$ raffine $\overline{Spec}$. Pour cela nous avons besoin d'un lemme correspondant exactement à l'illustration de la simulation en avant (voir aussi figure 6.7).

Lemme 20 *Soit*
- $\overline{Spec} \triangleq \{\overline{Algo}_1 \ldots \overline{Algo}_m\}$ *et* $\underline{Spec} \triangleq \{\underline{Algo}_1 \ldots \underline{Algo}_m\}$ *deux spécifications telles que* $\underline{Spec}$ *raffine* $\overline{Spec}$ *sous G et abstractPid, et avec le paramètre validPid étant* $\lambda pid.pid \leq n$
- *un système* $\overline{Sys} \triangleq \{\overline{Proc}_1 \ldots \overline{Proc}_n\}$ *correspondant à la spécification* $\overline{Spec}$ *et tel que* $\overline{pid}_i = i$,
- *un système* $\underline{Sys} \triangleq \{\underline{Proc}_1 \ldots \underline{Proc}_n\}$ *correspondant à la spécification* $\underline{Spec}$ *et tel que* $\underline{pid}_i = i$,
- $\overline{s}$ *une exécution (partielle) du système* $\overline{Sys}$, $\overline{s}^k$ *est donc un état de* $\overline{Sys}$,
- $\underline{s}$ *une exécution du système* $\underline{Sys}$ *telle que les deux états* $(\underline{s}^k, \underline{s}^{k+1})$ *correspondent à la transition* $\underline{\delta}_{\alpha(i)}$ *du processus* $\underline{Proc}_i$,
- $j \triangleq abstractPid(\underline{q}_i^k, \underline{q}_i^{k+1})$,

Si de plus $(\underline{s}^k, \overline{s}^k)$ *vérifie l'invariant de collage G,*
alors il existe un état $\overline{s}^{k+1}$
tel que $(\overline{s}^k, \overline{s}^{k+1})$ *correspond à la transition* $\overline{\delta}_{\alpha(j)}$ *du processus* $\overline{Proc}_j$
et tel que $(\underline{s}^{k+1}, \overline{s}^{k+1})$ *vérifie l'invariant de collage G.*

Preuve. De par la quatrième condition du raffinement, et comme $\underline{pid}_i = i$, on peut en déduire que $j \leq n$. On en déduit donc que le pas abstrait est fait par le processus $\overline{Proc}_j$.

Les prémisses de la troisième condition du raffinement sont toutes vérifiées par hypothèses. En effet, les *pid* sont bien valides, il y a bien une transition concrète $\underline{\delta}_{\alpha(i)}$, les invariants de collages sont assurés et les invariants abstraits et concrets aussi de par le théorème 17.

La conséquence de cette troisième condition nous assure donc l'existence d'un état suivant $\overline{s}^{k+1}$. De plus, La première partie de la conséquence implique que $(\overline{s}^k, \overline{s}^{k+1})$ est soit un *stuttering*, soit le pas du processus j.

La deuxième partie de la conséquence garantie l'invariant de collage des processus i et j selon que $i = j$ ou non. La figure 6.7 illustre le cas où $i = 3$ et $j = 2$. Enfin la troisième partie de la conséquence vérifie l'invariant de collage pour tous les autres processus que i et j (cas du processus 1 sur la figure). Ansi les invariants de collage sont bien vérifiés par les états $(\underline{s}^{k+1}, \overline{s}^{k+1})$. □

Théorème 21 *Soit*
- $\overline{Spec} \triangleq \{\overline{Algo}_1 \dots \overline{Algo}_m\}$ *et* $\underline{Spec} \triangleq \{\underline{Algo}_1 \dots \underline{Algo}_m\}$ *deux spécifications telles que* $\underline{Spec}$ *raffine* $\overline{Spec}$ *sous* G *et* $abstractPid$,
- *un système* $\overline{Sys} \triangleq \{\overline{Proc}_1 \dots \overline{Proc}_n\}$ *correspondant à la spécification* $\overline{Spec}$,
- *un système* $\underline{Sys} \triangleq \{\underline{Proc}_1 \dots \underline{Proc}_n\}$ *correspondant à la spécification* $\underline{Spec}$,
- $\underline{\sigma} \triangleq \underline{s}^0 \underline{s}^1 \dots$ *une exécution de* $\underline{Sys}$,

Alors il existe une exécution $\overline{\sigma} = \overline{s}^0 \overline{s}^1 \dots$ *de* $\overline{Sys}$ *tel que* $(\underline{s}^k, \overline{s}^k)$ *vérifie l'invariant de collage* G *pour tout* $k \in \mathbb{N}$.

Preuve. La preuve se fait par induction sur k. Pour $k = 0$, on obtient $\overline{s}^0$ par la première condition du raffinement. Pour k quelconque, soit $(\underline{s}^k, \underline{s}^{k+1})$ est une transition et on applique alors le lemme 20, soit c'est un *stuttering* et on utilise un lemme similaire au précédent. □

Ces théorèmes ont eux aussi été prouvés dans l'assistant à la preuve Isabelle/HOL. Avec les lemmes annexes, ils représentent environ 750 lignes de définitions et preuves en Isar [87]. Au total, nous avons donc :

300 lignes de lemmes utilitaires sur les listes,

700 lignes pour la formalisation et le théorème 19,

750 lignes pour le lemme 20 et le théorème 21.

Les conditions de validité d'une spécification sont données par la fonction Isabelle `valid_spec`, celui de l'état d'un système par `valid_state`. La quatrième condition du raffinement est décrit par `external_step` et le raffinement complet par `valid_refinement`, le tout est visible dans la section D.2. Le théorème 19 a été vérifié en Isabelle sous le nom `spec_refines_itself` (cf la section D.2.5 de l'annexe D). La vérification du lemme 20 est donnée en annexe dans la section D.3.3.

Ce dernier théorème permet d'assurer le raffinement. Les spécifications abstraites étant atomiques, le raffinement assure la linéarisabilité. Ce théorème démontre donc la correction de la méthode. Dans le chapitre suivant, nous présentons l'application de cette méthode à des exemples plus complets.

Chapitre 7

Application

7.1 Générateur d'obligation de preuves

La correction de la théorie, présentée dans les deux chapitres précédents, a été prouvée dans Isabelle/HOL [61, 66]. Pour pouvoir traiter les exemples présentés ci-après, un générateur d'obligation de preuve a été développé en Java™. Sloccount [12] mesure environ 12000 lignes de code Java™. Dans la réalité, une partie de celles-ci ont été générées par Tom [68]. Tom est un compilateur de *pattern matching* développé à l'INRIA. Il est utilisé pour manipuler et transformer la syntaxe abstraite des formules.

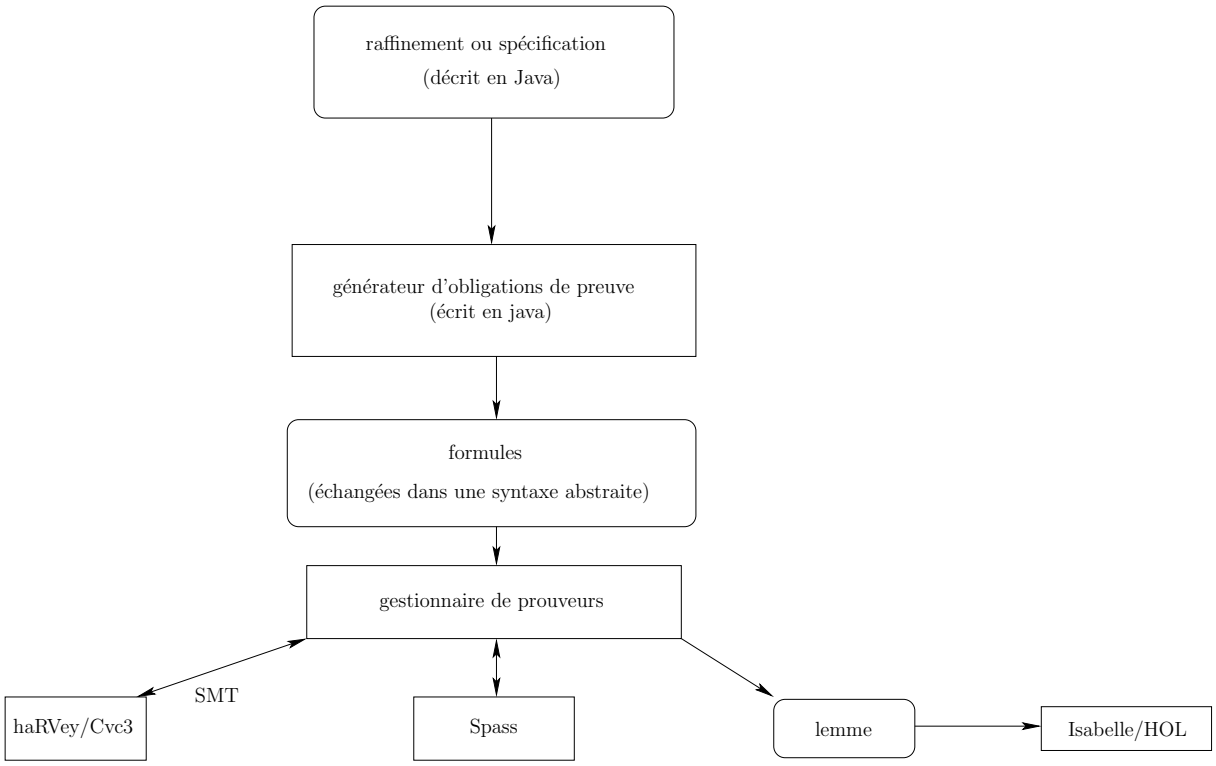


FIG. 7.1 – *workflow* du générateur d'obligations de preuve

Comme indiqué sur la figure 7.1, le générateur d'obligation de preuve prend en entrée une spécification et/ou un raffinement, puis génère les obligations de preuves dans une syntaxe abstraite. Ces formules sont alors envoyées à différents prouveurs. Le choix d'un prouveur dépend d'abord de la logique utilisée. Par exemple, la logique du premier ordre sans fonction interprétée et sans quantificateur est décidable et

	Preuve	Abandon	Timeout
CVC3 1.2.1	116	0	25
Z3 2007	123	18	0
haRVey modifié	130	4	7
Spass 3.0	95	—	2
E-prover	122	—	19

FIG. 7.2 – Performance des différents prouveurs sur les obligations de preuve de l'algorithme RDCSS (cf section 7.4). Les abandons de Spass et du E-prover ne sont pas donnés dû à des plantages sur certaines obligations de preuves.

les solveurs *Satisfiability Modulo Theories* (SMT) les gèrent particulièrement bien. En effet, les solveurs SMT utilisent un solveur booléen, *aka* solveur *Boolean satisfiability problem* (SAT) [60], pour la structure booléenne d'une obligation de preuve. Pour la partie du premier ordre et des théories particulières, notamment à des types abstraits (vecteur de bits, arithmétique linéaire, etc), ils utilisent une combinaison de procédures de décisions [58], pour la partie non-interprétée la clôture de congruence [59]. Il est connu que la combinaison est correcte et complète (*sound and complete*) sous certaines conditions [80]. De plus, ils sont capables de travailler sur des formules grandes (en nombre de symboles). Les solveurs basés sur la résolution savent, quant à eux, gérer la logique du premier ordre avec quantificateurs, mais en pratique on verra que, sur nos formules, ce ne sont pas toujours les plus performants.

Au minimum, notre méthode amène à utiliser une logique du premier ordre avec quantificateurs, avec des fonctions non-interprétées, et avec égalité. De plus, selon l'encodage de la mémoire ou de la structure de données, on pourrait vouloir utiliser une théorie particulière (tableau, liste, etc). Par ailleurs, les formules générées sont relativement grandes. Mais en fait, de par la composition des formules, les quantificateurs peuvent souvent être gérés par les solveurs SMT. Nous avons donc utilisé les deux types de prouveurs. Pour aider lors du cycle de modélisation, nous avons fait un export vers Isabelle. Les preuves sont effectivement réalisables en Isabelle bien que très lourdes à écrire.

Tout d'abord Spass [53, 86] et Cvc3 [77, 78, 5] ont été utilisés comme prouveurs automatiques. Spass est un prouveur automatique utilisant la résolution et la superposition. Cvc3 est un prouveur de problèmes SMT. Dans les exemples ci-après il n'y a pas de logique particulière utilisée (comme les tableaux, listes, etc). Il est fort probable par contre que cela soit nécessaire pour d'autres exemples.

Globalement, ces prouveurs sont performants. En effet, ils peuvent vérifier la validité des obligations de preuves des exemples qui suivent généralement en moins de 20 secondes. Mais à l'utilisation il y avait toujours quelques formules pour lesquelles ils ne terminaient pas. Nous sommes à la limite pratique de ces prouveurs à cause de l'utilisation des quantificateurs. Finalement, nous avons utilisé haRVey [65, 23, 18] après coopération avec les auteurs. En effet, ceux-ci ont pu intégrer un algorithme générique qui rend automatique plus de preuves que les autres prouveurs. Les exemples suivants ont ainsi pu être vérifiés automatiquement. La figure 7.2 récapitule la performance des différents prouveurs testés sur les obligations de preuves générées par la preuve de l'algorithme RDCSS. Cet exemple est traité en section 7.4. Seul la modélisation reste donc à la charge du programmeur !

Notons que ces performances sont aussi dues à quelques astuces de la génération. En effet, dans la théorie, il y a plusieurs conjonctions et disjonctions qui énumèrent les différentes places ou couples de places. Or, dans la modélisation, on connaît les différentes places et leurs relations. On peut donc "éclater" une obligation de preuves en obligations plus petites. Par exemple, dans le raffinement, on connaît explicitement que certains couples $(\overline{q}_j, \overline{q}'_j)$ de places ne satisfont pas la relation de transition. Apparaissant dans une disjonction, on peut donc s'abstenir de générer cette partie de la formule. Les prouveurs devraient pouvoir se passer de cette simplification, mais en pratique cela améliore souvent les temps de réponse. Comme nous allons le constater avec les exemples des sections suivantes, cet outil permet de simplifier grandement la phase de preuve d'algorithmes sans verrou, puisqu'il la rend automatique.

7.2 Affectation

```
void cas1Assign(word_t *x) {
  word_t r, c, v;
  do {
    r = *x;
    /* The following calculation is not atomic
       but it is side-effect free */
    v = f(r);
    /* invariant : v == f(r) */
    c = CAS1(x, r, v);
  } while (c != r);
}
```

FIG. 7.3 – Affectation avec CAS_1

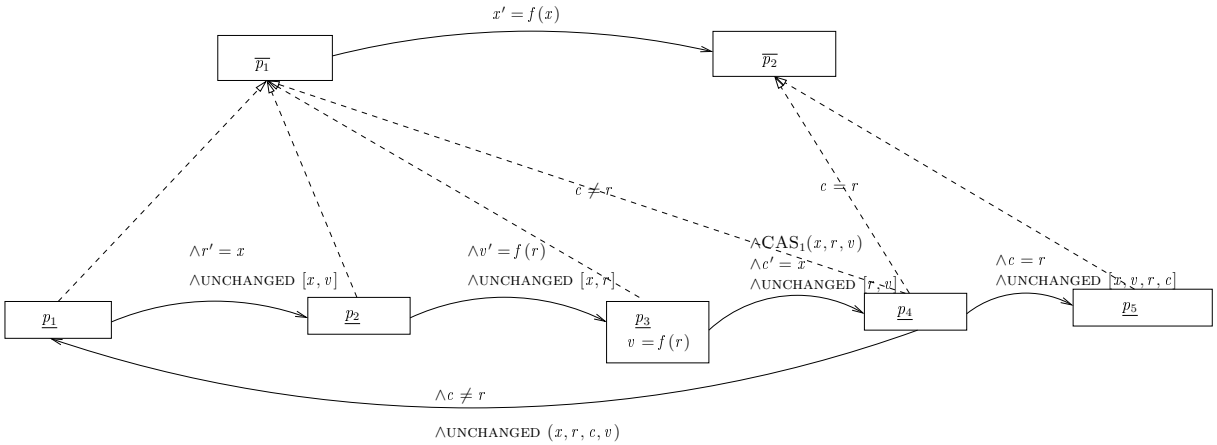


FIG. 7.4 – Simple non-locking Assignment

Cette section reprend l'exemple traité dans le chapitre 4 avec les autres méthodes. Cet algorithme consiste à remplacer l'affectation $x = f(x)$, où f est une fonction sans effet de bord mais non calculable en un seul cycle de processeur, par l'algorithme de la figure 7.3. Cet exemple est assez typique des algorithmes sans verrou à modèle concurrent optimiste. En effet, il introduit une boucle mais l'on suppose que cette boucle sera exécutée un faible nombre de fois. C'est-à-dire que la *contention* sera faible. La boucle consiste à copier la valeur de la variable globale x dans une variable locale r, calculer en local $v = f(r)$ puis, en utilisant CAS_1, effectuer l'affectation en affectant v à x. Bien évidemment si la valeur de x a changé, l'opération CAS_1 échouera et on recommence de nouveau.

La modélisation est donnée en figure 7.4. La spécification est l'algorithme abstrait du haut de la figure. L'algorithme concret est celui du bas. Comme on a pu le voir, l'exemple est relativement simple et pourtant son traitement dans les autres méthodes n'est pas si aisé. Conceptuellement, il n'y a pas de difficultés majeures. En effet, l'invariant local en $\underline{p}_3$ est facile à trouver. De plus, le point de linéarisabilité est lui aussi évidemment la transition effectuant CAS_1(quand il réussit). Il suffit donc de représenter que la place $\underline{p}_4$ raffine les places $\overline{p}_1$ *et* $\overline{p}_2$. Plus précisément si $c = r$ alors $\underline{p}_4$ raffine $\overline{p}_1$ sinon c'est $\overline{p}_2$. On remarque par ailleurs bien le fait que l'algorithme est modélisé par son flot de contrôle.

Il y a en tout 85 obligations de preuve de générées. Elles sont toutes automatiquement validées par haRVey. L'exportation dédiée vers Isabelle est néanmoins bénéfique lors de la modélisation afin de comprendre les erreurs de modélisation quand elles ne sont pas valides. Les prouveurs génèrent bien des contre-exemples lorsque l'obligation de preuve n'est pas valide, mais le plus souvent on a plutôt

besoin de savoir quelle est la sous-formule qui n'est pas valide. Cette problématique rejoint celle de la recherche automatique d'invariant. Il faut en effet rechercher une sous-formule non triviale qui rendrait valide l'obligation de preuve.

7.3 Affectation avec descripteur

Une astuce couramment utilisée dans les algorithmes sans verrou est l'utilisation de descripteurs. Un descripteur est une structure de donnée contenant toutes les données nécessaires afin d'effectuer une opération. On utilise un descripteur en deux temps. Dans le premier on l'installe à la place du résultat. Dans le deuxième temps, on remplace ce descripteur par le résultat de l'opération qu'il décrit. Évidemment il faut que les autres processus respectent les descripteurs. Pour cela, ils doivent enlever un descripteur s'ils en rencontrent un.

Par exemple, que doit contenir un descripteur si l'opération à effectuer est le calcul $x' = f(x, y)$ avec f une fonction sans effet de bord ? Un descripteur devra avoir un champ *old* contenant la valeur de x avant l'installation du descripteur. Un descripteur ne doit servir qu'une seule fois afin d'éviter le problème ABA évoqué dans la section 3.7. Dans notre modélisation, nous rajoutons donc un champ *pid* qui permet de le rendre unique à chaque appel de l'opération. En effet, on modélise un appel par processus. Mais du fait du raffinement et du *rely*, on peut le remplacer de manière sûre dans tout autre algorithme. Par ailleurs, les descripteurs ne doivent pas faire partie du co-domaine de f. On les définirait comme suit, néanmoins dans la formalisation on suppose que la fonction f est globale.

```
typedef struct {
  word_t old;
  Pid pid;
  word_t (*f)(word_t x, word_t y);
} Descriptor;
```

On définit la logique des descripteurs avec les axiomes suivants. Ils définissent un constructeur *newDesc*, et des accesseurs (*getter*) pour les champs *pid* et *old*. De plus, il existe des valeurs qui ne sont pas des descripteurs. Enfin les descripteurs ne font pas partie du co-domaine de f.

- $\forall o\ pid.\ desc(newDesc(o, pid))$
- $\forall o\ pid.\ o = getOld(newDesc(o, pid))$
- $\forall o\ pid.\ pid = getPid(newDesc(o, pid))$
- $\exists v.\ \neg desc(v)$
- $\forall x\ y.\ \neg desc(x) \wedge \neg desc(y) \implies \neg desc(f(x, y))$

Dans le cas purement séquentiel, nous aurions les triplets de Hoare suivant :

$$\frac{\{x = v\}x := d\{x = d \wedge getOld(d) = v\} \qquad \{x = d \wedge getOld(d) = v\}x := f(getOld(d), y)\{x = f(v, y)\}}{\{x = v\}x := d; x := f(getOld(d), y)\{x = f(v, y)\}}\ \text{Séquence}$$

Dans un environnement parallèle, nous ne sommes pas sûr que le descripteur soit encore en place, nous utilisons donc la primitive CAS_1 pour ne modifier x que si le descripteur est encore là. Le programme (qui nous servira dans le premier raffinement) devient donc :

```
void assignWithDesc1(Descriptor d) {
  x = d;
  CAS1(d.x, d, d.f(getOld(d), d.y));
}
```

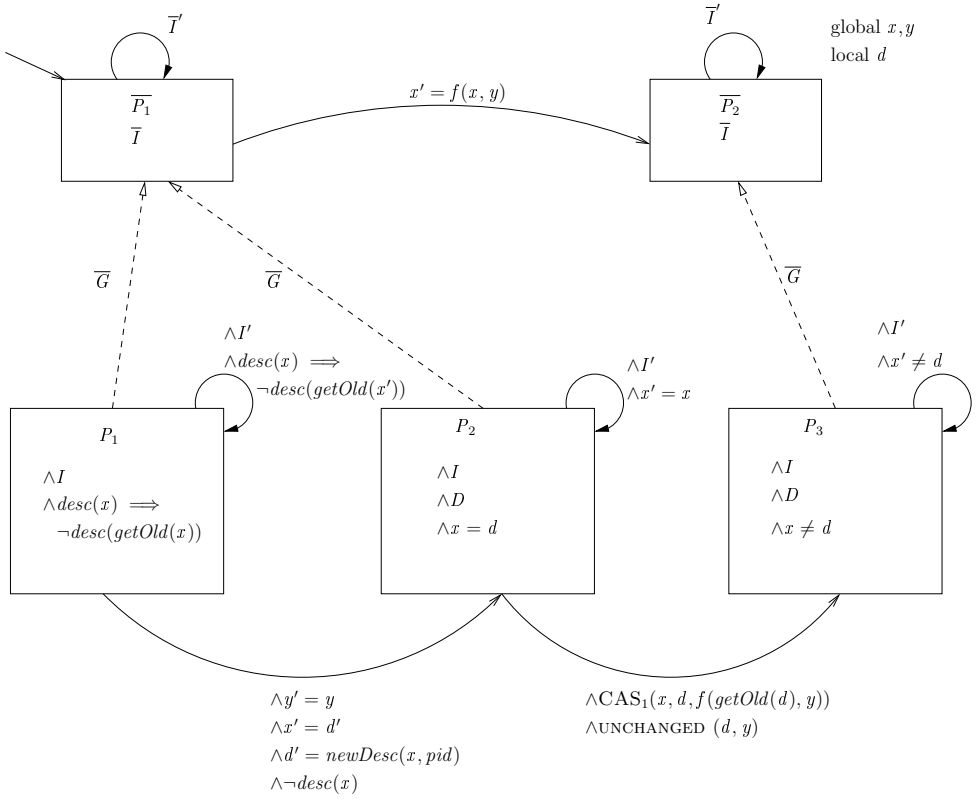
Mais ce programme impose toujours aux autres programmes de ne pas modifier x si $x = d$. Mais l'on peut se rendre compte que si l'on applique deux fois $\mathrm{CAS}_1(x, d, f(getOld(x), y)$, on obtient toujours le même résultat. Les autres processus peuvent donc eux-mêmes enlever un descripteur sans risque. Le programme à prouver est donc celui de la figure 7.5. Il commence par enlever (ou non) les descripteurs qu'il rencontre, jusqu'à ce qu'il puisse installer le sien. Alors il pourra calculer le résultat final en supposant la valeur de x figée. Enfin il remplace le descripteur (s'il est encore en place) par le résultat.

```
void assignWithDesc1(x,y,f) {
while desc(x) {
  choose :
    • CAS1(x,d,f(getOld(x),y))
    • SKIP
};
x := d || d := newDesc(x,pid);
CAS1(x,d,f(getOld(d),y))
}
```

FIG. 7.5 – Algorithme concret d'affectation avec descripteurs. Voir aussi le 2ème raffinement en figure 7.7. Ce code n'est qu'une approximation car il n'est pas possible de représenter proprement le grain d'atomicité. Il faut considérer la sortie de la boucle et l'installation du descripteur comme atomique. x n'est donc pas un descripteur au moment de l'installation de d.

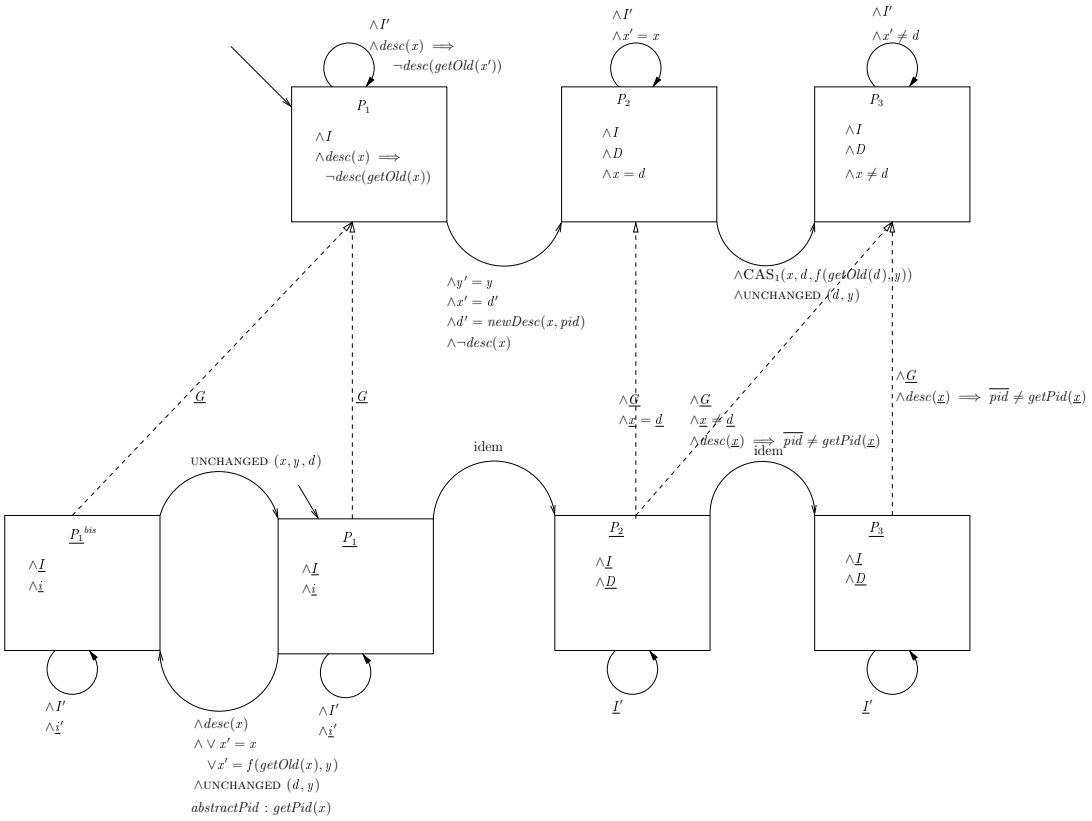
Comme dans l'explication, la preuve a été faite en deux temps. Dans le premier raffinement sur la figure 7.6, qui est très classique, on installe puis enlève le descripteur mais cette spécification n'est pas aisément composable avec d'autres. Dans le deuxième raffinement sur la figure 7.7, l'algorithme commence à enlever les descripteurs des autres. Le deuxième raffinement, bien que très simple à trouver, est plus technique. Tout d'abord il utilise le fait qu'un algorithme puisse effectuer le pas d'un autre. En effet, le pas de $\underline{p_1}$ à ${\underline{p_1}}^{bis}$ effectue le pas de l'algorithme dont le *pid* est $getPid(x)$. À ce moment là, $\underline{p_2}$ ne raffine plus seulement p_2, mais aussi p_3. L'invariant de collage est donc complété pour indiquer que le descripteur a été enlevé et ne peut donc pas être réutilisé.

Cet exemple génère 88 obligations de preuves et il illustre un usage typique des descripteurs. Il montre aussi la facilité de la preuve de la correction. En effet, comme nous allons le voir dans la prochaine section, peu d'algorithmes usant de descripteurs ainsi ont été prouvés. Au mieux, on a utilisé du *model-checking* en utilisant des simplifications qui ont ici leurs justifications. Ces simplifications servent à réduire l'espace d'états à vérifier. Malgré ces simplifications, le *model-checking* ne peut vérifier l'algorithme qu'avec un nombre de processus limité. De plus, aucune justification *générale* ne permet d'étendre la correction d'un petit nombre de processus à un nombre quelconque. Donc même si on a vérifié par *model-checking* que l'exécution est correcte à 3 processus, on ne peut rien en déduire pour n quelconque. Au cas par cas cela peut donc être possible, mais notre méthode reste valable et peut même servir de cadre afin de justifier cela.



$$
\begin{array}{ll}
\overline{I} \triangleq & \wedge \neg desc(x) \\
& \wedge \neg desc(y) \\
I \triangleq & \neg desc(y) \\
D \triangleq & \wedge pid = getPid(d) \\
& \wedge desc(d) \\
& \wedge \neg desc(getOld(d)) \\
\overline{G} \triangleq & \wedge \overline{y} = \underline{y} \\
& \wedge \text{IF } desc(\underline{x}) \text{ THEN } \overline{x} = getOld(\underline{x}) \text{ ELSE } \overline{x} = \underline{x}
\end{array}
$$

FIG. 7.6 – Premier raffinement de l'affectation avec descripteur.



$$
\begin{array}{lll}
I & \triangleq & \neg desc(y) \\
D & \triangleq & \wedge pid = getPid(d) \\
 & & \wedge desc(d) \\
 & & \wedge \neg desc(getOld(d)) \\
\underline{G} & \triangleq & \wedge \overline{y} = \underline{y} \\
 & & \wedge \overline{x} = \underline{x} \\
 & & \wedge \overline{d} = \underline{d} \\
\underline{I} & \triangleq & \wedge \neg desc(y) \\
 & & \wedge desc(x) \implies \neg desc(getOld(x)) \\
\underline{D} & \triangleq & \wedge pid = getPid(pid)) \\
 & & \wedge desc(d) \\
 & & \wedge \neg desc(getOld(d)) \\
\underline{i} & \triangleq & desc(x) \implies pid \neq getPid(x)
\end{array}
$$

FIG. 7.7 – Second raffinement de l'affectation avec descripteur.

7.4 RDCSS

```
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;
}
```

FIG. 7.8 – Définition séquentielle de RDCSS

Cet exemple, tiré de [28] a été l'objectif tout au long de la thèse. En effet, il est à la fois très simple à comprendre et difficile à prouver. Les auteurs n'ont d'ailleurs utilisé le *model-checking* que sur des instances particulières (maximum 3 processus et avec des hypothèses de simplification). L'idée de l'article est de proposer une implémentation de CAS_2 à partir de CAS_1. Pour cela, les auteurs vont utiliser deux fois le principe des descripteurs tout en s'aidant d'une fonction intermédiaire : RDCSS. RDCSS signifie *Reduce Double-Compare-And-Single-Swap* et sa définition séquentielle est donnée en figure 7.8. RDCSS compare donc les valeurs de 2 pointeurs avec 2 valeurs de référence, et ne modifie que la valeur du deuxième pointeur si la comparaison indique l'égalité.

```
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;
}

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

FIG. 7.9 – Implémentation de RDCSS à base de descripteurs et de CAS_1

Les auteurs proposent l'implémentation donnée en figure 7.9. Comme l'exemple précédent, il agit en deux temps. Il cherche tout d'abord à installer un descripteur. S'il échoue parce qu'il y en a déjà un en place, il l'enlève puis réessaie. Ensuite il cherche à enlever son descripteur. De plus, il optimise ce processus par le fait que si a_2 est différent de o_2 alors il ne fait rien. En fait, on retrouve presque l'algorithme de l'affectation avec descripteurs. C'est effectivement le cas si l'on pose :

$$f_{o_1,o_2,n_2}(a_2, a_1) \triangleq \text{IF } a_2 = o_2 \wedge a_1 = o_1 \text{ THEN } n_2 \text{ ELSE } a_2$$

Ne restent alors que trois différences. La première concerne la conditionnelle après la boucle cherchant à installer le descripteur. En effet, si $r \neq d \rightarrow o_2$ alors forcément le descripteur n'a pas été installé et les CAS_1 de la fonction "Complete" échoueront. Le point de linéarisation est alors l'exécution du CAS_1

d'installation. La deuxième est la double utilisation de CAS_1 dans la fonction "Complete" justement. Mais il est évident que les deux programmes suivant sont équivalents, v et d étant des variables locales :

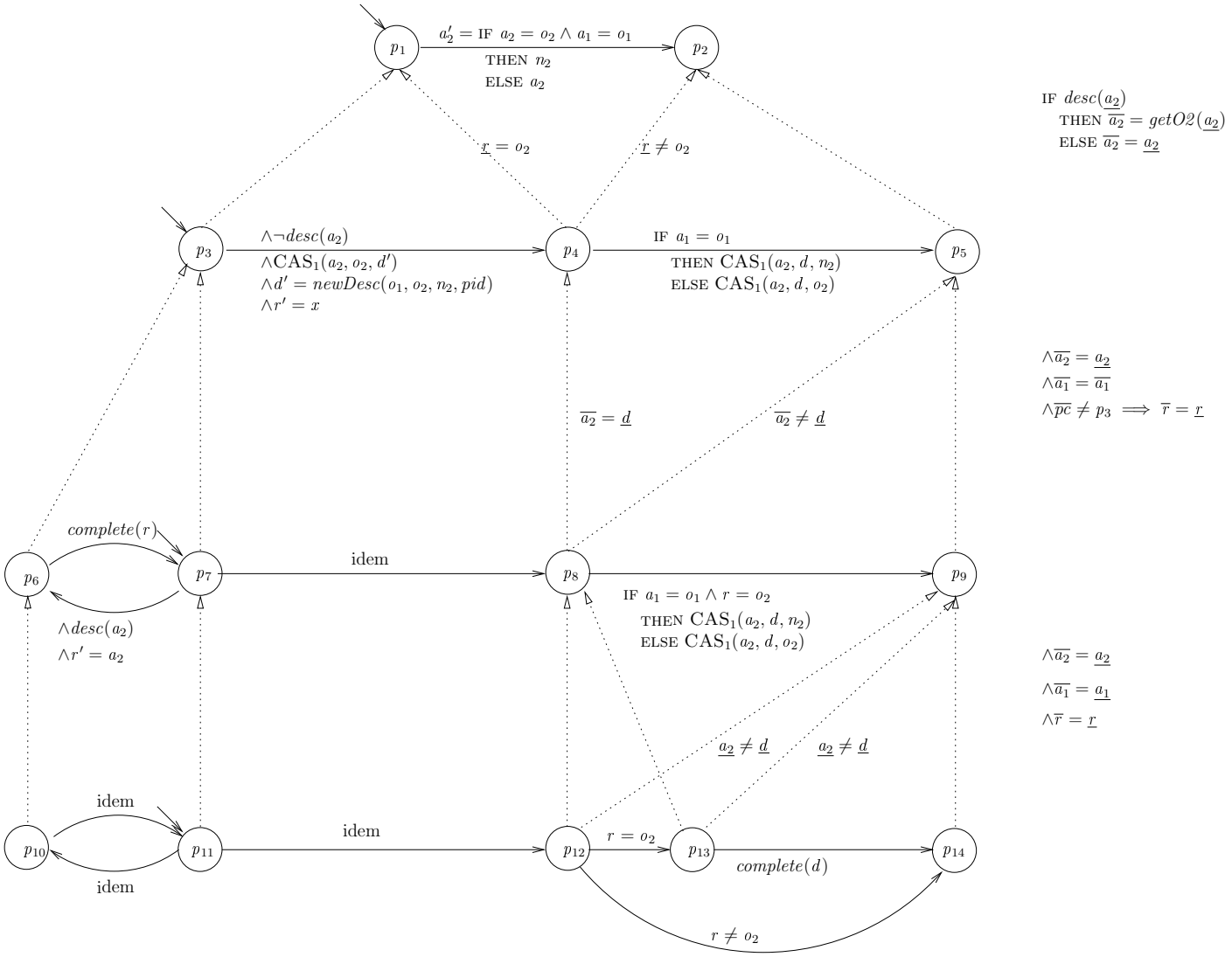
```
if (v == d->o1) {
  CAS1(d->a2,d,d->n2); // C2
} else {
  CAS1(d->a2,d,d->o2); // C3
}
```

```
CAS1(d->a2, d,(v == d->o1)?d->n2 :d->o2);
```

La dernière différence est que l'algorithme proposé combine à la fois l'affectation simple du descripteur (cf section 7.2) et la boucle qui enlève le descripteur de l'algorithme de la section 7.3.

Ce développement démontre que l'algorithme de [28] est dérivé de l'affectation avec descripteurs. Il est bien sûr possible de montrer le lien de raffinement entre les deux algorithmes. Néanmoins, pour comprendre le RDCSS seul, la figure 7.10 est plus intelligible. Le raffinement complet y est repris. Là encore toutes les preuves ont été automatiques. Le niveau concret est très proche de l'algorithme à démontrer. Il faudrait maintenant prouver le raffinement de l'algorithme implémentant la fonction *complete*. Deux possibilités sont offertes : soit on prouve l'implémentation en rajoutant un ou deux raffinements à ceux de la figure 7.10, soit on la prouve dans un développement à part. En effet, comme on démontre le raffinement pour toutes traces concrètes, on peut directement "remplacer" une transition par un algorithme prouvé par ailleurs. Il suffit "juste" de vérifier les *rely*. Cette dernière étape n'a pas été accomplie, étant très similaire aux autres cas et ne présentant pas de difficultés particulières.

Ces trois exemples ont permis de montrer la simplicité d'expression et d'utilisation de la méthode. A tout moment le raisonnement est localisé et donc simple. La compositionnalité et le raffinement sont malgré tout conservés. De plus, les obligations peuvent être déchargées automatiquement, ce qui soulage le travail de vérification, comparé aux autres méthodes. Outre les points communs sur les concepts, le seul point commun gênant reste la modélisation en elle-même. En effet, modéliser un système quel qu'il soit, n'est pas trivial, aussi bons que soient la méthode et l'outil. Cela nécessite de faire des choix non naturels pour un programmeur. De plus, le moment d'insertion des détails au cours des raffinements n'est pas quelconque. Un introduction trop tardive peut être impossible. À l'inverse, trop tôt cela complique le modèle. La part d'expérience est donc très importante, et a un impact très grand. Une aide pourrait consister en une bibliothèque de schémas de raffinements (affectation, variables temporaires, descripteurs, etc).



$\text{CAS}_1(a_1, o_1, n_1) \triangleq \text{IF } a_1 = o_1 \text{ THEN } a_1' = n_1 \text{ ELSE } a_1' = a_1$

$complete(v) \triangleq \text{IF } a_1 = getO_1(v) \text{ THEN } \text{CAS}_1(a_2, v, getN_2(v)) \text{ ELSE } \text{CAS}_1(a_2, v, getN_2(v))$

$I(p_2) = I(p_5) = I(p_9) = I(p_{14}) \triangleq a_2 \neq d$

$I(p_4) \triangleq ((a_2 = d \wedge r = o_2) \vee (a_2 \neq d \wedge r \neq o_2))$

$R(p_2) = R(p_5) = R(p_9) = R(p_{14}) \triangleq a_2' \neq d$

$R(p_4) \triangleq \text{IF } a_2 = d \text{ THEN } a_2' = a_2 \text{ ELSE } a_2' \neq d$

$R(p_8) = R(p_{12}) \triangleq a_2' = a_2 \vee (a_2 = d \wedge complete(a_2))$

$I(p_8) = I(p_{12}) \triangleq a_2 = d \implies r = o2$

$I(p_{11}) = I(p_7) = I(p_6) \triangleq desc(a_2) \implies getPid(a_2) \neq pid$

FIG. 7.10 – Raffinements du RDCSS. Les annotations sont simplifiées à des fins de lisibilité. Par ailleurs, les invariants et les *rely* concernant les descripteurs sont omis. Ceux-ci doivent assurer qu'un descripteur ne contient pas lui-même de descripteurs. De plus, les champs o_1, o_2 et n_2 doivent évidemment avoir les mêmes valeurs que les variables locales du même nom.

Chapitre 8

Conclusion et travaux futurs

Cette thèse présente une méthode dédiée au développement de structures de données sans verrou. Les algorithmes utilisés pour implémenter les opérations de ces structures sont relativement astucieux. Ils sont donc difficiles à concevoir, difficultés principalement dues aux nombreux entrelacements possibles.

L'exclusion mutuelle nous permet d'implémenter correctement des structures de données correctes. Mais elle empêche d'utiliser toute la puissance mise à disposition par nos machines qui sont intrinsèquement parallèles puisque du point de vue de la structure elle supprime le parallélisme.

Cet aperçu nous a montré la nécessité de définir précisément ce qu'est une spécification et les propriétés désirées. Nous avons donc présenté formellement ce qu'est une structure de donnée au travers des types abstraits. La *linéarisabilité* est alors apparue comme une propriété simplifiant le raisonnement sur les programmes parallèles. En effet, elle indique que l'effet d'une opération doit paraître comme *atomique* entre l'appel et le retour de la méthode implémentant cette opération.

La linéarisabilité dépend beaucoup de l'architecture matérielle utilisée. Elle est notamment plus facile à implémenter si les processeurs disposent de primitives spéciales à la synchronisation de processus. Une primitive remarquable est CAS_1. En effet, elle permet d'implémenter toutes les autres primitives existantes. Ces précisions nous ont permis de définir le cadre formel dans lequel nous allons faire la preuve de nos implémentations.

Ces preuves s'appuient sur des méthodes formelles de développement. Un petit exemple nous a permis de présenter l'état de l'art de la vérification de programmes concurrents, et ainsi de comparer chacune de ces méthodes. Deux éléments essentiels ressortent : le raffinement et la compositionnalité. Ces deux éléments facilitent grandement la vérification des programmes concurrents. Le raffinement permet en effet d'introduire petit à petit des détails d'implémentation. Il facilite donc à la fois la compréhension et la documentation de la conception. La compositionnalité simplifie elle aussi le raisonnement en permettant de séparer les différents composants du système.

Mais à l'usage, un troisième élément est important : l'automatisation des preuves. Une preuve est toujours difficile à faire, son automatisation est donc souhaitable. Le *model-checking* est d'ailleurs un compromis entre vérification formelle et automatisation, d'où l'engouement pour celui-ci.

Malheureusement aucune des méthodes présentées dans le chapitre 4 ne présente toutes ces caractéristiques. Notre méthode tire donc son essence de chacune des méthodes précédentes. Elle utilise ainsi le raffinement pour démontrer la linéarisabilité. Elle est compositionnelle grâce au principe du *rely-guarantee*. De plus, les obligations de preuves générées peuvent être déchargées automatiquement. Mais la conception d'une telle méthode est sujette aux erreurs. La preuve de correction de notre méthode est donc une part importante du travail présenté. Cette preuve a aussi été vérifiée avec l'assistant à la preuve Isabelle/HOL. Cet assistant est constitué d'un petit noyau de code assurant l'exactitude des preuves écrites. La confiance en les preuves est donc beaucoup plus grande qu'avec une preuve "papier".

Les fondations de notre méthode étant assurées, nous l'appliquons dans le chapitre 7 à différents exemples. Le premier est un schéma d'algorithme très courant. Le deuxième utilise les *descripteurs*, une astuce commune dans les algorithmes sans verrou. Enfin la preuve de l'algorithme RDCSS est présentée. Cet algorithme non trivial permettra d'implémenter CAS_n à partir de CAS_1. Ces exemples mettent en lumière les promesses de notre méthode. L'expression des algorithmes est simple, les preuves

automatiques, les spécifications composables.

Les travaux futurs s'organisent autour de plusieurs axes. Le premier consiste à prouver d'autres algorithmes. Parmi les premiers exemples à traiter, il faudrait bien sûr vérifier l'implémentation de CAS_n à partir de RDCSS [28]. Ensuite, on pourrait s'intéresser aux nombreuses librairies [36, 41, 73, 8, 72, 33, 79, 11] et prouver leurs implémentations.

Par ailleurs, les obligations de preuves pourraient être un peu simplifiées, ce qui constitue un deuxième axe. En effet, comme on l'a vu dans la section 6.1, il est obligatoire de vérifier que chaque pas concret conserve la relation de collage de tous les autres processus. Cette quantification est coûteuse en temps pour les prouveurs. Or si l'invariant de collage était stable sous un *rely* concret, on pourrait ne pas vérifier cette partie.

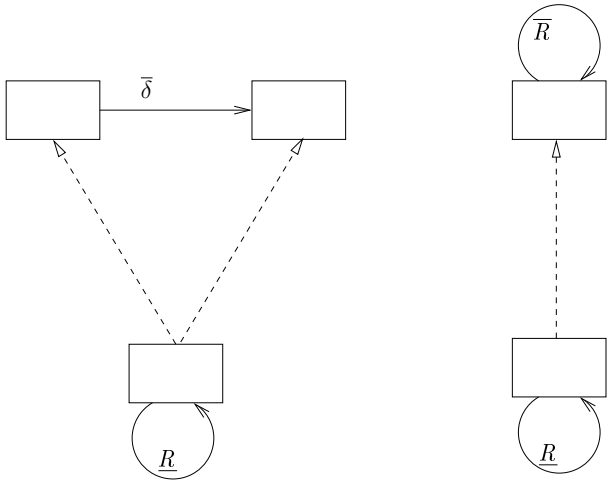


FIG. 8.1 – Différent cas de raffinement impliquant un *rely* concret.

Ainsi dans les deux cas de la figure 8.1, il faut qu'un *rely* concret implique soit un *rely* abstrait, soit un pas abstrait. Comme on vérifie déjà que chaque pas concret implique le *rely* concret de toutes les places, nous aurions la garantie que l'invariant de collage est encore vérifié après. Il y a deux écueils prévisibles. Le premier est que la sémantique des traces serait changée. En effet, plusieurs processus abstraits pourraient alors effectuer un pas abstrait en même temps. Ce que l'on avait interdit puisque l'on désirait être sûr de la linéarisabilité. A priori si cela est possible cela signifie que l'ordre de ces pas abstraits est interchangeable et donc que la linéarisabilité est conservée. Mais il faudrait étudier plus attentivement cette question. Le deuxième écueil est que, sans changer la modélisation, les obligations ne seraient jamais valides.

Ainsi, dans l'exemple de l'affectation avec descripteur, et pour le collage des places $(p_1, \overline{p_1})$, on devrait avoir quelque chose comme : l'invariant abstrait étant vérifié, l'invariant concret étant vérifié, l'invariant de collage étant vérifié, le *rely* étant vérifié, cela implique le *rely* abstrait et le collage après.

$$
\begin{array}{lll}
\forall \underline{x}\ \overline{x}\ \underline{y}\ \overline{y}\ \underline{x}'\ \overline{x}'. & \wedge\ \neg desc(\overline{x}) \wedge \neg desc(\overline{y}) & \overline{I}(\overline{p_1}) \\
& \wedge\ \neg desc(\underline{x}) \wedge (desc(\underline{x}) \implies \neg desc(getOld(\underline{x}))) & \underline{I}(p_1) \\
& \wedge\ \overline{y} = \underline{y} \wedge \text{IF } desc(\underline{x}) \text{ THEN } \overline{x} = getOld(\underline{x}) \text{ ELSE } \overline{x} = \underline{x} & G(p_1, \overline{p_1}) \\
& \wedge\ \neg desc(\underline{y}') \wedge (desc(\underline{x}') \implies \neg desc(getOld(\underline{x}'))) & \underline{R}(p_1) \\
\implies & & \\
& \wedge\ \neg desc(\overline{x'}) \wedge \neg desc(\overline{y'}) & \overline{R}(\overline{p_1}) \\
& \wedge\ \overline{y}' = \underline{y}' \wedge \text{IF } desc(\underline{x}') \text{ THEN } \overline{x}' = getOld(\underline{x}') \text{ ELSE } \overline{x}' = \underline{x}' & G(p_1, \overline{p_1})'
\end{array}
$$

Bien entendu cette obligation de preuve n'est pas valide. En effet, il manque une relation liant les valeurs des variables abstraites de l'état d'après avec les variables concrètes d'après. Éclater l'invariant de collage entre une partie locale et une partie globale pourrait certainement résoudre le problème. Ainsi il serait possible de simplifier la définition 18 en omettant la partie sur les autres processus. En conséquence, les obligations de preuves seraient elles-mêmes plus simples et donc encore plus facilement automatisables.

Cet argument pourrait paraître comme contradictoire avec les résultats du chapitre précédent. Mais dans ces exemples, il n'y a pas de théories particulières d'utilisées. Rien n'assure en effet que d'autres prouveurs soient aussi performants dans ces cas-là.

Un troisième axe possible est l'intégration de la vérification de propriétés de vivacité. En effet, les algorithmes pourraient être complétés avec des annotations d'équité de manière similaire à ce qui a été fait avec les diagrammes de prédicats [9]. Un des buts serait de pouvoir classer les algorithmes dans les catégories *wait-free*, *lock-free* et *obstruction-free* présentées au chapitre 2. Cette approche temporelle ne devrait être qu'un premier pas vers une analyse des temps d'exécutions. En effet, une grandeur utile à la comparaison d'implémentations de structures de données sans verrou est le nombre de cycle CPU par opérations [24]. Cette grandeur dépend de la contention. Généralement, ces algorithmes ont une approche optimiste de la concurrence. Les conflits sont considérés comme rare. Étant donné une probabilité des ces conflits, il faudrait pouvoir calculer le temps moyens d'une opération. Un utilisateur pourrait alors aisément faire un choix de l'implémentation à utiliser en fonction de la contention de son problème.

Enfin le dernier axe, et probablement le plus ambitieux, serait de développer le raffinement sur des modèles mémoires non-standard. En effet, le modèle de mémoire partagée utilisé est à la fois le plus satisfaisant intellectuellement, mais aussi le moins usité au niveau matériel. Les architectures les plus courantes n'implémentent malheureusement pas un tel modèle (voir la section 3.6). Par exemple, le modèle mémoire de la machine virtuelle de Java est assez particulier [52]. Les différences se situent principalement dans la propagation des effets d'une lecture ou d'une écriture qui ne sont pas immédiatement visibles par les autres processus. De même, les instructions peuvent être réordonnées et donc les lectures, par exemple, survenir plus tôt par rapport à l'ordre donné par l'algorithme.

Pour modéliser ces modèles mémoires [52, 62, 37, 3], il serait nécessaire de modéliser les mémoires caches des processus. La mémoire globale pourrait donc être "distribuée" dans les mémoires locales. Des contraintes, possiblement incluses dans les invariants, assureraient alors la cohérence de ceux-ci vis à vis de la mémoire globale. La difficulté réside dans l'expression des contraintes. En effet, un cache peut être dans plusieurs états différents. On ne considère alors plus un état unique de la mémoire locale (resp. globale) mais un ensemble d'états possibles.

De manière générale, la vérification formelle est encore trop coûteuse en temps et en hommes. Elle nécessite des compétences pointues pour un résultat qui n'est actuellement pas à la hauteur des enjeux. Comme le fait le *model-checking*, il est nécessaire de faciliter au maximum toutes les étapes, que ce soit l'étape de modélisation comme celle de la preuve. Cette thèse entend répondre à l'automatisation des preuves en organisant la modélisation dans un cadre précis. De nombreuses pistes existent, comme la création d'une librairie de schémas d'implémentations réutilisables, afin de simplifier encore plus le processus et faciliter ainsi la vérification de programmes parallèles.

Annexe A

Affectation en Assume-guarantee

A.1 Assignment

theory *RG_Assign* **imports** *RG_Syntax* **begin**

lemmas *definitions [simp]= stable_def Pre_def Rely_def Guar_def Post_def Com_def*

A.1.1 Atomic assignement

```
record AtomicAssign =
  x  :: nat

lemma AtomicAssign:
shows "⊢ ´x := f ´x sat [{ True } ,{ True } ,{ᵃx=f ᵒx ∨ ᵃx=ᵒx } ,{ True } ]"
proof (rule Basic)
  show "{True} ⊆ {´(AtomicAssign.x_update (f ´AtomicAssign.x))  ∈  {True}}"
    by auto
  next
    show "stable {True} {True}"
      by auto
  next
    show "stable {True} {True}"
      by auto
  next
    show "{(s, t). s  ∈  {True} ∧ (t = s(|AtomicAssign.x := f (AtomicAssign.x s)|) ∨ t = s)} ⊆ {ᵃx
= f ᵒx ∨ ᵃx = ᵒx}"
      by (simp, auto)
qed
```

A.1.2 Atomic assignement with CAS

```
record AssignWithCAS =
  x  :: nat
  v  :: nat
  fv :: nat
  b  :: nat


lemma AssignWithCAS:
  shows "⊢ ´b := (0::nat);; WHILE ´b = (0::nat) DO ´v := ´x;; ´fv := f ´v;; ´b := BCAS ´x, ´v,
´fv SACB OD sat [{ True } ,{ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv} ,{ᵃx=ᵒx ∨ ᵃx=f ᵒx} ,{ True } ]"
proof (rule_tac mid="{ ´b = 0 }" in Seq)
  show "⊢ ´b := 0 sat [{ True } ,{ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv} ,{ᵃx=ᵒx ∨ ᵃx=f ᵒx } ,{ ´b = 0 } ]"
```

```
    by (rule Basic, auto)
next
  show "⊢ WHILE ´b = (0::nat) DO ´v := ´x;; ´fv := f ´v;; ´b := BCAS ´x, ´v, ´fv SACB OD sat [{|
´b = 0 |} ,{|ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv|} ,{|ᵃx=ᵒx ∨ ᵃx=f ᵒx|} ,{| True |}]"
  proof (rule_tac pre'="{| True |}" and
      guar'="{| ᵃx=ᵒx ∨ ᵃx=f ᵒx |}" and
      rely'="{| ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv |}" and
      post'="{| True |}" in Conseq)
    show "{| ´b = 0 |} ⊆ {| True |}"
      by auto
  next
    show "{|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|} ⊆ {|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}"
      by auto
  next
    show "{|ᵃx = ᵒx ∨ ᵃx = f ᵒx|} ⊆ {|ᵃx = ᵒx ∨ ᵃx = f ᵒx|}"
      by auto
  next
    show "{|True|} ⊆ {|True|}"
      by auto
  next
    show "⊢ WHILE ´b = 0 DO ´v := ´x;; ´fv := f ´v;; ´b := BCAS ´x, ´v, ´fv SACB OD sat [{| True
|} ,{|ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv|} ,{|ᵃx=ᵒx ∨ ᵃx=f ᵒx|} ,{| True |}]"
    proof (rule While)
      show "stable {|True|} {|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}"
        by auto
    next
      show "{|True|} ∩ - {|´b = 0|} ⊆ {|True|}"
        by auto
    next
      show "stable {|True|} {|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}"
        by auto
    next
      show "∀s. (s, s) ∈ {|ᵃx = ᵒx ∨ ᵃx = f ᵒx|}"
        by auto
    next
      show "⊢ ´v := ´x;; ´fv := f ´v;; ⟨IF ´x = ´v THEN ´x := ´fv;; ´b := 1 ELSE ´b := 0 FI⟩ sat
[{|True|} ∩ {|´b = 0|}, {|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}, {|ᵃx = ᵒx ∨ ᵃx = f ᵒx|}, {|True|}]"
      proof (rule_tac pre'="{| ´b = 0 |}" and
          guar'="{| ᵃx=ᵒx ∨ ᵃx=f ᵒx |}" and
          rely'="{| ᵃv=ᵒv ∧ ᵃb=ᵒb ∧ ᵃfv=ᵒfv |}" and
          post'="{| True |}" in Conseq)
        show "{|True|} ∩ {|´b = 0|} ⊆ {|´b = 0|}"
          by auto
      next
        show "{|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|} ⊆ {|ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}"
          by auto
      next
        show "{|ᵃx = ᵒx ∨ ᵃx = f ᵒx|} ⊆ {|ᵃx = ᵒx ∨ ᵃx = f ᵒx|}"
          by auto
      next
        show "{|True|} ⊆ {|True|}"
          by auto
      next
        show "⊢ ´v := ´x;; ´fv := f ´v;; ⟨IF ´x = ´v THEN ´x := ´fv;; ´b := 1 ELSE ´b := 0 FI⟩
sat [{|´b = 0|}, {|ᵃv = ᵒv ∧  ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}, {|ᵃx = ᵒx ∨ ᵃx = f ᵒx|}, {|True|}]"
        proof (rule_tac mid="{| ´fv = f ´v  |}" in Seq)
          show "⊢ ´v := ´x;; ´fv := f ´v sat [{|´b = 0|}, {|ᵃv = ᵒv ∧  ᵃb = ᵒb ∧ ᵃfv = ᵒfv|}, {|ᵃx
= ᵒx ∨ ᵃx = f ᵒx|}, {| ´fv = f ´v |}]"
```

```
          proof (rule_tac mid="⦃True⦄" in Seq)
            show "⊢ ´v := ´x sat [⦃´b = 0⦄, ⦃ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv⦄, ⦃ᵃx = ᵒx ∨ ᵃx =
f ᵒx⦄, ⦃ True ⦄]"
              by (rule Basic, auto)
          next
            show "⊢ ´fv := f ´v sat [⦃True⦄, ⦃ᵃv = ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv⦄, ⦃ᵃx = ᵒx ∨ ᵃx =
f ᵒx⦄, ⦃ ´fv = f ´v ⦄]"
              by (rule Basic, auto)
          qed
        next
          show "⊢ ⟨IF ´x = ´v THEN ´x := ´fv;; ´b := 1 ELSE ´b := 0 FI⟩ sat [⦃´fv = f ´v ⦄, ⦃ᵃv
= ᵒv ∧ ᵃb = ᵒb ∧ ᵃfv = ᵒfv⦄, ⦃ᵃx = ᵒx ∨ ᵃx = f ᵒx⦄, ⦃True⦄]"
          proof (rule Await,auto)
            show "⋀V. ⊢ IF ´x = ´v THEN ´x := ´fv;; ´b := Suc 0 ELSE ´b := 0 FI sat [⦃´fv = f
´v⦄ ∩ {V}, {(s, t). s = t}, UNIV, ⦃´x = x V ∨ ´x = f (x V)⦄]"
              proof (rule Cond, auto)
                show "⋀V. ⊢ ´x := ´fv;; ´b := Suc 0 sat [⦃´fv = f ´v⦄ ∩ {V} ∩  ⦃´x = ´v⦄, {(s,
t). s = t}, UNIV, ⦃´x = x V ∨ ´x = f (x V)⦄]"
                proof (rule_tac mid="⦃´x = x V ∨ ´x = f (x V)⦄" in Seq)
                  show "⋀V. ⊢ ´x := ´fv sat [⦃´fv = f ´v⦄ ∩ {V} ∩ ⦃´x = ´v⦄, {(s, t). s = t}, UNIV,⦃´x
= x V ∨ ´x = f (x V)⦄ ]"
                    by (rule Basic, auto)
                next
                  show "⋀V. ⊢ ´b :=  Suc 0 sat [⦃´x = x V ∨ ´x = f (x V)⦄, {(s, t). s = t}, UNIV,
⦃´x = x V ∨ ´x = f (x V)⦄]"
                    by (rule Basic, auto)
                qed
              next
                show " ⋀V. ⊢ ´b := 0 sat [⦃´fv = f ´v⦄ ∩ {V} ∩   - ⦃´x = ´v⦄, {(s, t). s = t}, UNIV,
⦃´x = x V ∨ ´x = f (x V)⦄]"
                  by (rule Basic, auto)
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed

end
```

Annexe B

Affectation en B évènementiel

B.1 Spécification abstraite

MODEL
$AbstractSpec$
SETS
$VAL; PLACE = \{PA1, PA2\}; PROCESS$
VARIABLES
x, pp
CONSTANTS
f
PROPERTIES
$f \in VAL \rightarrow VAL \wedge$
$\forall v.(v \in VAL)$
INVARIANT
$x \in VAL \wedge$
$pp \in PROCESS \rightarrow PLACE$
INITIALISATION
$x :\in VAL \parallel$
$pp :\in PROCESS \rightarrow PLACE$
EVENTS
$rely =$ **ANY** v **WHERE**
$v \in VAL$
THEN
$x := v$
END ;

$assign =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp(p) = PA1$
THEN
$x := f(x) \parallel$
$pp(p) := PA2$
END ;
$p1p2 = skip;$
$p2p3 = skip;$
$p3p4b = skip;$
$p4bp1 = skip$
END

B.2 Spécification concrète

REFINEMENT
$Spec2$
REFINES
$AbstractSpec$
/*
Those theorems help a lot in proofs :
$\forall(f,g,x,y,s,q,t).(f \in s \rightarrow q \wedge g \in q \rightarrow t \wedge x \in s \wedge y \in q \wedge g(f(x)) = g(y) \Rightarrow (f;g) = (f \lhd \{x \mapsto y\};g)))$
$\forall(f,g,x,y,s,q,t).(f \in s \rightarrow q \wedge g \in q \rightarrow t \wedge x \in s \wedge y \in t \wedge g(f(x)) = y \Rightarrow (f;g) = (f;g) \lhd \{x \mapsto y\}))$
*/

SETS
$PLACE2 = \{P1, P2, P3, P4A, P4B, P5\}$
CONSTANTS
map
PROPERTIES
$map \in PLACE2 \rightarrow PLACE \wedge$
$map(P1) = PA1 \wedge map(P2) = PA1 \wedge$
$map(P3) = PA1 \wedge map(P4B) = PA1 \wedge$
$map(P4A) = PA2 \wedge map(P5) = PA2$
VARIABLES
$x, pp2, r, v, c$
INVARIANT
$x \in VAL \wedge (pp2 \in PROCESS \rightarrow PLACE2) \wedge pp = (pp2; map) \wedge$
$r \in PROCESS \rightarrow VAL \wedge v \in PROCESS \rightarrow VAL \wedge c \in PROCESS \rightarrow VAL \wedge$
$(\forall p.(p \in PROCESS \Rightarrow ($
$(pp2(p) = P4A \Rightarrow c(p) = r(p)) \wedge$
$(pp2(p) = P4B \Rightarrow c(p) \neq r(p)) \wedge$
$(pp2(p) = P3 \Rightarrow v(p) = f(r(p)))$
$)))$
INITIALISATION
$x :\in VAL \parallel$
$pp2 :\in PROCESS \rightarrow \{P1\} \parallel$
$r :\in PROCESS \rightarrow VAL \parallel$
$v :\in PROCESS \rightarrow VAL \parallel$
$c :\in PROCESS \rightarrow VAL$

EVENTS
$rely =$ **ANY** v **WHERE**
$v \in VAL$
THEN
$x := v$
END ;

$p1p2 =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp2(p) = P1$
THEN
$r(p) := x \parallel$
$pp2(p) := P2$
END ;

$p2p3 =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp2(p) = P2$
THEN
$v(p) := f(r(p)) \parallel$
$pp2(p) := P3$
END ;

$/ * p3p4a\, c = CAS1(x, r, v) * /$
$assign =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp2(p) = P3 \wedge$
$r(p) = x$
THEN
$x := v(p) \parallel$
$pp2(p) := P4A \parallel$
$c(p) := x$
END ;

$/ * c = CAS1(x, r, v) * /$
$p3p4b =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp2(p) = P3 \wedge$
$x \neq r(p)$
THEN
$c(p) := x$
END ;

$p4bp1 =$ **ANY** p **WHERE**
$p \in PROCESS \wedge$
$pp2(p) = P4B$
THEN
$pp2(p) := P1$
END
END

Annexe C

Affectation en $^+$CAL/TLA$^+$

C.1 Spécification abstraite

```
–algorithm Assign variables x ;

process ProcAssign ∈ 1 .. M begin
   OP : x := f(x)
 end process

process ProcRely ∈ M .. N begin
   OPR : with val ∈ Nat do
        x := val
      end with ;
 end process

end algorithm
```

MODULE *Assign*

```
EXTENDS Naturals, TLC, Sequences
CONSTANT M, N, SomeValues

f ≜ [x ∈ Nat ↦ x + 4]

ASSUME
 f ∈ [Nat → Nat]
 **** BEGIN TRANSLATION **
CONSTANT defaultInitValue
VARIABLES x, pc

vars ≜ ⟨x, pc⟩

ProcSet ≜ (1 .. M) ∪ (M .. N)

Init ≜  Global variables
        ∧ x = defaultInitValue
        ∧ pc = [self ∈ ProcSet ↦ CASE self ∈ 1 .. M  → "OP"
                                    □self    ∈ M .. N → "OPR"]

OP(self) ≜  ∧ pc[self] = "OP"
            ∧ x' = f[x]
            ∧ pc' = [pc EXCEPT ![self] = "Done"]

ProcAssign(self) ≜ OP(self)

OPR(self) ≜  ∧ pc[self] = "OPR"
```

$\land \exists\, val \in Nat :$
$\quad x' = val$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}]$

$ProcRely(self) \triangleq OPR(self)$

$Next \triangleq (\exists\, self \in 1 \,..\, M : ProcAssign(self))$
$\quad \lor (\exists\, self \in M \,..\, N : ProcRely(self))$
$\quad \lor$ Disjunct to prevent deadlock on termination
$\quad ((\forall\, self \in ProcSet : pc[self] = \text{“Done”}) \land \text{UNCHANGED } vars)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] = \text{“Done”})$

**** END *TRANSLATION* **

$Invariant \triangleq x \in Nat$

C.2 Spécification concrète

–algorithm *casAssign* variables x ;

```
macro BCAS1(x, o, n, r) begin
  if (x = o) then
    x := n ;
    r := TRUE ;
  else r := FALSE ;
  end if
end macro

procedure CasAssign() variables b, v, fv ; begin
A :     b := FALSE ; A1 :    while ¬b do
B :       v := x ;
C :       fv := f(v) ;
D :       BCAS1(x, v, fv, b) ; end while
end procedure

process ProcAssign ∈ 1 .. M begin
    OP : call CasAssign() ;
  end process

process ProcRely ∈ M .. N begin
    OPR : with val ∈ Nat do
        x := val
      end with ;
  end process

end algorithm
```

MODULE *CasAssign*

EXTENDS *Naturals*, *TLC*, *Sequences*
CONSTANT M, N

**** BEGIN *TRANSLATION* **
CONSTANT *defaultInitValue*
VARIABLES x, pc, $stack$, b, v, fv

$vars \triangleq \langle x, pc, stack, b, v, fv \rangle$

$ProcSet \triangleq (1 \,..\, M) \cup (M \,..\, N)$

$Init \triangleq$ Global variables
$\land\, x = defaultInitValue$
Procedure $CasAssign$
$\land\, b = [self \in ProcSet \mapsto defaultInitValue]$
$\land\, v = [self \in ProcSet \mapsto defaultInitValue]$
$\land\, fv = [self \in ProcSet \mapsto defaultInitValue]$
$\land\, stack = [self \in ProcSet \mapsto \langle\rangle]$
$\land\, pc = [self \in ProcSet \mapsto \text{CASE } self \in 1 \,..\, M \to \text{“OP”}$
$\quad \Box self \in M \,..\, N \to \text{“OPR”}]$

$A(self) \triangleq \land\, pc[self] = \text{“A”}$
$\land\, b' = [b \text{ EXCEPT } ![self] = \text{FALSE}]$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“A1”}]$
$\land \text{ UNCHANGED } \langle x,\, stack,\, v,\, fv\rangle$

$A1(self) \triangleq \land\, pc[self] = \text{“A1”}$
$\land \text{ IF } \neg b[self]$
$\text{THEN } \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“B”}]$
$\text{ELSE } \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“Error”}]$
$\land \text{ UNCHANGED } \langle x,\, stack,\, b,\, v,\, fv\rangle$

$B(self) \triangleq \land\, pc[self] = \text{“B”}$
$\land\, v' = [v \text{ EXCEPT } ![self] = x]$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“C”}]$
$\land \text{ UNCHANGED } \langle x,\, stack,\, b,\, fv\rangle$

$C(self) \triangleq \land\, pc[self] = \text{“C”}$
$\land\, fv' = [fv \text{ EXCEPT } ![self] = f[v[self]]]$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“D”}]$
$\land \text{ UNCHANGED } \langle x,\, stack,\, b,\, v\rangle$

$D(self) \triangleq \land\, pc[self] = \text{“D”}$
$\land \text{ IF } (x = v[self])$
$\text{THEN } \land\, x' = fv[self]$
$\land\, b' = [b \text{ EXCEPT } ![self] = \text{TRUE}]$
$\text{ELSE } \land\, b' = [b \text{ EXCEPT } ![self] = \text{FALSE}]$
$\land \text{ UNCHANGED } x$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“A1”}]$
$\land \text{ UNCHANGED } \langle stack,\, v,\, fv\rangle$

$CasAssign(self) \triangleq A(self) \lor A1(self) \lor B(self) \lor C(self) \lor D(self)$

$OP(self) \triangleq \land\, pc[self] = \text{“OP”}$
$\land\, stack' = [stack \text{ EXCEPT } ![self] = \langle [procedure \mapsto \text{“CasAssign”},$
$pc \mapsto \text{“Done”},$
$b \mapsto b[self],$
$v \mapsto v[self],$
$fv \mapsto fv[self]]\rangle$
$\circ\, stack[self]]$
$\land\, b' = [b \text{ EXCEPT } ![self] = defaultInitValue]$
$\land\, v' = [v \text{ EXCEPT } ![self] = defaultInitValue]$
$\land\, fv' = [fv \text{ EXCEPT } ![self] = defaultInitValue]$
$\land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“A”}]$
$\land \text{ UNCHANGED } x$

$ProcAssign(self) \triangleq OP(self)$

$OPR(self) \triangleq \land pc[self] = \text{“OPR”}$
$\land \exists\, val \in Nat :$
$\quad x' = val$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}]$
$\land \text{UNCHANGED } \langle stack,\ b,\ v,\ fv \rangle$

$ProcRely(self) \triangleq OPR(self)$

$Next \triangleq (\exists\, self \in ProcSet : CasAssign(self))$
$\quad \lor (\exists\, self \in 1 \,..\, M : ProcAssign(self))$
$\quad \lor (\exists\, self \in M \,..\, N : ProcRely(self))$
$\quad \lor$ Disjunct to prevent deadlock on termination
$((\forall\, self \in ProcSet : pc[self] = \text{“Done”}) \land \text{UNCHANGED } vars)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] = \text{“Done”})$

**** END *TRANSLATION* **

$Invariant \triangleq x \in Nat$

C.3 Vérification du raffinement

MODULE *CasAssignRefine*

The *CasAssign* model is a refinement of the *Assign* one.

EXTENDS *CasAssign*

$Atomic \triangleq \text{INSTANCE } Assign$
$AtomicSpec \triangleq Atomic!Spec$

THEOREM
$\quad Spec \implies AtomicSpec$

Annexe D

Théories Isabelle

D.1 ListUtil

theory *ListUtil* **imports** *Main List* **begin**

D.1.1 List-all utility

lemma *list_all_conj:*
```
  "( (list_all P1 L) ∧ (list_all P2 L) ) = (list_all (λ x. (P1 x) ∧ (P2 x)) L)"
```
by *(auto simp add: list_all_iff)*

lemma *list_all_disj:*
```
  "( (list_all P1 L) ∨ (list_all P2 L) ) ⟹ (list_all (λ x. (P1 x) ∨ (P2 x)) L)"
```
by *(auto simp add: list_all_iff)*

D.1.2 List-all2 utility

lemma *list_all2_conj:*
```
  "( (list_all2 P1 L1 L2) ∧ (list_all2 P2 L1 L2) ) =
     (list_all2 (λ x y. (P1 x y) ∧ (P2 x y)) L1 L2)"
```
by *(auto simp add: list_all2_def list_all_iff)*

lemma *list_all2_map_list_all:*
```
  "(list_all2 (P::'a ⇒ 'b ⇒ bool) L1 (map (f::'a ⇒ 'b) L1)) =
   (list_all (λ x. P x (f x)) L1)"
```
by *(auto simp add: list_all_iff list_all2_conv_all_nth all_set_conv_all_nth)*

D.1.3 List-all3

consts
```
  list_all3 :: "('a => 'b ⇒ 'c => bool) => 'a list => 'b list => 'c list => bool"
```

defs
```
  list_all3_def:
  "list_all3 P xs ys zs ⇔
  length xs = length ys ∧ length ys = length zs ∧
  (∀ (x, (y, z))  ∈  set (zip xs (zip ys zs)). P x y z)"
```

lemma *list_all3_lengthD [intro?]:*
```
  "list_all3 P xs ys zs ⟹ length xs = length ys ∧ length ys = length zs"
```
  by *(simp add: list_all3_def)*

lemma *list_all3_Nil [iff,code]: "list_all3 P [] ys zs = ((ys = []) ∧ (zs = []))"*

```
  by (auto simp add: list_all3_def)

lemma list_all3_Nil2[iff]: "list_all3 P xs [] zs = (xs = [] ∧ zs=[])"
  by (auto simp add: list_all3_def)

lemma list_all3_Nil3[iff]: "list_all3 P xs ys [] = (xs = [] ∧ ys=[])"
  by (auto simp add: list_all3_def)

lemma list_all3_Cons [iff,code]:
  "list_all3 P (x # xs) (y # ys) (z # zs)= (P x y z ∧  list_all3 P xs ys zs)"
  by (auto simp add: list_all3_def)




lemma list_all3_rev [iff]:
  "list_all3 P (rev xs) (rev ys) (rev zs)= list_all3 P xs ys zs"
  by (simp add: list_all3_def zip_rev cong: conj_cong)

lemma list_all3_rev1:
  "list_all3 P (rev xs) ys zs = list_all3 P xs (rev ys) (rev zs)"
  by (subst list_all3_rev [symmetric]) simp

lemma list_all3_rev2:
  "list_all3 P xs (rev ys) zs = list_all3 P (rev xs) ys (rev zs)"
  by (subst list_all3_rev [symmetric]) simp

lemma list_all3_rev3:
  "list_all3 P xs ys (rev zs) = list_all3 P (rev xs) (rev ys) zs"
  by (subst list_all3_rev [symmetric]) simp



lemma list_all3_conv_all_nth:
"list_all3 P xs ys zs =
(length xs = length ys ∧length ys = length zs ∧  (∀ i < length xs. P (xs!i) (ys!i) (zs!i)))"
  by (force simp add: list_all3_def set_zip)


lemma list_all3_all_nthI [intro?]:
  "⟦ length a = length b; length b = length c ⟧ ⟹ (!!n. n < length a ⟹ P (a!n) (b!n) (c!n)) ⟹
list_all3 P a b c"
  by (simp add: list_all3_conv_all_nth)



lemma list_all3_nthD:
  "⟦ list_all3 P xs ys zs ; p < size xs ⟧ ⟹ P (xs!p) (ys!p) (zs!p)"
  by (clarsimp simp add: list_all3_conv_all_nth)



lemma list_all3_conj:
  "( (list_all3 P1 L1 L2 L3) ∧ (list_all3 P2 L1 L2 L3) ) =
   (list_all3 (λ x y z. (P1 x y z) ∧ (P2 x y z)) L1 L2 L3)"
by (auto simp add: list_all3_def list_all_iff)
```

D.1.4 list-all4

constdefs
```
  list_all4 :: "('a => 'b ⇒ 'c => 'd ⇒ bool) => 'a list => 'b list => 'c list ⇒ 'd list => bool"
  "list_all4 P ws xs ys zs ⇔
  length ws = length xs ∧ length xs = length ys ∧ length ys = length zs ∧
  (∀ (w, (x, (y, z))) ∈ set (zip ws (zip xs (zip ys zs))). P w x y z)"
```

lemma *list_all4_lengthD [intro?]:*
```
  "list_all4 P ws xs ys zs ==> length ws = length xs ∧ length xs = length ys ∧ length ys = length
zs"
```
  by *(simp add: list_all4_def)*

lemma *list_all4_Nil [iff,code]: "list_all4 P [] xs ys zs = ((xs = []) ∧ (ys = []) ∧ (zs = []))"*
  by *(auto simp add: list_all4_def)*

lemma *list_all4_Nil2[iff]: "list_all4 P ws [] ys zs = (ws = [] ∧ ys = [] ∧ zs=[])"*
  by *(auto simp add: list_all4_def)*

lemma *list_all4_Nil3[iff]: "list_all4 P ws xs [] zs = (ws = [] ∧ xs = [] ∧ zs=[])"*
  by *(auto simp add: list_all4_def)*

lemma *list_all4_Nil4[iff]: "list_all4 P ws xs ys [] = (ws = [] ∧ xs = [] ∧ ys=[])"*
  by *(auto simp add: list_all4_def)*

lemma *list_all4_Cons [iff,code]:*
```
  "list_all4 P (w # ws) (x # xs) (y # ys) (z # zs)= (P w x y z ∧ list_all4 P ws xs ys zs)"
```
  by *(auto simp add: list_all4_def)*

lemma *list_all4_conv_all_nth:*
```
"list_all4 P ws xs ys zs =
(length ws = length xs ∧ length xs = length ys ∧length ys = length zs ∧ (∀ i < length xs. P (ws!i)
(xs!i) (ys!i) (zs!i)))"
```
  by *(force simp add: list_all4_def set_zip)*

lemma *list_all4_all_nthI [intro?]:*
```
  "⟦ length a = length b; length b = length c; length c = length d ⟧ ⟹ (!!n. n < length a ⟹ P
(a!n) (b!n) (c!n) (d!n)) ⟹ list_all4 P a b c d"
```
  by *(simp add: list_all4_conv_all_nth)*

lemma *list_all4_nthD:*
```
  "⟦ list_all4 P ws xs ys zs ; p < size ws ⟧ ⟹ P (ws!p) (xs!p) (ys!p) (zs!p)"
```
  by *(clarsimp simp add: list_all4_conv_all_nth)*

D.1.5 list-all5

constdefs
```
  list_all5 :: "('a => 'b ⇒ 'c => 'd ⇒ 'e ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ 'c list ⇒ 'd list
⇒ 'e list ⇒ bool"
  "list_all5 P vs ws xs ys zs ⇔
  length vs = length ws ∧ length ws = length xs ∧ length xs = length ys ∧ length ys = length zs
∧
  (∀ (v, (w, (x, (y, z)))) ∈ set (zip vs (zip ws (zip xs (zip ys zs))))). P v w x y z)"
```

lemma *list_all5_lengthD [intro?]:*
```
  "list_all5 P vs ws xs ys zs ⟹ length vs = length ws ∧ length ws = length xs ∧ length xs = length
ys ∧ length ys = length zs"
```
  by *(simp add: list_all5_def)*

lemma *list_all5_Nil [iff,code]: "list_all5 P [] ws xs ys zs = (ws = [] ∧ (xs = []) ∧ (ys = []) ∧ (zs = []))"*
by *(auto simp add: list_all5_def)*

lemma *list_all5_Nil2[iff]: "list_all5 P vs [] xs ys zs = (vs =[] ∧ xs = [] ∧ ys = [] ∧ zs=[])"*
by *(auto simp add: list_all5_def)*

lemma *list_all5_Nil3[iff]: "list_all5 P vs ws [] ys zs = (vs = [] ∧ ws=[] ∧ ys = [] ∧ zs=[])"*
by *(auto simp add: list_all5_def)*

lemma *list_all5_Nil4[iff]: "list_all5 P vs ws xs [] zs = (vs = [] ∧ ws = [] ∧ xs = [] ∧ zs=[])"*
by *(auto simp add: list_all5_def)*

lemma *list_all5_Nil4[iff]: "list_all5 P vs ws xs ys [] = (vs = [] ∧ ws = [] ∧ xs = [] ∧ ys=[])"*
by *(auto simp add: list_all5_def)*

lemma *list_all5_Cons [iff,code]:*
"list_all5 P (v # vs) (w # ws) (x # xs) (y # ys) (z # zs)= (P v w x y z ∧ list_all5 P vs ws xs ys zs)"
by *(auto simp add: list_all5_def)*

D.1.6 list-all6

constdefs
list_all6 :: "('a => 'b ⇒ 'c => 'd ⇒ 'e ⇒ 'f ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ 'c list ⇒ 'd list ⇒ 'e list ⇒ 'f list ⇒ bool"
"list_all6 P us vs ws xs ys zs ⇔
length us = length vs ∧ length vs = length ws ∧ length ws = length xs ∧ length xs = length ys ∧ length ys = length zs ∧
(∀ (u, (v, (w, (x, (y, z))))) ∈ set (zip us (zip vs (zip ws (zip xs (zip ys zs)))))). P u v w x y z)"

lemma *list_all6_lengthD [intro?]:*
"list_all6 P us vs ws xs ys zs ⟹ length us = length vs ∧ length vs = length ws ∧ length ws = length xs ∧ length xs = length ys ∧ length ys = length zs"
by *(simp add: list_all6_def)*

lemma *list_all6_Cons [iff,code]:*
"list_all6 P (u # us) (v # vs) (w # ws) (x # xs) (y # ys) (z # zs)= (P u v w x y z ∧ list_all6 P us vs ws xs ys zs)"
by *(auto simp add: list_all6_def)*

D.1.7 Other

constdefs
inside :: "'a list ⇒ 'a ⇒ nat ⇒ bool"
"inside ll oo ii == (ll!ii) = oo ∧ ii < length ll"

first_index :: "'a list ⇒ 'a ⇒ nat"
"first_index ll oo == ϵ ii. ∀ jj. (((ll!jj) = oo) ⟶ (ii <= jj))"

constdefs
index_of :: "'a list ⇒ 'a ⇒ nat"
"index_of ll oo == ϵ ii. inside ll oo ii"

lemma *index_of_is_in_length:*
assumes *xIn : "x ∈ set ll"*
shows *"(index_of ll x) < length ll"*

```
proof -
  from xIn have "(∃ i. inside ll x i)" (is "∃ i. ?P i")
    by (auto simp add: inside_def in_set_conv_nth)
  then have "?P (ϵ i. ?P i)"
    by (clarsimp simp: someI_ex)
  thus ?thesis
    by (auto simp add:index_of_def inside_def)
qed

lemma index_of_is_item:
  assumes xIn : "x ∈ set ll"
  shows "ll!(index_of ll x) = x"
proof -
  from xIn have "(∃ i. inside ll x i)" (is "∃ i. ?P i")
    by (auto simp add: inside_def in_set_conv_nth)
  then have "?P (ϵ i. ?P i)"
    by (clarsimp simp: someI_ex)
  thus ?thesis
    by (auto simp add:index_of_def inside_def)
qed

end
```

D.2 Formalisation

```
theory Formalisation imports Main ListUtil begin
```

D.2.1 Definition

A process' state is the program counter and the local environement.

```
record ('place,'lenv)ProcessState =
  lenv :: "'lenv"
  pc :: "'place"
```

A system state is composed of the global environement and all the processes' states. Note that the pid of a process is its index in the ProcessState list.

```
record ('place,'genv,'lenv)SystemState =
  global_env :: "'genv"
  prs_states :: "('place, 'lenv)ProcessState list"
```

To make things more clear, we introduce PartialState wich is a view of the real states. It extracts global environement, a process' state, a pid and the algorithm index the process runs. See the function extract-pst on how to construct partial state from a system state.

```
record ('place,'genv,'lenv)PartialState =
  genv :: "'genv"
  lenv :: "'lenv"
  pc :: "'place"
  pid :: "nat"
  algo_index :: "nat"

types ValidPidFunctionType =
  "nat ⇒ bool"
```

An algorithm is characterized by an initial environement, the transition, the rely, and the local invariant.

```
record ('place,'genv,'lenv)Algorithm =
```

```
  init :: "('place,'genv,'lenv)PartialState ⇒ bool"
  trans :: "('place,'genv,'lenv)PartialState ⇒ ('place,'genv,'lenv)PartialState ⇒ bool"
  rely :: "('place,'genv,'lenv)PartialState ⇒ 'genv ⇒ bool"
  linv :: "ValidPidFunctionType ⇒ ('place,'genv,'lenv)PartialState ⇒ bool"
```

A specification is just a sequence of algorithms.

```
record ('place,'genv,'lenv)Specification =
  algos :: "('place,'genv,'lenv)Algorithm list"
```

A Process has two field. A pid which is its position in the system's process list. And the index of the algorithm it is running.

```
record Process =

  algo :: nat
```

A system runs a specification and is composed of a list of processes.

```
record ('place,'genv,'lenv)System =
  prs :: "Process list"
  spec :: "('place,'genv,'lenv)Specification"
```

A gluing invariant is attached to a specific algorithm and links two partial state.

```
types ('genvc,'placec,'lenvc,'genva,'placea,'lenva)proc_gluing =
"(('placec,'genvc,'lenvc)PartialState ⇒ ('placea,'genva,'lenva)PartialState ⇒ bool)"
```

Each concrete transition does an abstract one. The abstractPid function of the following type will give the abstract process' pid.

```
types ('placec,'genvc,'lenvc)AbstractPidType =
  "('placec,'genvc,'lenvc)Algorithm ⇒ ('placec,'genvc,'lenvc)PartialState ⇒ ('placec,'genvc,'lenvc)PartialState
⇒ nat"
```

A refinement explain how a concrete specification refines an abstract one. the gluing invariant is the data refinement relation. Also a technical detail is that an algorithm can do an abstract step (in refinement). Usually it is itself, but in lock-free algorithm it can be another one, depending an the concrete transition.

```
record ('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement =
  concrete_spec :: "('placec,'genvc,'lenvc)Specification"
  abstract_spec :: "('placea,'genva,'lenva)Specification"
  gluing :: "(('genvc, 'placec, 'lenvc, 'genva,'placea, 'lenva)proc_gluing) list"
  abstractPid :: "('placec,'genvc,'lenvc)AbstractPidType"
```

D.2.2 Stuttering definition

```
constdefs
  is_proc_stuttering :: "('place,'lenv)ProcessState ⇒ ('place,'lenv)ProcessState ⇒ bool"
  "is_proc_stuttering st1 st2 == st1 = st2"


constdefs
  is_stuttering :: "('place,'genv,'lenv)SystemState ⇒ ('place,'genv,'lenv)SystemState ⇒ bool"
  "is_stuttering st1 st2 ==
  st1 = st2"
```

Two partial states are linked by local stuttering if they keep same process state, pid and algo-index.

```
constdefs
  proc_stuttering :: "('placea,'genva,'lenva)PartialState ⇒ ('placea,'genva,'lenva)PartialState ⇒
bool"
  "proc_stuttering pst1 pst2 ==
```

```
  (lenv pst1) = (lenv pst2) ∧
  (pc pst1) = (pc pst2) ∧
  (pid pst1) = (pid pst2) ∧
  (algo_index pst1) = (algo_index pst2)
  "

lemma proc_stuttering_is_reflexive:
  shows "proc_stuttering pst1 pst1"
  by (auto simp add: proc_stuttering_def)

constdefs
  stuttering :: "('placea,'genva,'lenva)PartialState ⇒ ('placea,'genva,'lenva)PartialState ⇒ bool"
  "stuttering pst1 pst2 ==
  pst1 = pst2
  "
```

D.2.3 Utility functions

Return algorithm index used by process n.

```
constdefs
  get_algo_index :: "('place,'genv,'lenv)System ⇒ nat ⇒ nat"
  "get_algo_index sys n ==
  (algo ((prs sys)!n))
  "
```

return the algorithm used by the process n

```
constdefs
  get_algo :: "('place,'genv,'lenv)System ⇒ nat ⇒ ('place,'genv,'lenv)Algorithm"
  "get_algo sys n ==
  (algos (spec sys))!(get_algo_index sys n)
  "
```

Construct the partial state of the given System and System state.

```
constdefs
  extract_pst :: "('place,'genv,'lenv)System ⇒ ('place,'genv,'lenv)SystemState ⇒ nat ⇒ ('place,'genv,'len
  "extract_pst sys st n ==
  (|
  genv = (global_env st),
  lenv = (ProcessState.lenv ((prs_states st)!n)),
  pc = (ProcessState.pc ((prs_states st)!n)),
  pid = n,
  algo_index = (get_algo_index sys n) |)
  "

lemma pid_is_nth:
  shows "(pid (extract_pst sys st n)) = n"
  by (auto simp add: extract_pst_def)
```

Given a partial state, update the global env

```
constdefs
  update_genv :: "('place,'genv,'lenv)PartialState ⇒ 'genv ⇒ ('place,'genv,'lenv)PartialState"
  "update_genv pst genv_after == pst(| genv := genv_after |)"
```

Given a system state and a new partial state, create the new system state.

```
constdefs
  update_stc :: "('place,'genv,'lenv)SystemState ⇒ ('place,'genv,'lenv)PartialState ⇒ ('place,'genv,'lenv)
  "update_stc stc pstc ==
```

```
  (| global_env = (genv pstc),
  prs_states = list_update (prs_states stc) (pid pstc) (ProcessState.make (lenv pstc) (pc pstc))  |)"

lemma extract_update_stc:
  assumes "(pid pstc) < length (prs_states stc)"
  and "(algo_index pstc) = algo (prs sys ! pid pstc)"
  shows "pstc = extract_pst sys (update_stc stc pstc) (pid pstc)"
  by (auto! simp add: extract_pst_def update_stc_def ProcessState.make_def get_algo_index_def nth_list_update)

lemma extract_update_stc_genv:
  assumes "i ≠ (pid pstc)"
  shows "extract_pst sys (update_stc stc pstc) i = (update_genv (extract_pst sys stc i) (genv pstc))"
  by (auto! simp add: extract_pst_def update_stc_def update_genv_def)

lemma update_stc_keep_valid:
  shows "length (prs_states stc) = length (prs_states (update_stc stc pstc))"
  by (auto simp add: update_stc_def)
```

Two partial states are linked by a step if they keep same pid and algo-index, but change the program counter.

```
constdefs
  local_step :: "('placea,'genva,'lenva)PartialState ⇒ ('placea,'genva,'lenva)PartialState ⇒ bool"
  "local_step pst1 pst2 ==
  (pc pst1) ≠ (pc pst2) ∧
  (pid pst1) = (pid pst2) ∧
  (algo_index pst1) = (algo_index pst2)
  "
```

Two system states are valid step if one processus has done a real step and the others are proc-stuttering.

```
constdefs
  is_valid_step :: "('place,'genv,'lenv)System ⇒ ('place,'genv,'lenv)SystemState ⇒ ('place,'genv,'lenv)SystemSta
⇒ bool"
  "is_valid_step sys st1 st2 ==
  length (prs_states st1) = length (prs_states st2) ∧
  (
    ∃j < length (prs sys).
    ((trans (get_algo sys j)) (extract_pst sys st1 j) (extract_pst sys st2 j)) ∧
    (local_step (extract_pst sys st1 j) (extract_pst sys st2 j)) ∧
    (∀ i < length (prs sys). (
    ((i ≠ j) ⟶
      (is_proc_stuttering ((prs_states st1)!i) ((prs_states st2)!i))
    )))
  )
  "
```

Two system states are valid transition if it is a real step or a stuttering.

```
constdefs
  is_valid_trans :: "('place,'genv,'lenv)System ⇒ ('place,'genv,'lenv)SystemState ⇒ ('place,'genv,'lenv)SystemSt
⇒ bool"
  "is_valid_trans sys st1 st2 ==
  (is_stuttering st1 st2) ∨ (is_valid_step sys st1 st2)"
```

local invariant holds if the local invariant of all processes holds.

```
constdefs
  linv_hold :: "ValidPidFunctionType ⇒ ('place,'genv,'lenv)System ⇒ ('place,'genv,'lenv)SystemState
⇒ bool"
  "linv_hold valid_pid sys st1 ==
  (∀ i.
```

```
  i < length (prs sys)
  ⟶
  ((linv (get_algo sys i)) valid_pid (extract_pst sys st1 i))
  )
  "
```

A system state is valid towards a system if local inv holds and they have equals number of process, resp. process states

constdefs

```
  valid_state :: "ValidPidFunctionType ⇒ ('place,'genv,'lenv)System ⇒ ('place,'genv,'lenv)SystemState
⇒ bool"
  "valid_state valid_pid sys st1 ==
  length (prs sys) = length (prs_states st1) ∧
  (linv_hold valid_pid sys st1)"
```

lemma *update_stc_unchanged:*

```
  assumes safe_i: "i < length (prs_states stc)"
  shows "stc = update_stc stc (extract_pst sys stc i)"
proof -
  from safe_i
  have "ProcessState.make (ProcessState.lenv (prs_states stc ! i)) (ProcessState.pc (prs_states stc
! i))
    = (prs_states stc)!i"
    by (auto simp add: ProcessState.make_def)
  with safe_i have "prs_states stc
              [i := ProcessState.make (ProcessState.lenv (prs_states stc ! i)) (ProcessState.pc (prs_states
stc ! i))]
    = prs_states stc"
    by (auto simp add: nth_list_update_eq)
  thus ?thesis
    by (auto simp add: update_stc_def extract_pst_def)
qed
```

Just for sake of readability we have introduced PartialState, but we need to ensure that duplicated information is coherent. The global environement must be the same. If pids are same, ie we extract same process, then we must have same local extract.

constdefs

```
  coherent_state :: "('place,'genv,'lenv)PartialState ⇒ ('place,'genv,'lenv)PartialState ⇒ bool"
  "coherent_state pst1 pst2 ==
  (genv pst1) = (genv pst2) ∧
  ((pid pst1) = (pid pst2) ⟶ (proc_stuttering pst1 pst2))"
```

lemma *coherent_state_is_reflexive:*

```
  shows "coherent_state pst1 pst1"
  by (auto simp add: coherent_state_def proc_stuttering_def)
```

lemma *extract_coherent_state:*

```
  shows "coherent_state (extract_pst sys st pid1) (extract_pst sys st pid2)"
  by (auto simp add: coherent_state_def extract_pst_def proc_stuttering_def)
```

lemma *same_pid_imply_same_state:*

```
  assumes "coherent_state pst1 pst2"
  and "pid pst1 = pid pst2"
  shows "pst1 = pst2"
  by (auto! simp add: coherent_state_def proc_stuttering_def)
```

Ensure that processes that refines, use the same algorithm.

constdefs

```
  coherent_sys :: "('placec,'genvc,'lenvc)System ⇒ ('placea,'genva,'lenva)System ⇒ bool"
  "coherent_sys sysc sysa ==
  (list_all2
    (λ prsc prsa. (algo prsc) = (algo prsa))
    (prs sysc)
    (prs sysa))
  "
```

constdefs

```
  valid_gluing :: "
  ('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement ⇒
  ('placec,'genvc,'lenvc)PartialState ⇒
  ('placea,'genva,'lenva)PartialState ⇒
  bool"
  "valid_gluing ref pstc psta ==
  (((gluing ref)!(algo_index pstc)) pstc psta)
  "
```

constdefs

```
  is_valid_gluing :: "
  ('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement ⇒
  ('placec,'genvc,'lenvc)System ⇒
  ('placea,'genva,'lenva)System ⇒
  ('placec,'genvc,'lenvc)SystemState ⇒
  ('placea,'genva,'lenva)SystemState ⇒
  bool"
  "is_valid_gluing ref sysc sysa stc1 sta1 ==
  (length (prs_states sta1)) = (length (prs_states stc1)) ∧
  (∀ i < (length (prs_states sta1)).
  (valid_gluing ref (extract_pst sysc stc1 i) (extract_pst sysa sta1 i)))
  "
```

D.2.4 Valid algorithm

For sake of readability we need transition (trans field) to code stuttering too.

constdefs

```
  stuttering_is_authorized_in_trans :: "( 'place, 'genv, 'lenv)Algorithm ⇒ bool"
  "stuttering_is_authorized_in_trans algoc ==
  ∀ pst valid_pid.
  ((linv algoc) valid_pid pst)
  ⟶
  ((trans algoc) pst pst)"
```

Part of this proof obligation is usefull only in theory, it could be valid by construction. The main point is that the transition validate the local invariant.

constdefs

```
  stut_or_one_step :: "( 'place, 'genv, 'lenv)Algorithm ⇒ bool"
  "stut_or_one_step algoc ==
  ∀ pst1 pst1_after valid_pid.(
  (((linv algoc) valid_pid pst1) ∧
   ((trans algoc) pst1 pst1_after)
   ⟶ ((linv algoc) valid_pid pst1_after)) ∧
  (((trans algoc) pst1 pst1_after)
  ⟶
    (
      (stuttering pst1 pst1_after) ∨
      (local_step pst1 pst1_after)
    )
```

```
  ))
  "
```

constdefs

```
  valid_linv_re_rely :: "( 'place, 'genv, 'lenv)Algorithm ⇒ bool"
  "valid_linv_re_rely algo1 ==
  ∀ valid_pid pst1. (∀ genv_after. (linv algo1) valid_pid pst1 ∧ rely (algo1) pst1 genv_after ⟶
(linv algo1) valid_pid (update_genv pst1 genv_after))
  "
```

constdefs

```
  valid_algo :: "( 'place, 'genv, 'lenv)Algorithm ⇒ bool"
  "valid_algo algo1 ==
  (stut_or_one_step algo1) ∧
  (stuttering_is_authorized_in_trans algo1) ∧
  (valid_linv_re_rely algo1)
  "
```

constdefs

```
  valid_external_rely :: "( 'place, 'genv, 'lenv)Algorithm ⇒ ( 'place, 'genv, 'lenv)Algorithm ⇒
bool"
  "valid_external_rely algo1 algo2 ==
  ∀ valid_pid pst1 pst1_after. ∀ pst2. linv algo1 valid_pid pst1 ∧ trans algo1 pst1 pst1_after ⟶
rely algo2 pst2 (genv pst1_after)
  "
```

constdefs

```
  valid_external_rely_spec :: "('place,'genv,'lenv)Specification ⇒ ( 'place, 'genv, 'lenv)Algorithm
⇒ bool"
  "valid_external_rely_spec specif algo1 ==
  list_all (valid_external_rely algo1) (algos specif)"
```

This one can be done by construction too.

constdefs

```
  gluing_must_ensure_same_pid_and_algo_index :: "('genvc,'placec,'lenvc,'genva,'placea,'lenva)proc_gluing
⇒ bool"
  "gluing_must_ensure_same_pid_and_algo_index gluing1 ==
  ∀ pstc psta. (gluing1 pstc psta) ⟶ (pid pstc) = (pid psta) ∧ (algo_index pstc) = (algo_index
psta)"
```

The idea is that we must be able to construct a valid abstract trace given a concrete one. The construction will be done step by step. If we are in a situation where pst1c refines pst1a, pst2c refines pst2a. We take a concrete step (pst2c, pst2c-after). Then there must exist two abstract states pst1a-after and pst2a-after that are valid abstract transitions. But they need also to respect the gluing invariant. This must be true for all processes, so for all algorithms. To respect interleaving semantic, there must be only one abstract step. The process doing the step is given by the abstract-pid function.

constdefs

```
  external_step :: "ValidPidFunctionType ⇒ ('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement

  ⇒ nat ⇒ nat
  ⇒ ('placec,'genvc,'lenvc)PartialState ⇒ ('placec,'genvc,'lenvc)PartialState ⇒ ('placec,'genvc,'lenvc)Pa

  ⇒ ('placea,'genva,'lenva)PartialState ⇒ ('placea,'genva,'lenva)PartialState
  ⇒ bool"
  "external_step valid_pid ref i j pst1c pst2c pst2c_after pst1a pst2a ==
  let

    algo2c = (algos (concrete_spec ref))!j;
```

```
  algo2a = (algos (abstract_spec ref))!j;
  algo1c = (algos (concrete_spec ref))!i;
  algo1a = (algos (abstract_spec ref))!i;
  gluing1 = (gluing ref)!i;
  gluing2 = (gluing ref)!j;
  trans2c = trans algo2c;
  trans2a = trans algo2a;
  trans1a = trans algo1a;
  pida = abstractPid ref algo2c pst2c pst2c_after
in
i = (algo_index pst1c) ∧
j = (algo_index pst2c) ∧
linv algo2a valid_pid pst2a ∧
linv algo2c valid_pid pst2c ∧
trans2c pst2c pst2c_after ∧
gluing2 pst2c pst2a ∧
coherent_state pst2c pst1c ∧
coherent_state pst2a pst1a ∧
gluing1 pst1c pst1a ∧
(pid pst1a) = pida ∧
linv algo1c valid_pid pst1c ∧
linv algo1a valid_pid pst1a
⟶ (
  ∃ pst1a_after.(
    trans1a pst1a pst1a_after ∧
    linv algo1a valid_pid pst1a_after ∧
   (if (pid pst2c ≠ pida) then
     gluing1 (update_genv pst1c (genv pst2c_after)) pst1a_after ∧
     gluing2 pst2c_after (update_genv pst2a (genv pst1a_after)) ∧
     linv algo2a valid_pid (update_genv pst2a (genv pst1a_after))
   else (
     gluing1 pst2c_after pst1a_after
   ))
    ∧ (
   ∀ k < (length (gluing ref)).
   ∀ pst3c pst3a. (let
       algo3c = (algos (concrete_spec ref))!k;
       algo3a = (algos (abstract_spec ref))!k;
       gluing3 = (gluing ref)!k
     in
     (
     k = (algo_index pst3c) ∧
     gluing3 pst3c pst3a ∧
     coherent_state pst2c pst3c ∧
     coherent_state pst1c pst3c ∧
     coherent_state pst2a pst3a ∧
     coherent_state pst1a pst3a ∧
     (pid pst3c ≠ pid pst1c) ∧
     (pid pst3c ≠ pid pst2c) ∧
     linv algo3a valid_pid pst3a
     )
     ⟶
    (
   gluing3 (update_genv pst3c (genv pst2c_after)) (update_genv pst3a (genv pst1a_after)) ∧
   linv algo3a valid_pid (update_genv pst3a (genv pst1a_after))
   )
))))
"
```

constdefs

```
  valid_external_step :: "
  ('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement ⇒ bool"
  "valid_external_step ref ==
  ∀ i < (length (gluing ref)).
  ∀ j < (length (gluing ref)).
  ∀ pst1c pst2c pst2c_after pst1a pst2a valid_pid.
  external_step valid_pid ref i j pst1c pst2c pst2c_after pst1a pst2a
  "
```

constdefs

```
  valid_spec :: "('place,'genv,'lenv)Specification ⇒ bool"
  "valid_spec specif ==
  (length (algos specif)) ≠ 0 ∧
  (list_all
    valid_algo
    (algos specif)) ∧
  (list_all
    (valid_external_rely_spec specif)
    (algos specif))
  "
```

constdefs

```
  valid_abstractPid :: "ValidPidFunctionType ⇒ ('placec,'genvc,'lenvc)AbstractPidType ⇒ ('placec,
'genvc, 'lenvc)Algorithm ⇒ bool"
  "valid_abstractPid valid_pid abstractPid_fct anAlgo ==
  ∀ pstc pstc_after.
  valid_pid (pid pstc) ∧
  (linv anAlgo) valid_pid pstc ∧
  (trans anAlgo) pstc pstc_after
  ⟶
  (valid_pid (abstractPid_fct anAlgo pstc pstc_after))"
```

constdefs

```
  valid_refinement :: "('placec,'genvc,'lenvc,'placea,'genva,'lenva)Refinement ⇒ bool"
  "valid_refinement ref ==
  (valid_spec (concrete_spec ref)) ∧
  (valid_spec (abstract_spec ref)) ∧
  (valid_external_step ref) ∧
  length (algos (concrete_spec ref)) = length (algos (abstract_spec ref)) ∧
  length (algos (concrete_spec ref)) = length (gluing ref) ∧
  (list_all
    gluing_must_ensure_same_pid_and_algo_index
    (gluing ref)) ∧
  (∀ valid_pid.
    (list_all
      (valid_abstractPid valid_pid (abstractPid ref))
      (algos (concrete_spec ref))))
  "
```

Ensure that algorithm index of processes are valid.

constdefs

```
  valid_system :: "('place,'genv,'lenv)System ⇒ bool"
  "valid_system sys ==
  (list_all
    (λ prs. (algo prs) < length (algos (spec sys)))
    (prs sys))
  "
```

D.2.5 Self-refinement of a valid specification

It is obvious that a valid specification should refine itself. Let us check that !

```
lemma spec_refines_itself:
  assumes valid_spec: "valid_spec (specif::('placec,'genvc,'lenvc)Specification)"
  shows "valid_refinement (|
    concrete_spec = specif,
    abstract_spec = specif,
    gluing = (replicate (length (algos specif)) (op  =)),
    abstractPid = (λ algos pst pst_after. (pid pst))
  |)"
proof (auto simp add: valid_refinement_def)
  from valid_spec
  show "valid_spec specif"
    by auto
next
  show "list_all gluing_must_ensure_same_pid_and_algo_index (replicate (length (algos specif)) op
=)"
    by (auto simp add: list_all_iff all_set_conv_all_nth gluing_must_ensure_same_pid_and_algo_index_def)
next
  show "valid_external_step
     (|concrete_spec = specif, abstract_spec = specif, gluing = replicate (length (algos specif)) op
=,
       abstractPid = λalgos pst pst_after. pid pst |)"
  proof (auto simp add: valid_external_step_def external_step_def Let_def)
    fix pst1c pst2c pst2c_after
    assume same_pid: "pid (pst1c::('placec,'genvc,'lenvc)PartialState) = pid (pst2c::('placec,'genvc,'lenvc)Partia
    assume trans: "Algorithm.trans (algos specif ! algo_index (pst2c::('placec,'genvc,'lenvc)PartialState))
(pst2c::('placec,'genvc,'lenvc)PartialState) (pst2c_after::('placec,'genvc,'lenvc)PartialState)"
    assume cs_pst2c_pst1c: "coherent_state pst2c pst1c"
    from same_pid and cs_pst2c_pst1c
    have "pst2c = pst1c"
      by (auto simp add: same_pid_imply_same_state)
    with trans
    show "Algorithm.trans (algos specif ! algo_index pst1c) pst1c pst2c_after"
      by (auto)
  next
    fix pst1c pst2c pst2c_after valid_pid
    assume linv_pst2c: "linv (algos specif ! algo_index pst2c) valid_pid pst2c"
    assume trans_2c: "Algorithm.trans (algos specif ! algo_index pst2c) pst2c pst2c_after"
    assume safe_algo_index_2c: "algo_index pst2c < length (algos specif)"
    assume coherent: "coherent_state pst2c pst1c"
    assume same_pid: "pid pst1c = pid pst2c"
    from coherent and same_pid
    have same_algo_index: "algo_index pst2c = algo_index pst1c"
      by (auto simp add: coherent_state_def proc_stuttering_def)
    from valid_spec and safe_algo_index_2c
    have "valid_algo (algos specif ! algo_index pst2c)"
      by (auto simp add: valid_spec_def list_all_iff all_set_conv_all_nth)
    hence "stut_or_one_step (algos specif ! algo_index pst2c)"
      by (auto simp add: valid_algo_def)
    hence "∀pst1 pst1_after.
       (linv (algos specif ! algo_index pst2c) valid_pid pst1 ∧ Algorithm.trans (algos specif ! algo_index
pst2c) pst1 pst1_after ⟶
        linv (algos specif ! algo_index pst2c) valid_pid pst1_after)"
      by (auto simp add:  stut_or_one_step_def)
    with linv_pst2c and trans_2c
    have "linv (algos specif ! algo_index pst2c) valid_pid pst2c_after"
```

```
      by auto
    with same_algo_index
    show "linv (algos specif ! algo_index pst1c) valid_pid pst2c_after"
      by auto
  next
    fix pst1c pst2c pst2c_after pst3c valid_pid
    assume linv_2c: "linv (algos specif ! algo_index pst2c) valid_pid pst2c"
    assume linv_3c: "linv (algos specif ! algo_index pst3c) valid_pid pst3c"
    assume trans_2c: "Algorithm.trans (algos specif ! algo_index pst2c) pst2c pst2c_after"
    assume safe_algo_index_3c: "algo_index pst3c < length (algos specif)"
    assume safe_algo_index_2c: "algo_index pst2c < length (algos specif)"
    from valid_spec and safe_algo_index_3c
    have valid_algo_3c: "valid_algo (algos specif ! algo_index pst3c)"
      by (auto simp add: valid_spec_def list_all_iff all_set_conv_all_nth)
    hence valid_linv_rely: "valid_linv_re_rely (algos specif ! algo_index pst3c)"
      by (auto simp add: valid_algo_def)
    from valid_spec
    have "(list_all
    (valid_external_rely_spec specif)
    (algos specif))"
      by (auto simp add: valid_spec_def)
    with safe_algo_index_2c
    have "valid_external_rely_spec specif (algos specif ! algo_index pst2c)"
      by (auto simp add: list_all_iff all_set_conv_all_nth)
    with safe_algo_index_3c
    have "valid_external_rely (algos specif ! algo_index pst2c) (algos specif ! algo_index pst3c)"
      by (auto simp add: valid_external_rely_spec_def list_all_iff all_set_conv_all_nth)
    with trans_2c and linv_2c
    have rely_3c: "Algorithm.rely (algos specif ! algo_index pst3c) pst3c (genv pst2c_after)"
      by (auto simp add: valid_external_rely_def)
    with linv_3c and valid_linv_rely
    show "linv (algos specif ! algo_index pst3c) valid_pid (update_genv pst3c (genv pst2c_after))"
      by (auto simp add: valid_linv_re_rely_def)
  qed
next
  fix valid_pid
  show "list_all (valid_abstractPid valid_pid (%algos pst pst_after. pid pst)) (algos specif)"
    by (auto simp add: list_all_iff all_set_conv_all_nth valid_abstractPid_def)
qed
```

D.2.6 A few helper lemmas about valid transition

```
lemma update_stc_step:
  assumes trans: "trans (get_algo sys (pid pstc)) (extract_pst sys stc (pid pstc)) pstc"
  assumes safe_pid: "(pid pstc) < length (prs_states stc)"
  and valid_algo_pstc: "(algo_index pstc) = algo (prs sys ! pid pstc)"
  and valid_algo: "valid_algo (get_algo sys (pid pstc))"
  and valid_state_stc: "length (prs_states stc) = length (prs sys)"
  shows "is_valid_trans sys stc (update_stc stc pstc)"
proof (cases "pstc = (extract_pst sys stc (pid pstc))")
  assume unchanged_pst: "pstc = extract_pst sys stc (pid pstc)"
  with safe_pid
  have "stc = (update_stc stc pstc)"
  proof -
    def i == "pid pstc"
    from safe_pid
    have "i < length (prs_states stc)"
      by (auto simp add: i_def)
```

```
    hence "stc = update_stc stc (extract_pst sys stc i)"
      by (auto simp add: update_stc_unchanged)
    with unchanged_pst and i_def
    show "stc = update_stc stc pstc"
      by auto
  qed
  hence "is_stuttering stc (update_stc stc pstc) "
    by (auto simp add: is_stuttering_def)
  thus ?thesis
    by (auto simp add: is_valid_trans_def)
next
  assume trans_pst: "pstc ≠ extract_pst sys stc (pid pstc)"
  have same_length: "length (prs_states stc) = length (prs_states (update_stc stc pstc))"
    by (auto simp add: update_stc_def)
  from valid_algo and trans
  have "(stuttering (extract_pst sys stc (pid pstc)) pstc) ∨
      (local_step (extract_pst sys stc (pid pstc)) pstc)"
    by (auto simp add: valid_algo_def stut_or_one_step_def)
  with trans_pst
  have local_step_pstc: "local_step (extract_pst sys stc (pid pstc)) pstc"
    by (auto simp add: stuttering_def)
  from safe_pid and valid_algo_pstc
  have pstc_extract: "pstc = (extract_pst sys (update_stc stc pstc) (pid pstc))"
    by (rule extract_update_stc)
  with local_step_pstc
  have local_step: "local_step (extract_pst sys stc (pid pstc)) (extract_pst sys (update_stc stc pstc)
(pid pstc))"
    by auto
  have stut_other: "∀ i < length (prs_states stc). i ≠ (pid pstc) ⟶ is_proc_stuttering (prs_states
stc ! i) (prs_states (update_stc stc pstc) ! i)"
    by (auto simp add: is_proc_stuttering_def update_stc_def)
  have "is_valid_step sys stc (update_stc stc pstc)"
  proof (auto simp add: is_valid_step_def)
    from same_length
    show "length (prs_states stc) = length (prs_states (update_stc stc pstc))"
      by auto
  next
    show "∃j<length (prs sys).
       Algorithm.trans (get_algo sys j) (extract_pst sys stc j) (extract_pst sys (update_stc stc pstc)
j) ∧
       local_step (extract_pst sys stc j) (extract_pst sys (update_stc stc pstc) j) ∧
       (∀i<length (prs sys). i ≠ j ⟶ is_proc_stuttering (prs_states stc ! i) (prs_states (up-
date_stc stc pstc) ! i))"
    proof (rule exI)
      let ?j="pid pstc"
      from same_length and local_step and safe_pid and valid_state_stc and stut_other and pstc_extract
and trans
      show "?j < length (prs sys) ∧
        Algorithm.trans (get_algo sys ?j) (extract_pst sys stc ?j) (extract_pst sys (update_stc stc
pstc) ?j) ∧
        local_step (extract_pst sys stc ?j) (extract_pst sys (update_stc stc pstc) ?j) ∧
        (∀i<length (prs sys). i ≠ ?j ⟶ is_proc_stuttering (prs_states stc ! i) (prs_states (up-
date_stc stc pstc) ! i))"
        by (auto)
    qed
  qed
  thus "is_valid_trans sys stc (update_stc stc pstc)"
    by (auto simp add: is_valid_trans_def)
```

qed

lemma *trans_imply_same_algo_and_pid:*
assumes *trans: "trans anAlgo pst1 pst2"*
and *validalgo: "valid_algo anAlgo"*
shows *"(algo_index pst1) = (algo_index pst2) ∧ (pid pst1) = (pid pst2)"*
proof -
from *validalgo*
have *"stut_or_one_step anAlgo"*
by *(auto simp add: valid_algo_def)*
with *trans*
have *"*
(stuttering pst1 pst2) ∨
(local_step pst1 pst2)"
by *(auto simp add: stut_or_one_step_def)*
thus *?thesis*
by *(auto simp add: stuttering_def local_step_def)*
qed

end

D.3 Correction of the formalization

theory *Correction* **imports** *Formalisation* **begin**

D.3.1 Trace definition

A system trace is an infinite sequence of system states.

types *('place,'genv,'lenv)trace = "nat ⇒ ('place,'genv,'lenv)SystemState"*

D.3.2 Helper lemmas

A valid step means that either there is a transition by process j or it is a stuttering...

lemma *get_j_step:*
assumes *valid_step: "is_valid_step sysc stc stc_after"*
shows *"*
∃ j < length (prs sysc).(
local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j) ∧
((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j)) ∧
(∀ i<length (prs sysc). i ≠ j ⟶ is_proc_stuttering (prs_states stc ! i) (prs_states stc_after ! i)))
"
by *(auto! simp add: is_valid_step_def)*

A step of one process is a valid external step for others...

lemma *concrete_step_j:*
assumes *valid_ref: "valid_refinement ref"*
and *valid_sysc: "valid_system sysc"*
and *coherentc: "(spec sysc) = (concrete_spec ref)"*
and *valid_sysa: "valid_system sysa"*
and *coherenta: "(spec sysa) = (abstract_spec ref)"*
and *coherentsys: "coherent_sys sysc sysa"*
and *valid_step: "is_valid_step sysc stc stc_after"*
and *valid_gluing: "is_valid_gluing ref sysc sysa stc sta"*
and *valid_stc: "valid_state (λ pid. pid < length (prs sysa)) sysc stc"*
and *valid_sta: "valid_state (λ pid. pid < length (prs sysa)) sysa sta"*
shows *"∀ valid_pid. ∃ j < length (prs sysc).*

```
  local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j) ∧
  ((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j))  ∧
  (∀ i<length (prs sysc).
    (i ≠ j ⟶ is_proc_stuttering (prs_states stc ! i) (prs_states stc_after ! i)) ∧
    external_step valid_pid ref (get_algo_index sysc i) (get_algo_index sysc j) (extract_pst sysc
stc i) (extract_pst sysc stc j) (extract_pst sysc stc_after j) (extract_pst sysa sta i) (extract_pst
sysa sta j))
  "
proof -
  from valid_ref and coherentc have valid_specc: "valid_spec (spec sysc)"
    by (auto simp add: valid_refinement_def)
  from valid_ref and coherenta have valid_speca: "valid_spec (spec sysa)"
    by (auto simp add: valid_refinement_def)
  hence valid_algo_i: "∀ i < length (algos (spec sysa)). valid_algo ((algos (spec sysa))!i)"
    by (auto simp add: valid_spec_def list_all_iff)
  from valid_sysa have safea_algo_index_i: "∀ i < length (prs sysa). (get_algo_index sysa i) < length
(algos (spec sysa))"
    by (auto simp add: valid_system_def list_all_iff all_set_conv_all_nth get_algo_index_def)
  from coherentsys have same_number_proc: "length (prs sysa) = length (prs sysc)"
    by (auto simp add: coherent_sys_def list_all2_def)
  from coherentsys have same_algo_index_i: "∀ i < length (prs sysa). (get_algo_index sysa i) = (get_algo_index
sysc i)"
    by (auto simp add: get_algo_index_def coherent_sys_def list_all2_conv_all_nth)

  from valid_step
  have "∃ j < length (prs sysc).
    local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j) ∧
    ((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j)) ∧
    (∀ i<length (prs sysc). i ≠ j ⟶ is_proc_stuttering (prs_states stc ! i) (prs_states stc_after
! i))
    "
    by (rule get_j_step)
  with this  obtain j
    where safe_j: "j < length (prs sysc)"
    and step_j: "local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j)"
    and trans_j: "((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after
j))"
    and proc_stut_other_j: "(∀ i<length (prs sysc). i ≠ j ⟶ is_proc_stuttering (prs_states stc
! i) (prs_states stc_after ! i))"
    by auto

  with coherentc have trans_j: "Algorithm.trans (algos (concrete_spec ref) ! (get_algo_index sysc
j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j)"
    by (auto simp add:get_algo_def)

  from safe_j and coherentsys have safea_j: "j < length (prs sysa)"
    by (auto simp add: coherent_sys_def list_all2_def)
  from safe_j and  valid_stc have safestc_j: "j < length (prs_states stc)"
    by (auto simp add: valid_state_def)

  from coherentsys and safe_j have same_algo_index_j: "(get_algo_index sysa j) = (get_algo_index
sysc j)"
    by (auto simp add: get_algo_index_def coherent_sys_def list_all2_conv_all_nth)

  from valid_sysa and coherenta and safea_j have "(get_algo_index sysa j) < length (algos (abs-
tract_spec ref))"
    by (auto simp add: get_algo_index_def valid_system_def list_all_iff all_set_conv_all_nth)
  with valid_ref
```

```
  have "∀ valid_pid. ∀ i < length (prs sysc).
    external_step valid_pid ref (get_algo_index sysc i) (get_algo_index sysc j) (extract_pst sysc
stc i) (extract_pst sysc stc j) (extract_pst sysc stc_after j) (extract_pst sysa sta i) (extract_pst
sysa sta j)"
  proof (auto simp add:valid_refinement_def valid_external_step_def Let_def get_algo_index_def)
    fix i valid_pid
    assume safe_i: "i < length (prs sysc)"
    from safe_i and valid_sysc and coherentc have "(get_algo_index sysc i) < length (algos (concrete_spec
ref))"
      by (auto simp add: get_algo_index_def valid_system_def list_all_iff all_set_conv_all_nth)
    with valid_ref
    have safe_algo_index_i: "(get_algo_index sysc i) < length (algos (abstract_spec ref))"
      by (auto simp add: valid_refinement_def)
    from valid_sysc and safe_j and valid_ref and coherentc
    have safe_algo_index_j: "(get_algo_index sysc j) < length (algos (abstract_spec ref))"
      by (auto simp add:get_algo_index_def valid_system_def list_all_iff all_set_conv_all_nth valid_refineme
    assume "ALL i<length (algos (abstract_spec ref)).
             ALL j<length (algos (abstract_spec ref)).
                ALL pst1c pst2c pst2c_after pst1a pst2a valid_pid.
                   external_step valid_pid ref i j pst1c pst2c pst2c_after pst1a pst2a"
    with safe_algo_index_i and safe_algo_index_j have "
      ∀pst1c pst2c pst2c_after pst1a pst2a. external_step valid_pid ref (get_algo_index sysc i) (get_algo_in
sysc j) pst1c pst2c pst2c_after pst1a pst2a"
      by (auto)
    thus "external_step valid_pid ref (algo (prs sysc ! i)) (algo (prs sysc ! j)) (extract_pst sysc
stc i) (extract_pst sysc stc j) (extract_pst sysc stc_after j) (extract_pst sysa sta i) (extract_pst
sysa sta j)"
      by (auto simp add: get_algo_index_def)
  qed
  with safe_j and step_j and proc_stut_other_j and trans_j and coherentc
  have all_prop: "∀ valid_pid.
    local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j) &
    ((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j))  ∧
    (ALL i<length (prs sysc).
    (i ~= j -->
     is_proc_stuttering (prs_states stc ! i) (prs_states stc_after ! i)) &
    external_step valid_pid ref (get_algo_index sysc i) (get_algo_index sysc j) (extract_pst sysc
stc i) (extract_pst sysc stc j) (extract_pst sysc stc_after j) (extract_pst sysa sta i) (extract_pst
sysa sta j))
    "
    by (auto simp add: get_algo_def get_algo_index_def)
  show ?thesis
  proof (rule allI,rule exI)
    fix valid_pid
    let ?j1 = "j"
    from all_prop and safe_j
    show "?j1 < length (prs sysc) &
    local_step (extract_pst sysc stc ?j1) (extract_pst sysc stc_after ?j1) &
    ((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j))  ∧
    (ALL i<length (prs sysc).
        (i ~= ?j1 -->
          is_proc_stuttering (prs_states stc ! i) (prs_states stc_after ! i)) &
        external_step valid_pid ref (get_algo_index sysc i) (get_algo_index sysc ?j1) (extract_pst
sysc stc i) (extract_pst sysc stc ?j1)
         (extract_pst sysc stc_after ?j1) (extract_pst sysa sta i) (extract_pst sysa sta ?j1))"
      by (auto)
  qed
qed
```

D.3.3 Main lemma

The goal here is to show the forward simulation.

Given two concrete system states with a transition relation between them, given one abstract system state with a gluing relation with the former one, show that there exists a second abstract system state that match abstract transition and gluing.

```
lemma
  assumes  valid_ref: "valid_refinement ref"
  and valid_sysc: "valid_system sysc"
  and coherentc: "(spec sysc) = (concrete_spec ref)"
  and valid_sysa: "valid_system sysa"
  and coherenta: "(spec sysa) = (abstract_spec ref)"
  and coherentsys: "coherent_sys sysc sysa"
  and valid_step: "is_valid_step sysc stc stc_after"
  and valid_gluing: "is_valid_gluing ref sysc sysa stc sta"
  and valid_stc: "valid_state (λ pid. pid < length (prs sysa)) sysc stc"
  and valid_sta: "valid_state (λ pid. pid < length (prs sysa)) sysa sta"
  shows "∃ sta_after.
  is_valid_trans sysa sta sta_after ∧
  is_valid_gluing ref sysc sysa stc_after sta_after ∧
  valid_state (λ pid. pid < length (prs sysa)) sysa sta_after"
proof -
  from valid_ref and coherentc have valid_specc: "valid_spec (spec sysc)"
    by (auto simp add: valid_refinement_def)
  from valid_ref and coherenta have valid_speca: "valid_spec (spec sysa)"
    by (auto simp add: valid_refinement_def)
  hence valid_algo_i: "∀ i < length (algos (spec sysa)). valid_algo ((algos (spec sysa))!i)"
    by (auto simp add: valid_spec_def list_all_iff)
  hence stut_auth_i: "∀ i < length (algos (spec sysa)). stuttering_is_authorized_in_trans ((algos
(spec sysa))!i)"
    by (auto simp add: valid_algo_def)
  from valid_sysa have safea_algo_index_i: "∀ i < length (prs sysa). (get_algo_index sysa i) < length
(algos (spec sysa))"
    by (auto simp add: valid_system_def list_all_iff all_set_conv_all_nth get_algo_index_def)
  from valid_sysc have safec_algo_index_i: "∀ i < length (prs sysc). (get_algo_index sysc i) < length
(algos (spec sysc))"
    by (auto simp add: valid_system_def list_all_iff all_set_conv_all_nth get_algo_index_def)
  from coherentsys have same_length_sysa_sysc: "length (prs sysa) = length (prs sysc)"
    by (auto simp add: coherent_sys_def list_all2_def)
  from valid_stc have same_length_sysc_stc: "length (prs sysc) = length (prs_states stc)"
    by (auto simp add: valid_state_def)
  from valid_sta have same_length_sysa_sta: "length (prs sysa) = length (prs_states sta)"
    by (auto simp add: valid_state_def)
  from valid_step have same_length_stc_stc_after: "length (prs_states stc) = length (prs_states stc_after)"
    by (auto simp add: is_valid_step_def)
  from same_length_sysc_stc and same_length_stc_stc_after have same_length_sysc_stc_after: "length
(prs sysc) = length (prs_states stc_after)"
    by auto
  from valid_sta and same_length_sysa_sysc and same_length_sysc_stc_after have same_length_sysa_stc_after:
"length (prs sysa) = length (prs_states stc_after)"
    by (auto simp add: is_valid_step_def)
  from valid_sta and same_length_sysa_sysc and same_length_sysc_stc_after have same_length_sta_stc_after:
"length (prs_states sta) = length (prs_states stc_after)"
    by (auto simp add: valid_state_def)
  from same_length_sysc_stc and same_length_sysc_stc_after have same_length_stc_stc_after: "length
(prs_states stc) = length (prs_states stc_after)"
    by auto
  from same_length_sysa_sysc and same_length_sysa_sta
```

```
  have same_length_sysc_sta: "length (prs_states sta) = length  (prs sysc)"
    by auto
  from same_length_sysc_stc and same_length_sysc_sta
  have same_length_stc_sta: "length (prs_states stc) = length (prs_states sta)"
    by auto
  from coherentsys have same_algo_index_i: "∀ i < length (prs sysa). (get_algo_index sysa i) = (get_algo_in
sysc i)"
    by (auto simp add: get_algo_index_def coherent_sys_def list_all2_conv_all_nth)
  have get_algo_and_extract_are_same: "∀ k < length (prs sysc). get_algo_index sysc k = algo_index
(extract_pst sysc stc k)"
    by (auto simp add: get_algo_index_def extract_pst_def)
  from valid_gluing
  have valid_gluing_k: "∀ k < length (prs_states stc). (gluing ref ! get_algo_index sysc k) (ex-
tract_pst sysc stc k) (extract_pst sysa sta k)"
    by (auto simp add: is_valid_gluing_def valid_gluing_def extract_pst_def)
  from valid_sta
  have valid_linva_k: "∀ k < length (prs_states sta). linv (get_algo sysa k) (λ pid. pid < (length
(prs sysa))) (extract_pst sysa sta k)"
    by (auto simp add: valid_state_def linv_hold_def list_all_iff all_set_conv_all_nth)
  from valid_stc
  have valid_linvc_k: "∀ k < length (prs_states stc). linv (get_algo sysc k) (λ pid. pid < (length
(prs sysa))) (extract_pst sysc stc k)"
    by (auto simp add: valid_state_def linv_hold_def list_all_iff all_set_conv_all_nth)
  from stut_auth_i and safea_algo_index_i have stut_auth_proc_i: "∀ i < length (prs sysa). stut-
tering_is_authorized_in_trans (get_algo sysa i)"
    by (auto simp add: get_algo_def get_algo_index_def)

  with valid_ref and valid_sysc and coherentc and valid_sysa
    and coherenta and coherentsys and valid_step and valid_gluing
    and valid_stc and valid_sta
  have "EX j<length (prs sysc).(
       (local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j)) &
       ((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j))  ∧
       (ALL i<length (prs sysc).
           (i ~= j -->
             is_proc_stuttering (prs_states stc ! i) (prs_states stc_after ! i)) &
           (external_step (λ pid. pid < (length (prs sysa))) ref (get_algo_index sysc i) (get_algo_index
sysc j) (extract_pst sysc stc i) (extract_pst sysc stc j)
            (extract_pst sysc stc_after j) (extract_pst sysa sta i) (extract_pst sysa sta j))))
    "
    by (auto simp add: concrete_step_j)

  with this obtain j where
    safe_j: "j < length (prs sysc)" and
    step_j: "local_step (extract_pst sysc stc j) (extract_pst sysc stc_after j)" and
    esj: "∀ i<length (prs sysc).external_step (λ pid. pid < (length (prs sysa))) ref (get_algo_index
sysc i) (get_algo_index sysc j) (extract_pst sysc stc i) (extract_pst sysc stc j) (extract_pst sysc
stc_after j) (extract_pst sysa sta i) (extract_pst sysa sta j)" and
    proc_stut_other_j: "(∀ i<length (prs sysc). i ≠ j ⟶ is_proc_stuttering (prs_states stc ! i)
(prs_states stc_after ! i))" and
    trans_j: "((trans (get_algo sysc j)) (extract_pst sysc stc j) (extract_pst sysc stc_after j))"
    by (auto)
  from proc_stut_other_j
  have extract_stc_after_not_j: "∀ k<length (prs sysc). k≠ j ⟶ (extract_pst sysc stc_after k)
= update_genv (extract_pst sysc stc k) (genv (extract_pst sysc stc_after j))"
    by (auto simp add: update_genv_def extract_pst_def is_proc_stuttering_def)
  from valid_sta and valid_stc and coherentsys and valid_gluing
  have "∀ k < length (prs sysc).
```

```
    (((gluing ref)!(get_algo_index sysa k)) (extract_pst sysc stc k) (extract_pst sysa sta k))"
    by (auto simp add: is_valid_gluing_def valid_gluing_def valid_state_def extract_pst_def get_algo_index_def
coherent_sys_def list_all2_conv_all_nth)

  from valid_ref and coherentsys and valid_sysa and coherenta have safe_algo_index_k: "∀ k < length
(prs sysc). (get_algo_index sysa k) < (length (gluing ref))"
    by (auto simp add: valid_refinement_def coherent_sys_def list_all2_conv_all_nth get_algo_index_def
valid_system_def list_all_iff all_set_conv_all_nth)

  from coherentsys and coherenta have same_algo_index_k: "∀ k < length (prs sysc). (get_algo_index
sysa k) = (get_algo_index sysc k)"
    by (auto simp add: get_algo_index_def coherent_sys_def list_all2_conv_all_nth)

  from valid_gluing and same_length_sysc_stc
  have gluing_before: "∀ k < length (prs sysc). (gluing ref ! algo_index (extract_pst sysc stc k))
(extract_pst sysc stc k) (extract_pst sysa sta k)"
    by (auto simp add: is_valid_gluing_def valid_gluing_def)
  def pid1a == "((abstractPid ref) (get_algo sysc j) (extract_pst sysc stc j) (extract_pst sysc stc_after
j))"
  from valid_linvc_k and safe_j and same_length_sysc_stc
  have linvc_j: "linv (get_algo sysc j) (λ pid. pid < (length (prs sysa))) (extract_pst sysc stc
j)"
    by auto
  from valid_ref and safe_j and safec_algo_index_i and coherentc
  have valid_abstractPid_j: "valid_abstractPid (λ pid. pid < (length (prs sysa))) (abstractPid ref)
(get_algo sysc j)"
    by (auto simp add: valid_refinement_def list_all_iff all_set_conv_all_nth get_algo_def)
  have valid_pid_extract_j: "pid (extract_pst sysc stc j) < length (prs sysa)"
  proof -
    from pid_is_nth
    have "pid (extract_pst sysc stc j) = j"
      by auto
    with safe_j
    have "pid (extract_pst sysc stc j) < length (prs sysc)"
      by auto
    with same_length_sysa_sysc
    show ?thesis
      by auto
  qed
  from linvc_j and valid_abstractPid_j and trans_j and valid_pid_extract_j
  have safe_pid1a: "pid1a < length (prs sysa)"
    by (auto simp add: valid_abstractPid_def pid1a_def)
  show ?thesis
  proof -
    from safe_pid1a and same_length_sysa_sysc
    have safec_pid1a: "pid1a < length (prs sysc)"
      by (auto)
    from safe_pid1a and esj and same_length_sysa_sysc
    have esj_pid1a: "external_step (λ pid. pid < (length (prs sysa))) ref (get_algo_index sysc pid1a)
(get_algo_index sysc j) (extract_pst sysc stc pid1a) (extract_pst sysc stc j) (extract_pst sysc stc_after
j) (extract_pst sysa sta pid1a) (extract_pst sysa sta j)"
      by (auto)
    from get_algo_and_extract_are_same and safe_j and safec_pid1a and gluing_before
    have ante_esj:"
      let
      algo_i = (get_algo_index sysc pid1a);
      algo_j = (get_algo_index sysc j);
      pst1c = (extract_pst sysc stc pid1a);
```

```
      pst2c = (extract_pst sysc stc j);
      pst2c_after = (extract_pst sysc stc_after j);
      pst1a = (extract_pst sysa sta pid1a);
      pst2a = (extract_pst sysa sta j);
      algo2c = (algos (concrete_spec ref))!algo_j;
      algo2a = (algos (abstract_spec ref))!algo_j;
      algo1c = (algos (concrete_spec ref))!algo_i;
      algo1a = (algos (abstract_spec ref))!algo_i;
      gluing1 = (gluing ref)!algo_i;
      gluing2 = (gluing ref)!algo_j;
      trans2c = trans algo2c;
      trans2a = trans algo2a;
      trans1a = trans algo1a;
      pida = abstractPid ref algo2c pst2c pst2c_after
      in
      algo_i = (algo_index pst1c) ∧
      algo_j = (algo_index pst2c) ∧
      linv algo2a (λ pid. pid < (length (prs sysa))) pst2a ∧
      linv algo2c (λ pid. pid < (length (prs sysa))) pst2c ∧
      trans2c pst2c pst2c_after ∧
      gluing2 pst2c pst2a ∧
      coherent_state pst2c pst1c ∧
      coherent_state pst2a pst1a ∧
      gluing1 pst1c pst1a ∧
      (pid pst1a) = pida ∧
      linv algo1c (λ pid. pid < (length (prs sysa))) pst1c ∧
      linv algo1a (λ pid. pid < (length (prs sysa))) pst1a"
    proof (auto simp add: extract_coherent_state Let_def pid_is_nth)
      fix k
      from trans_j and coherentc
      show "Algorithm.trans (algos (concrete_spec ref) ! algo_index (extract_pst sysc stc j)) (ex-
tract_pst sysc stc j) (extract_pst sysc stc_after j)"
        by (auto simp add: get_algo_def get_algo_index_def extract_pst_def)
    next
      fix k
      from coherentc
      show "pid1a =
           abstractPid ref (algos (concrete_spec ref) ! algo_index (extract_pst sysc stc j)) (ex-
tract_pst sysc stc j)
            (extract_pst sysc stc_after j)"
        by (auto simp add: pid1a_def get_algo_def get_algo_index_def extract_pst_def )
    next
      from valid_sta and safe_j and coherenta and same_length_sysc_sta and coherentsys
      show "linv (algos (abstract_spec ref) ! algo_index (extract_pst sysc stc j)) (λ pid. pid <
(length (prs sysa))) (extract_pst sysa sta j)"
        by (auto simp add: valid_state_def linv_hold_def get_algo_def get_algo_index_def extract_pst_def
coherent_sys_def list_all2_conv_all_nth)
    next
      from valid_sta and safe_pid1a and coherenta and same_length_sysc_sta and coherentsys
      show "linv (algos (abstract_spec ref) ! algo_index (extract_pst sysc stc pid1a)) (λ pid. pid
< (length (prs sysa))) (extract_pst sysa sta pid1a)"
        by (auto simp add: valid_state_def linv_hold_def get_algo_def get_algo_index_def extract_pst_def
coherent_sys_def list_all2_conv_all_nth)
    next
      from valid_stc and safec_pid1a and coherentc
      show "linv (algos (concrete_spec ref) ! algo_index (extract_pst sysc stc pid1a)) (λ pid. pid
< (length (prs sysa))) (extract_pst sysc stc pid1a)"
        by (auto simp add: valid_state_def linv_hold_def get_algo_def get_algo_index_def extract_pst_def
```

```
coherent_sys_def list_all2_conv_all_nth)
    next
      from valid_stc and safe_j and coherentc
      show "linv (algos (concrete_spec ref) ! algo_index (extract_pst sysc stc j)) (λ pid. pid <
(length (prs sysa))) (extract_pst sysc stc j)"
        by (auto simp add: valid_state_def linv_hold_def get_algo_def get_algo_index_def extract_pst_def
coherent_sys_def list_all2_conv_all_nth)
    qed
    with esj_pid1a
    have "      let
      algo_i = (get_algo_index sysc pid1a);
      algo_j = (get_algo_index sysc j);
      pst1c = (extract_pst sysc stc pid1a);
      pst2c = (extract_pst sysc stc j);
      pst2c_after = (extract_pst sysc stc_after j);
      pst1a = (extract_pst sysa sta pid1a);
      pst2a = (extract_pst sysa sta j);
      algo2c = (algos (concrete_spec ref))!algo_j;
      algo2a = (algos (abstract_spec ref))!algo_j;
      algo1c = (algos (concrete_spec ref))!algo_i;
      algo1a = (algos (abstract_spec ref))!algo_i;
      gluing1 = (gluing ref)!algo_i;
      gluing2 = (gluing ref)!algo_j;
      trans2c = trans algo2c;
      trans2a = trans algo2a;
      trans1a = trans algo1a;
      pida = abstractPid ref algo2c pst2c pst2c_after
in
   ∃ pst1a_after.(
      trans1a pst1a pst1a_after ∧
      linv algo1a (λ pid. pid < (length (prs sysa))) pst1a_after ∧
     (if (pid pst2c ≠ pida) then
       gluing1 (update_genv pst1c (genv pst2c_after)) pst1a_after ∧
       gluing2 pst2c_after (update_genv pst2a (genv pst1a_after)) ∧
      linv algo2a (λ pid. pid < (length (prs sysa))) (update_genv pst2a (genv pst1a_after))
     else
       gluing1 pst2c_after pst1a_after)
      ∧ (
     ∀ k < (length (gluing ref)).
     ∀ pst3c pst3a. (let
         algo3c = (algos (concrete_spec ref))!k;
         algo3a = (algos (abstract_spec ref))!k;
         gluing3 = (gluing ref)!k
       in
       (
       k = (algo_index pst3c) ∧
       gluing3 pst3c pst3a ∧
       coherent_state pst2c pst3c ∧
       coherent_state pst1c pst3c ∧
       coherent_state pst2a pst3a ∧
       coherent_state pst1a pst3a ∧
       (pid pst3c ≠ pid pst1c) ∧
       (pid pst3c ≠ pid pst2c) ∧
       linv algo3a (λ pid. pid < (length (prs sysa))) pst3a
       )
       ⟶
      (
     gluing3 (update_genv pst3c (genv pst2c_after)) (update_genv pst3a (genv pst1a_after)) ∧
```

```
      linv algo3a (λ pid. pid < (length (prs sysa))) (update_genv pst3a (genv pst1a_after))
     ))))
"
      by (auto simp add: Let_def external_step_def)
    from this obtain pst1a_after
      where trans1: "trans ((algos (abstract_spec ref))!(get_algo_index sysc pid1a)) (extract_pst
sysa sta pid1a) pst1a_after"
      and linv1: "linv ((algos (abstract_spec ref))!(get_algo_index sysc pid1a)) (λ pid. pid < (length
(prs sysa))) pst1a_after"
      and gluing1and2: "let
      algo_i = (get_algo_index sysc pid1a);
      algo_j = (get_algo_index sysc j);
      pst1c = (extract_pst sysc stc pid1a);
      pst2c = (extract_pst sysc stc j);
      pst2c_after = (extract_pst sysc stc_after j);
      pst1a = (extract_pst sysa sta pid1a);
      pst2a = (extract_pst sysa sta j);
      algo2c = (algos (concrete_spec ref))!algo_j;
      algo2a = (algos (abstract_spec ref))!algo_j;
      algo1c = (algos (concrete_spec ref))!algo_i;
      algo1a = (algos (abstract_spec ref))!algo_i;
      gluing1 = (gluing ref)!algo_i;
      gluing2 = (gluing ref)!algo_j;
      trans2c = trans algo2c;
      trans2a = trans algo2a;
      trans1a = trans algo1a;
      pida = abstractPid ref algo2c pst2c pst2c_after
in
     (if (pid pst2c ≠ pida) then
       gluing1 (update_genv pst1c (genv pst2c_after)) pst1a_after ∧
       gluing2 pst2c_after (update_genv pst2a (genv pst1a_after)) ∧
       linv algo2a (λ pid. pid < (length (prs sysa))) (update_genv pst2a (genv pst1a_after))
     else
       gluing1 pst2c_after pst1a_after)"
      and gluing3: "let
      algo_i = (get_algo_index sysc pid1a);
      algo_j = (get_algo_index sysc j);
      pst1c = (extract_pst sysc stc pid1a);
      pst2c = (extract_pst sysc stc j);
      pst2c_after = (extract_pst sysc stc_after j);
      pst1a = (extract_pst sysa sta pid1a);
      pst2a = (extract_pst sysa sta j);
      algo2c = (algos (concrete_spec ref))!algo_j;
      algo2a = (algos (abstract_spec ref))!algo_j;
      algo1c = (algos (concrete_spec ref))!algo_i;
      algo1a = (algos (abstract_spec ref))!algo_i;
      gluing1 = (gluing ref)!algo_i;
      gluing2 = (gluing ref)!algo_j;
      trans2c = trans algo2c;
      trans2a = trans algo2a;
      trans1a = trans algo1a
in
∀ k < (length (gluing ref)).
     ∀ pst3c pst3a. (let
         algo3c = (algos (concrete_spec ref))!k;
         algo3a = (algos (abstract_spec ref))!k;
         gluing3 = (gluing ref)!k
       in
```

```
      (
      k = (algo_index pst3c) ∧
      gluing3 pst3c pst3a ∧
      coherent_state pst2c pst3c ∧
      coherent_state pst1c pst3c ∧
      coherent_state pst2a pst3a ∧
      coherent_state pst1a pst3a ∧
      (pid pst3c ≠ pid pst1c) ∧
      (pid pst3c ≠ pid pst2c) ∧
      (linv algo3a (λ pid. pid < (length (prs sysa))) pst3a)
      )
      ⟶
     (
     gluing3 (update_genv pst3c (genv pst2c_after)) (update_genv pst3a (genv pst1a_after)) ∧
     linv algo3a (λ pid. pid < (length (prs sysa))) (update_genv pst3a (genv pst1a_after))
    ))"
     by (auto simp add: Let_def pid1a_def)
   from trans1 and coherenta and coherentsys and safe_pid1a
   have trans1a: "trans (get_algo sysa pid1a) (extract_pst sysa sta pid1a) pst1a_after"
     by (auto simp add: get_algo_def coherent_sys_def list_all2_conv_all_nth get_algo_index_def)
   from valid_algo_i and safe_pid1a and coherenta and same_length_sysa_sta and valid_sysa
   have valid_algo_pid1a: "valid_algo (get_algo sysa pid1a)"
     by (auto simp add: get_algo_def get_algo_index_def valid_system_def list_all_iff all_set_conv_all_nth)
   with trans1a
   have same_pid_and_algo_index_pst1a_after: "pid pst1a_after = pid (extract_pst sysa sta pid1a)
∧ (algo_index (extract_pst sysa sta pid1a)) = (algo_index pst1a_after)"
     by (auto simp add: trans_imply_same_algo_and_pid)
   hence pid_of_pst1a_after: "pid pst1a_after = pid1a"
     by (auto simp add: pid_is_nth)
   from same_pid_and_algo_index_pst1a_after
   have algo_pst1a_after: "(algo_index pst1a_after) = algo (prs sysa ! pid pst1a_after)"
     by (auto simp add: extract_pst_def get_algo_index_def)
   def sta_after == "update_stc sta pst1a_after"
   have same_length_sta_sta_after: "length (prs_states sta_after) = length (prs_states sta)"
     by (auto simp add: sta_after_def update_stc_def)
   with same_length_sysc_sta
   have same_length_sysc_sta_after: "length (prs_states sta_after) = length (prs sysc )"
     by auto
   have extract_pid1a: "pst1a_after = (extract_pst sysa sta_after pid1a)"
   proof -
     from safe_pid1a and pid_of_pst1a_after and same_length_sysa_sta
     have good_pid: "(pid pst1a_after) < length (prs_states sta)"
       by (auto)
     from same_pid_and_algo_index_pst1a_after
     have good_algo: "(algo_index pst1a_after) = algo (prs sysa ! pid pst1a_after)"
       by (auto simp add: extract_pst_def get_algo_index_def)
     from good_pid and good_algo and  pid_of_pst1a_after
     show ?thesis
       by(auto simp add: extract_update_stc sta_after_def)
   qed
   from  pid_of_pst1a_after
   have extract_not_pid1a: "∀ k. k ≠ pid1a ⟶ (extract_pst sysa sta_after k) = (update_genv (ex-
tract_pst sysa sta k) (genv pst1a_after))"
     by (auto simp add: sta_after_def extract_pst_def update_genv_def update_stc_def)
   with extract_pid1a
   have extract_not_pst1a: "∀ k. k ≠ pid1a ⟶ (extract_pst sysa sta_after k) = (update_genv (ex-
tract_pst sysa sta k) (genv (extract_pst sysa sta_after pid1a)))"
     by auto
```

```
    have "is_valid_trans sysa sta sta_after ∧ is_valid_gluing ref sysc sysa stc_after sta_after ∧
valid_state (λ pid. pid < length (prs sysa)) sysa sta_after"
    proof (auto)
      show "is_valid_trans sysa sta sta_after"
      proof -
        from trans1 and pid_of_pst1a_after and coherenta and coherentsys and safe_pid1a
        have trans_pst1a: "trans (get_algo sysa (pid pst1a_after)) (extract_pst sysa sta (pid pst1a_after))
pst1a_after"
          by (auto simp add: get_algo_def coherent_sys_def list_all2_conv_all_nth get_algo_index_def)
        from pid_of_pst1a_after and safe_pid1a and same_length_sysa_sta
        have safe_pid_pst1a: "(pid pst1a_after) < length (prs_states sta)"
          by auto
        from valid_algo_pid1a and pid_of_pst1a_after
        have "stut_or_one_step (get_algo sysa (pid pst1a_after))"
          by (auto simp add: valid_algo_def)
        hence "∀ pst1 pst1_after.(
            (((trans (get_algo sysa (pid pst1a_after))) pst1 pst1_after)
            ⟶
            (
              (stuttering pst1 pst1_after) ∨
              (local_step pst1 pst1_after)
            )
            ))"
          by (auto simp add: stut_or_one_step_def)
        with trans_pst1a
        have "stuttering (extract_pst sysa sta (pid pst1a_after)) pst1a_after ∨
          local_step (extract_pst sysa sta (pid pst1a_after)) pst1a_after"
          by auto
        hence "(algo_index pst1a_after) = algo_index  (extract_pst sysa sta (pid pst1a_after))"
          by (auto simp add: stuttering_def local_step_def)
        hence valid_algo_pst1a: "(algo_index pst1a_after) = algo (prs sysa ! pid pst1a_after)"
          by (auto simp add: extract_pst_def get_algo_index_def)
        from same_length_sysa_sta
        have valid_state_sta: "length (prs_states sta) = length (prs sysa)"
          by auto
        from safe_pid_pst1a and same_length_sysa_sta
        have safe_pid: "pid pst1a_after < length (prs_states sta)"
          by (auto)
        from trans_pst1a and safe_pid and valid_algo_pst1a and valid_algo_pid1a and valid_state_sta
and pid_of_pst1a_after
        show ?thesis
          by (auto simp add: update_stc_step sta_after_def)
      qed
    next
      show "is_valid_gluing ref sysc sysa stc_after sta_after"
      proof (auto simp add: is_valid_gluing_def)
        from same_length_sysa_sta and same_length_stc_stc_after and same_length_sysa_sysc and same_length_
        show "length (prs_states sta_after) = length (prs_states stc_after)"
          by (auto simp add: update_stc_def sta_after_def)
      next
        fix i
        assume safe_i: "i < length (prs_states sta_after)"
        with same_length_sysc_sta and same_length_sta_sta_after
        have safec_i: "i < length (prs sysc)"
          by auto
        show "valid_gluing ref (extract_pst sysc stc_after i) (extract_pst sysa sta_after i)"
        proof (auto simp add: valid_gluing_def)
          from coherentc
```

```
          have pid1a_def: "abstractPid ref (algos (concrete_spec ref) ! get_algo_index sysc j) (ex-
tract_pst sysc stc j) (extract_pst sysc stc_after j) = pid1a"
            by (auto simp add: pid1a_def get_algo_def)
          with gluing1and2
          have gluing: "let algo_i = get_algo_index sysc pid1a; algo_j = get_algo_index sysc j; pst1c
= extract_pst sysc stc pid1a;
      pst2c = extract_pst sysc stc j; pst2c_after = extract_pst sysc stc_after j; pst1a = extract_pst
sysa sta pid1a;
      pst2a = extract_pst sysa sta j; algo2c = algos (concrete_spec ref) ! algo_j; algo2a = algos
(abstract_spec ref) ! algo_j;
      algo1c = algos (concrete_spec ref) ! algo_i; algo1a = algos (abstract_spec ref) ! algo_i; gluing1
= gluing ref ! algo_i;
      gluing2 = gluing ref ! algo_j; trans2c = Algorithm.trans algo2c; trans2a = Algorithm.trans algo2a;
      trans1a = Algorithm.trans algo1a
  in if pid pst2c ~= pid1a
     then gluing1 (update_genv pst1c (genv pst2c_after)) pst1a_after & gluing2 pst2c_after (update_genv
pst2a (genv pst1a_after))
          else gluing1 pst2c_after pst1a_after"
            by (auto simp add: Let_def pid_is_nth)
          have gluing_j_pid1a: "(gluing ref ! algo_index (extract_pst sysc stc_after j)) (extract_pst
sysc stc_after j) (extract_pst sysa sta_after j) ∧ (gluing ref ! algo_index (extract_pst sysc stc_after
pid1a)) (extract_pst sysc stc_after pid1a) (extract_pst sysa sta_after pid1a)"
          proof (cases "j = pid1a")
            assume j_eq_pid1a: "j = pid1a"
            with extract_pid1a and gluing
            show "(gluing ref ! algo_index (extract_pst sysc stc_after j)) (extract_pst sysc stc_after
j) (extract_pst sysa sta_after j) ∧ (gluing ref ! algo_index (extract_pst sysc stc_after pid1a)) (ex-
tract_pst sysc stc_after pid1a)
         (extract_pst sysa sta_after pid1a)"
              by (auto simp add: Let_def pid_is_nth get_algo_index_def extract_pst_def)
          next
            assume j_neq_pid1a: "j ≠ pid1a"
            with gluing and extract_not_pid1a and extract_pid1a
            show "(gluing ref ! algo_index (extract_pst sysc stc_after j)) (extract_pst sysc stc_after
j) (extract_pst sysa sta_after j) ∧ (gluing ref ! algo_index (extract_pst sysc stc_after pid1a)) (ex-
tract_pst sysc stc_after pid1a)
         (extract_pst sysa sta_after pid1a)"
            proof (clarsimp simp add: Let_def pid_is_nth get_algo_index_def)
              assume gluing1: "(gluing ref ! algo (prs sysc ! pid1a)) (update_genv (extract_pst sysc
stc pid1a) (genv (extract_pst sysc stc_after j))) (extract_pst sysa sta_after pid1a)"
              assume gluing2: "(gluing ref ! algo (prs sysc ! j)) (extract_pst sysc stc_after j)
(update_genv (extract_pst sysa sta j) (genv (extract_pst sysa sta_after pid1a)))"
              with j_neq_pid1a and extract_not_pst1a
              have "(gluing ref ! algo (prs sysc ! j)) (extract_pst sysc stc_after j) (extract_pst
sysa sta_after j)"
                by auto
              hence gluingj: "(gluing ref ! algo_index (extract_pst sysc stc_after j)) (extract_pst
sysc stc_after j) (extract_pst sysa sta_after j)"
                by (auto simp add: extract_pst_def get_algo_index_def)
              from gluing1 and j_neq_pid1a and extract_stc_after_not_j and safec_pid1a
              have gluingpid1a: "(gluing ref ! algo (prs sysc ! pid1a)) (extract_pst sysc stc_after
pid1a) (extract_pst sysa sta_after pid1a)"
                by (auto)
              from gluingpid1a and gluingj and gluing2
              show "(gluing ref ! algo_index (extract_pst sysc stc_after j)) (extract_pst sysc stc_after
j)
         (update_genv (extract_pst sysa sta j) (genv (extract_pst sysa sta_after pid1a))) &
        (gluing ref ! algo_index (extract_pst sysc stc_after pid1a)) (extract_pst sysc stc_after pid1a)
```

```
        (extract_pst sysa sta_after pid1a)"
              by (auto simp add: extract_pst_def get_algo_index_def)
          qed
         qed
         from valid_sysc and safe_i and same_length_sysc_sta and same_length_sta_sta_after and
valid_ref and coherentc
         have safe_algo_index_i: "(get_algo_index sysc i) < length (gluing ref)"
           by (auto simp add: get_algo_index_def valid_system_def list_all_iff all_set_conv_all_nth
valid_refinement_def)
         with gluing3
         have "let
           k = get_algo_index sysc i;
           algo_i = get_algo_index sysc pid1a; algo_j = get_algo_index sysc j; pst1c = extract_pst
sysc stc pid1a;
     pst2c = extract_pst sysc stc j; pst2c_after = extract_pst sysc stc_after j; pst1a = extract_pst
sysa sta pid1a;
     pst2a = extract_pst sysa sta j; algo2c = algos (concrete_spec ref) ! algo_j; algo2a = algos
(abstract_spec ref) ! algo_j;
     algo1c = algos (concrete_spec ref) ! algo_i; algo1a = algos (abstract_spec ref) ! algo_i; gluing1
= gluing ref ! algo_i;
     gluing2 = gluing ref ! algo_j; trans2c = Algorithm.trans algo2c; trans2a = Algorithm.trans algo2a;
     trans1a = Algorithm.trans algo1a
  in
       ALL pst3c pst3a.
          let algo3c = algos (concrete_spec ref) ! k; algo3a = algos (abstract_spec ref) ! k; gluing3
= gluing ref ! k
           in k = algo_index pst3c &
              gluing3 pst3c pst3a &
              coherent_state pst2c pst3c &
              coherent_state pst1c pst3c &
              coherent_state pst2a pst3a & coherent_state pst1a pst3a & pid pst3c ~= pid pst1c & pid
pst3c ~= pid pst2c
           ∧ linv (algos (abstract_spec ref) ! k) (λ pid. pid < (length (prs sysa))) pst3a
-->
             gluing3 (update_genv pst3c (genv pst2c_after)) (update_genv pst3a (genv pst1a_after))
∧
             linv algo3a (λ pid. pid < (length (prs sysa))) (update_genv pst3a (genv pst1a_after))
           "
           by ( auto simp add: Let_def)
         hence gluing3_i: "let
           pst3c = (extract_pst sysc stc i);
           pst3a = (extract_pst sysa sta i);
           k = get_algo_index sysc i; algo_i = get_algo_index sysc pid1a; algo_j = get_algo_index
sysc j;
           pst1c = extract_pst sysc stc pid1a; pst2c = extract_pst sysc stc j; pst2c_after = ex-
tract_pst sysc stc_after j;
           pst1a = extract_pst sysa sta pid1a; pst2a = extract_pst sysa sta j; algo2c = algos (concrete_spec
ref) ! algo_j;
           algo2a = algos (abstract_spec ref) ! algo_j; algo1c = algos (concrete_spec ref) ! algo_i;
           algo1a = algos (abstract_spec ref) ! algo_i; gluing1 = gluing ref ! algo_i; gluing2 =
gluing ref ! algo_j;
           trans2c = Algorithm.trans algo2c; trans2a = Algorithm.trans algo2a; trans1a = Algorithm.trans
algo1a;
           algo3c = algos (concrete_spec ref) ! k; algo3a = algos (abstract_spec ref) ! k; gluing3
= gluing ref ! k
           in k = algo_index pst3c &
             gluing3 pst3c pst3a &
             coherent_state pst2c pst3c &
```

```
            coherent_state pst1c pst3c &
            coherent_state pst2a pst3a & coherent_state pst1a pst3a & pid pst3c ~= pid pst1c & pid
pst3c ~= pid pst2c  ∧
            linv (algos (abstract_spec ref) ! k) (λ pid. pid < (length (prs sysa))) pst3a
-->
            gluing3 (update_genv pst3c (genv pst2c_after)) (update_genv pst3a (genv pst1a_after))"
          by (auto simp add: Let_def)
        from valid_gluing_k and same_length_sysc_stc and safec_i
        have "let pst3c = extract_pst sysc stc i; pst3a = extract_pst sysa sta i; k = get_algo_index
sysc i;
            algo_i = get_algo_index sysc pid1a; algo_j = get_algo_index sysc j; pst1c = extract_pst
sysc stc pid1a;
            pst2c = extract_pst sysc stc j; pst2c_after = extract_pst sysc stc_after j; pst1a = ex-
tract_pst sysa sta pid1a;
            pst2a = extract_pst sysa sta j; algo2c = algos (concrete_spec ref) ! algo_j; algo2a =
algos (abstract_spec ref) ! algo_j;
            algo1c = algos (concrete_spec ref) ! algo_i; algo1a = algos (abstract_spec ref) ! algo_i;
gluing1 = gluing ref ! algo_i;
            gluing2 = gluing ref ! algo_j; trans2c = Algorithm.trans algo2c; trans2a = Algorithm.trans
algo2a;
            trans1a = Algorithm.trans algo1a; algo3c = algos (concrete_spec ref) ! k; algo3a = al-
gos (abstract_spec ref) ! k;
            gluing3 = gluing ref ! k
            in k = algo_index pst3c &
            gluing3 pst3c pst3a &
            coherent_state pst2c pst3c &
            coherent_state pst1c pst3c &
            coherent_state pst2a pst3a & coherent_state pst1a pst3a
            ∧ linv (algos (abstract_spec ref) ! k) (λ pid. pid < (length (prs sysa))) pst3a
            "
        proof (auto simp add: Let_def coherent_state_def extract_pst_def proc_stuttering_def)
          from valid_sta and safe_i and same_length_sta_sta_after and coherenta and coherentsys
          have "linv (algos (abstract_spec ref) ! get_algo_index sysc i) (λ pid. pid < (length
(prs sysa))) (extract_pst sysa sta i)"
            by (auto simp add: valid_state_def linv_hold_def get_algo_def get_algo_index_def co-
herent_sys_def list_all2_conv_all_nth)
          thus "linv (algos (abstract_spec ref) ! get_algo_index sysc i) (λ pid. pid < (length
(prs sysa)))
            (|genv = global_env sta, lenv = ProcessState.lenv (prs_states sta ! i), pc = ProcessS-
tate.pc (prs_states sta ! i), pid = i,
            algo_index = get_algo_index sysa i|)"
            by (auto simp add: extract_pst_def)
        qed
        with gluing3_i and safe_i and extract_stc_after_not_j and extract_not_pid1a and same_length_sysc_sta_
        have gluing_i: "i ≠ pid1a ∧ i ≠ j ⟶ (gluing ref ! algo_index (extract_pst sysc stc_after
i)) (extract_pst sysc stc_after i) (extract_pst sysa sta_after i)"
          by (auto simp add: Let_def pid_is_nth update_genv_def extract_pst_def)
        from gluing_j_pid1a and gluing_i
        show "(gluing ref ! algo_index (extract_pst sysc stc_after i)) (extract_pst sysc stc_after
i) (extract_pst sysa sta_after i)"
          by auto
      qed
    qed
  next
    from same_length_sysa_sta
    show "valid_state (λ pid. pid < length (prs sysa))  sysa sta_after"
    proof (auto simp add: valid_state_def update_stc_def)
      show "length (prs_states sta) = length (prs_states sta_after)"
```

```
          by (auto simp add: sta_after_def update_stc_def)
      next
        show "linv_hold  (λ pid. pid < length (prs_states sta)) sysa sta_after"
        proof (auto simp add: linv_hold_def)
          fix i
          assume safe_i: "i < length (prs sysa)"
          show "linv (get_algo sysa i) (λ pid. pid < length (prs_states sta))(extract_pst sysa sta_after
i)"
          proof (cases "i = pid1a", auto)
            from linv1 and extract_pid1a and coherentsys and coherenta and safe_pid1a and same_length_sy
            show "linv (get_algo sysa pid1a) (λ pid. pid < length (prs_states sta)) (extract_pst
sysa sta_after pid1a)"
              by (auto simp add: get_algo_def get_algo_index_def coherent_sys_def list_all2_conv_all_nth)
          next
            assume i_neq_pid1a: "i ≠ pid1a"
            with pid_of_pst1a_after
            have extract_sta_after_pst1a_after: "(extract_pst sysa sta_after i) = update_genv (ex-
tract_pst sysa sta i) (genv pst1a_after)"
              by (auto simp add: sta_after_def extract_update_stc_genv)
            have "linv (get_algo sysa i) (λ pid. pid < length (prs_states sta)) (update_genv (ex-
tract_pst sysa sta i) (genv pst1a_after))"
            proof (cases "i=j")
              assume i_eq_j: "i=j"
              with gluing1and2 and coherenta and coherentsys and i_neq_pid1a and coherentc and
safe_i and same_length_sysa_sta
              show "linv (get_algo sysa i) (λ pid. pid < length (prs_states sta)) (update_genv (ex-
tract_pst sysa sta i) (genv pst1a_after))"
                by (auto simp add: Let_def pid_is_nth get_algo_def pid1a_def get_algo_index_def co-
herent_sys_def list_all2_conv_all_nth)
            next
              assume i_neq_j: "i ≠ j"
              with i_neq_pid1a and valid_gluing_k and valid_linva_k and safe_i and same_length_stc_sta
and coherentsys
              have "let
algo_i = get_algo_index sysc pid1a; algo_j = get_algo_index sysc j; pst1c = extract_pst sysc stc pid1a;
      pst2c = extract_pst sysc stc j; pst2c_after = extract_pst sysc stc_after j; pst1a = extract_pst
sysa sta pid1a;
      pst2a = extract_pst sysa sta j; algo2c = algos (concrete_spec ref) ! algo_j; algo2a = algos
(abstract_spec ref) ! algo_j;
      algo1c = algos (concrete_spec ref) ! algo_i; algo1a = algos (abstract_spec ref) ! algo_i; gluing1
= gluing ref ! algo_i;
      gluing2 = gluing ref ! algo_j; trans2c = Algorithm.trans algo2c; trans2a = Algorithm.trans algo2a;
      trans1a = Algorithm.trans algo1a;
                pst3a = (extract_pst sysa sta i);
                pst3c = (extract_pst sysc stc i);
                k = (get_algo_index sysa i);
                algo3c = algos (concrete_spec ref) ! k; algo3a = algos (abstract_spec ref) ! k; gluing3
= gluing ref ! k
        in
                k < length (gluing ref) ∧
                k = algo_index pst3c ∧
                gluing3 pst3c pst3a ∧
                coherent_state pst2c pst3c ∧
                coherent_state pst1c pst3c ∧
                coherent_state pst2a pst3a ∧
                coherent_state pst1a pst3a ∧ pid pst3c ≠ pid pst1c ∧ pid pst3c ≠ pid pst2c ∧ linv
algo3a (λ pid. pid < length (prs_states sta)) pst3a
                "
```

```
              proof (auto simp add: Let_def get_algo_index_def coherent_sys_def extract_coherent_state
list_all2_conv_all_nth pid_is_nth)
                from coherentsys and safe_i
                show "algo (prs sysa ! i) = algo_index (extract_pst sysc stc i)"
                  by (auto simp add: extract_pst_def coherent_sys_def list_all2_conv_all_nth  get_algo_index_def)
              next
                from safe_i and same_length_sysa_sta and coherentsys and valid_gluing_k and same_length_stc_sta
                show "(gluing ref ! algo (prs sysa ! i)) (extract_pst sysc stc i) (extract_pst sysa
sta i)"
                  by (auto simp add: get_algo_index_def coherent_sys_def list_all2_conv_all_nth )
              next
                from safe_i and valid_linva_k and same_length_sysa_sta and coherenta
                show "linv (algos (abstract_spec ref) ! algo (prs sysa ! i)) (λ pid. pid < length
(prs_states sta)) (extract_pst sysa sta i)"
                  by (auto simp add: get_algo_def get_algo_index_def)
              next
                from safe_i and valid_sysa  and valid_ref and coherenta
                show "algo (prs sysa ! i) < length (gluing ref)"
                  by (auto simp add: valid_system_def list_all_iff all_set_conv_all_nth valid_refinement_def)
              qed
              with gluing3 and same_length_sysa_sta
              have "linv (algos (abstract_spec ref) ! algo_index (extract_pst sysc stc i)) (λ pid.
pid < length (prs_states sta)) (update_genv (extract_pst sysa sta i) (genv pst1a_after))"
                by (auto simp add: Let_def)
              with coherenta and coherentsys and safe_i
              show "linv (get_algo sysa i) (λ pid. pid < length (prs_states sta)) (update_genv (ex-
tract_pst sysa sta i) (genv pst1a_after))"
                by(auto simp add: get_algo_def get_algo_index_def extract_pst_def coherent_sys_def
list_all2_conv_all_nth)
            qed
            with extract_sta_after_pst1a_after
            show "linv (get_algo sysa i) (λ pid. pid < length (prs_states sta)) (extract_pst sysa
sta_after i)"
              by auto
          qed
        qed
      qed
    qed
    thus ?thesis
      by auto
  qed
qed
end
```

Bibliographie

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2) :253—284, 1991.

[2] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] S. V. Adve. Shared memory consistency models : A tutorial. Technical report, CiteSeer [http ://cs1.ist.psu.edu/cgi-bin/oai.cgi] (United States), 1995.

[4] R.-J. Back and J. von Wright. Refinement calculus, part i : Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, pages 42–66, London, UK, 1990. Springer-Verlag.

[5] C. Barrett and S. Berezin. CVC Lite : A new implementation of the cooperating validity checker. In R. Alur and D. Peled, editors, *16th Intl. Conf. Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, Boston, MA, 2004. Springer Verlag.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot : Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[7] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense ! (extended abstract). In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 363–372, Los Angeles, California, United States, 1994. ACM.

[8] L. Bush. Generic concurrent lock-free linked list. See `http://www.cs.rpi.edu/~bushl2/project_web/page5.html`.

[9] D. Cansell, D. Méry, and S. Merz. Predicate diagrams for the verification of reactive systems. In *2nd Intl. Conf. Integrated Formal Methods (IFM 2000)*, volume 1945 of *Lecture Notes in Computer Science*, pages 380–397, Dagstuhl, Germany, Nov. 2000. Springer-Verlag.

[10] Clearsy. B4free. See `http://www.b4free.com/`.

[11] M. Corporation. System.threading.interlocked. See `http://msdn.microsoft.com/en-us/library/system.threading.interlocked.aspx`.

[12] David A. Wheeler. Sloccount. See `http://www.dwheeler.com/sloccount/`.

[13] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency verification : introduction to compositional and noncompositional methods*. Cambridge University Press, 1st edition, 2001.

[14] W. DeRoever and K. Engelhardt. *Data Refinement : Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[15] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9) :569, 1965.

[16] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and J. Guy L. Steele. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA '04 : Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224, New York, NY, USA, 2004. ACM.

[17] S. Dunne. *Introducing Backward Refinement into B*, page 627. 2003.

[18] D. Déharbe, P. Fontaine, S. Ranise, and C. Ringeissen. Decision procedures for the formal analysis of software. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Intl. Coll. Theoretical Aspects of Computing (ICTAC 2007)*, volume 4281 of *Lecture Notes in Computer Science*, pages 366–370, Tunis, Tunisia, 2007. Springer.

[19] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2008.

[20] F. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 80–87, St. John's, Newfoundland, Canada, 2004. ACM.

[21] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05 : Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM.

[22] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9) :948, 1972.

[23] P. Fontaine. *Techniques for verification of concurrent systems with invariants.* PhD thesis, Institut Montefiore, Université de Liège, Belgium, Sept. 2004.

[24] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2) :5, 2007.

[25] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4) :367–382, July 1993.

[26] L. Groves and R. Colvin. Derivation of a scalable lock-free stack algorithm. *Electron. Notes Theor. Comput. Sci.*, 187 :55–74, 2007.

[27] J. V. Guttag. *The specification and application to programming of abstract data types.* PhD thesis, Toronto, Ont., Canada, Canada, 1975.

[28] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[29] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[30] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1) :124–149, January 1991.

[31] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, Mar. 2008.

[32] M. P. Herlihy and J. M. Wing. Linearizability : a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3) :463–492, 1990.

[33] Hewlett-Packard Development Company. Qprof. See `http://www.hpl.hp.com/research/linux/qprof/`.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[35] T. Hoare. The verifying compiler : A grand challenge for computing research. *J. ACM*, 50(1) :63–69, 2003.

[36] P. T. Håkan Sundell. Noble. See `http://www.cs.chalmers.se/~noble/`.

[37] Intel. *A Formal specification of Intel Itanium processor family memory ordering*, Oct. 2002.

[38] P. Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4) :592–614, 1997.

[39] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, June 1981. Printed as : Programming Research Group, Technical Monograph 25.

[40] C. B. Jones. Wanted : a compositional approach to concurrency. pages 5–15, 2003.

[41] C. P. Keir Fraser, Tim Harris. Practical lock-free data structures. See `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`.

[42] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9) :690–691, 1979.

[43] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, 1994.

[44] L. Lamport. The +CAL Algorithm Language. 2007.

[45] L. Lamport and F. B. Schneider. Pretending atomicity. Technical report, Ithaca, NY, USA, 1989.

[46] R. J. Lipton. Reduction : a new method of proving properties of systems of processes. In *POPL '75 : Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 78–86, New York, NY, USA, 1975. ACM.

[47] B. Liskov and J. M. Wing. Family values : A behavioral notion of subtyping. Technical report, Pittsburgh, PA, USA, 1993.

[48] W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us : Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3) :689–728, 2000.

[49] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes : a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII : Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, New York, NY, USA, 2008. ACM.

[50] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations – part i : untimed systems. Technical report, Amsterdam, The Netherlands, The Netherlands, 1993.

[51] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2) :145–159, 1985.

[52] J. Manson, W. Pugh, and S. V. Adve. The java memory model. *SIGPLAN Not.*, 40(1) :378–391, 2005.

[53] Max-Planck-Institut Informatik. Spass. See `http://www.spass-prover.org/`.

[54] Microsoft Corporation. Tlc. See `http://research.microsoft.com/users/lamport/tla/tla.html`.

[55] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, 8(1) :142–153, 1986.

[56] J. Misra and K. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4) :417–426, July 1981.

[57] C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[58] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2) :245–257, 1979.

[59] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2) :356–364, 1980.

[60] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll(). *J. ACM*, 53(6) :937–977, 2006.

[61] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002. See Also `http://isabelle.in.tum.de/index.html`.

[62] OpenMP Architecture Review Board. *OpenMP Application Program Interface 3.0*, May 2008.

[63] S. Owicki and D. Gries. Verifying properties of parallel programs : an axiomatic approach. *Commun. ACM*, 19(5) :279–285, 1976.

[64] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1) :297–302, 2007.

[65] Pascal Fontaine and David Déharde. haRVey. See `http://harvey.loria.fr`.

[66] L. C. Paulson. The Isabelle reference manual. Technical Report 283, 1993.

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

[68] Protheo INRIA. Tom. See `http://tom.loria.fr`.

[69] Rahul Jha and Guide Sridhar Iyer. Distributed shared memory : Concepts and systems. Technical report, CiteSeer [http ://cs1.ist.psu.edu/cgi-bin/oai.cgi] (United States), 2000.

[70] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1) :9–17, 1981.

[71] E. Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4) :1156–1168, 2000.

[72] J. Scholar. Nonblocking multiprocessor/multithread algorithms in c++. See `http://www.musicdsp.org/archive.php?classid=0#148`.

[73] J. Seigh. Atomic ptr plus. See `http://atomic-ptr-plus.sourceforge.net/`.

[74] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing*, pages 204–213, 1995.

[75] E. Shenk. The consensus hierarchy is not robust. In *PODC '97 : Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, page 279, New York, NY, USA, 1997. ACM.

[76] J. M. Spivey. *The Z notation : a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[77] Stanford University. Cvc3. See `http://www.cs.nyu.edu/acsys/cvc3/`.

[78] A. Stump, C. W. Barrett, and D. L. Dill. CVC : A cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.

[79] Sun Microsystems. java.util.concurrent.atomic. See `http://java.sun.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html`.

[80] C. Tinelli and M. T. Harandi. A new correctness proof of the {Nelson-Oppen} combination procedure. In *Frontiers of Combining Systems (FroCos)*, pages 103–119, 1996.

[81] J. D. Ullman. *Principles of Database Systems*. W. H. Freeman & Co., New York, NY, USA, 1983.

[82] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

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

[84] J. van Leeuwen. *Handbook of Theoretical Computer Science, Vol. B : Formal Models and Semantics*. The MIT Press, 1994.

[85] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08 : Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 125–135, New York, NY, USA, 2008. ACM.

[86] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description : Spass version 3.0. In F. Pfenning, editor, *21st Intl. Conf. Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520, Bremen, Germany, 2007. Springer.

[87] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs '99 : Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, London, UK, 1999. Springer-Verlag.

[88] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2) :149–174, 1997.

Résumé

Le sujet central de cette thèse est le développement d'une méthode dédiée à la preuve de structures de données sans verrou. La motivation première vient du constat que les programmes concurrents sont devenu monnaie courante. Ceci a été possible par l'apparition de nouvelles primitives de synchronisation dans les nouvelles architectures matérielles. La seconde motivation est la quête de logiciel prouvé et donc correct. La sûreté des logiciels est en effet devenue primordiale de par la diffusion des systèmes embarqués et enfouis.

La méthode proposée est basée sur le raffinement et dédiée à la conception et la vérification d'algorithme non-bloquant, en particulier ceux sans verrou. La méthode a été formalisée et sa correction prouvée en Isabelle/HOL. Un outil a par ailleurs été développé afin de générer des obligations de preuves à destination des solveurs SMT et des prouveurs de théorèmes du premier ordre. Nous l'avons utilisé afin de vérifier certains de ces algorithmes.

Mots-clés: structure de données, algorithme sans verrou, preuves, raffinement.

Abstract

The central topic of this thesis is the proof-based development of lock-free data-structure algorithms. First motivation comes from new computer architectures that come with new synchronisation features. Those features enable concurrent algorithms that do not use locks and are thus more efficient. The second motivation is the search for proved correct program. Nowadays embedded software are used everywhere included in systems where safety is central.

We propose a refinement-based method for designing and verifying non-blocking, and in particular lock-free, implementations of data structures. The entire method has been formalised in Isabelle/HOL. An associated prototype tool generates verification conditions that can be solved by SMT solvers or automatic theorem provers for first-order logic, and we have used this approach to verify a number of such algorithms.

Keywords: data structure, lock-free algorithm, proofs, refinement.