Mécanisation du modèle RC11 et de la propriété DRF-SC

Quentin Ladeveze

INRIA, France
quentin.ladeveze@inria.fr

Résumé

Un modèle mémoire décrit les comportements possibles d'un programme concurrent réalisant des accès à la mémoire partagée. Dans le modèle mémoire SC, ces comportements sont ceux que l'on pourrait observer si l'on exécutait séquentiellement un entrelacement des instructions des différents *threads* du programme. Ce modèle simple est généralement celui qui est attendu par le programmeur. Cependant, il exclut des comportements, dits relâchés, induits par les processeurs et les compilateurs lorsque ces derniers réalisent des optimisations. Pour que les utilisateurs puissent bénéficier de la simplicité de SC, tout en permettant de bonnes performances, on souhaite que les modèles mémoire de ces processeurs et de ces langages de programmation vérifient la propriété DRF-SC. Cette propriété garantit que si les exécutions conformes à SC d'un programme ne contiennent aucune *race*, le programme ne présentera aucune exécution non conforme à SC. Autrement dit, en respectant une condition simple, l'utilisateur peut ignorer le modèle mémoire de son langage et de son architecture. Le modèle RC11 est un modèle mémoire formel des programmes C/C++ respectant le standard C11. Les auteurs de ce modèle accompagnent ce dernier d'une preuve qu'il respecte la propriété DRF-SC. Nous présentons la mécanisation dans l'assistant de preuves Coq de ce modèle mémoire et de cette preuve. De plus, nous explorons les limites d'application de cette propriété dans le cadre des programmes C11. Enfin, nous discutons des façons dont cette preuve pourrait être adaptée à d'autres modèles et modifiée pour s'appliquer à une classe moins restreinte de programmes.

1 Introduction

Un modèle mémoire décrit le comportement de programmes réalisant des accès concurrents à une mémoire partagée. Dans le modèle *séquentiellement consistant* (SC) [8], le résultat de l'exécution d'un programme est tel qu'il serait si l'on exécutait ses instructions séquentiellement, en respectant l'ordre dans lequel elles apparaissent dans le programme. C'est généralement ce qui est attendu et souhaité par le programmeur.

Cependant, les programmes peuvent présenter des comportements non conformes au modèle SC. En effet, le compilateur traduisant ces programmes en instructions assembleur et le processeur exécutant ces instructions peuvent tous deux réaliser des optimisations.

```
x = 1;     || y = 1;
print(y);  || print(x);
```

(a) Possibilité d'optimisation matérielle

```
x = 1;  || print(x);
x = 2;  ||
```

(b) Possibilité d'optimisation logicielle

FIGURE 1 – Programmes pouvant présenter des comportement inattendus à cause d'optimisations

Dans le programme 1a par exemple, les écritures peuvent être d'abord réalisées dans un cache (un *store buffer*) des processeurs sur lesquels elles sont exécutées. Les lectures qui les suivent peuvent ensuite être effectuées avant que ces écritures aient atteint la mémoire centrale. Ce programme pourrait avoir pour résultat d'afficher deux fois 0 (la valeur initial de `x` et `y`).

Dans le programme 1b, l'écriture de la valeur 1 à l'emplacement `x` peut être supprimée par le compilateur, car elle précède directement l'écriture d'une autre valeur au même emplacement.

Ce programme ne pourrait donc jamais avoir pour résultat d'afficher la valeur 1, alors que c'est un des comportement que l'on peut attendre.

Ces optimisations n'affectent pas le résultat des programmes séquentiels, mais peuvent donner des comportements différents aux programmes concurrents par rapport à ceux attendus dans le modèle SC.

Un comportement non conforme à SC correspond le plus souvent à ce qu'un des *threads* du programme perçoive les effets des accès mémoire d'un autre *thread* dans un autre ordre que celui du programme. On dit que des instructions du programme peuvent être *réordonnées*. Lorsque de tels comportements sont possibles, on se trouve alors dans un modèle mémoire dit *relâché*.

Pour l'écriture et l'utilisation d'un compilateur d'un langage de haut niveau, le modèle mémoire doit être pris en compte à deux niveaux. Premièrement, il est souhaitable pour ce compilateur d'offrir un modèle mémoire uniforme, quelle que soit l'architecture cible. Le compilateur doit donc prendre en compte le modèle mémoire de chacune des architectures qu'il supporte, et générer un code assembleur en conséquence. Deuxièmement, le programmeur doit prendre en compte le modèle mémoire fourni par le compilateur pour anticiper les comportements de ses programmes.

Afin d'éviter les ambiguïtés, on peut exprimer les comportements possibles dans un modèle mémoire formalisé mathématiquement, qui définit précisément l'ensemble des résultats attendus lorsque l'on exécute un programme donné. Ce modèle mémoire formel définit notamment l'ensemble des instructions qui peuvent être réordonnées.

Un des formalismes possibles pour décrire un modèle mémoire est le formalisme axiomatique [1,2]. Dans ce cadre, une exécution est représentée par un ensemble de relations entre des accès à la mémoire partagée. On définit ensuite la *conformité* d'une exécution au modèle mémoire par un ensemble de contraintes, généralement d'acyclicité, sur ces relations.

Même lorsqu'ils sont formalisés précisément, les modèles mémoires des architectures et des langages les plus répandus restent souvent complexes à utiliser pour le programmeur. Pour cette raison, ces modèles sont généralement conçus pour vérifier la propriété DRF-SC : si aucune des exécutions conformes au modèle SC d'un programme ne contient de *races*, alors toutes les exécutions de ce programme conformes à un modèle qui respecte la propriété DRF-SC sont conformes au modèle SC. Une *race* est un accès simultané par deux *threads*, dont au moins une écriture, à un même emplacement mémoire (pour une définition formelle de cette notion, voir la section 3.2). Cette propriété permet au programmeur d'ignorer complètement le modèle mémoire relâché et de considérer le modèle SC à la place, à condition de respecter certaines règles.

Pour faciliter la vie du programmeur, certains langages proposent plusieurs types d'accès à la mémoire partagée. On peut les classer en deux grandes catégories : non-atomiques et atomiques.

Les premiers sont prévus pour accéder aux données locales à un *thread*, et une *race* entre deux accès de ce type est considérée comme une erreur de programmation.

Les seconds sont spécifiquement conçus pour la communication entre *threads*. Le compilateur fournit donc des garanties sur l'ordre dans lequel ils seront exécutés. Pour ce faire, le compilateur utilise des instructions assembleur supplémentaires destinées à synchroniser ces accès, comme par exemple des barrières. Ces accès à la mémoire sont donc moins efficaces que des accès mémoire non-atomiques.

RC11 [7] est un modèle mémoire axiomatique pour les programmes C/C++ respectant le standard C11. Ce modèle mémoire est accompagnée d'une preuve stipulant qu'il vérifie bien la condition DRF-SC. Nous présentons la formalisation dans l'assistant de preuves Coq de ce modèle et de cette preuve.

2 Le modèle RC11

Nous commencerons par présenter les détails du modèle RC11. Nous décrirons dans un premier temps les fonctionnalités spécifiques du standard C11 qui aident l'utilisateur à contrôler les comportements relâchés de son modèle mémoire. Nous poursuivrons par la formulation axiomatique du modèle qui se découpe en deux étapes. On génère d'abord l'ensemble des *exécutions* (représentées par des graphes) possibles d'un programme. On filtre ensuite ces exécutions pour ne garder que celles qui sont conformes au modèle en leur appliquant des prédicats de conformité.

2.1 Contrôle de la conformité

Le standard C11 contient plusieurs outils destinés à aider les programmeurs à écrire des programmes concurrents correctement synchronisés.

Premièrement, il définit plusieurs types d'accès atomiques. Chacun d'entre eux fournit des garanties plus ou moins fortes sur l'ordre d'exécution des accès, avec une pénalité de performance en conséquence :

1. Les atomiques SC sont les plus forts et les plus coûteux à implémenter. La sémantique souhaitée est qu'un programme qui ne comprend des *races* que sur des accès atomiques SC ait un comportement conforme à SC.
2. Les atomiques *release-acquire* (RA) ont pour objectif de permettre de passer des messages entre plusieurs threads, mais sans le coût d'accès SC. Les écritures en mémoire peuvent être des opérations *release* et les lectures peuvent être des opérations *acquire*. La sémantique souhaitée pour ces accès est la suivante : lorsqu'une lecture *acquire* lit une valeur écrite par une écriture *release* ou une écriture atomique qui suit une écriture *release* dans le programme, les effets de tous les accès à la mémoire précédant l'écriture *release* dans l'ordre du programme sont visibles pour tous les accès mémoire qui suivent la lecture *acquire* dans l'ordre du programme.
3. Les atomiques relâchés sont traduits par le compilateur en un accès unique à la mémoire, sans l'ajout d'aucune primitive de synchronisation dans le code assembleur. L'atomicité de ces accès est garanti, ce qui peut être important lorsque l'on accède une valeur de 64 bits sur une machine 32bits par exemple.

Deuxièmement, le standard fournit des *barrières* utilisables dans le langage de haut niveau, qui permettent d'indiquer au compilateur qu'il doit insérer des barrières au niveau assembleur à un emplacement spécifique. Comme pour les accès atomiques, il existe plusieurs types de barrières :

1. Les barrières peuvent être *release*, *acquire* ou les deux. Ces barrières peuvent se combiner de trois façons.
   - lorsqu'une barrière *release* précède une écriture atomique dont la valeur est lue par une lecture atomique suivie d'une barrière *acquire*, les effets des accès précédant la barrière *release* sont visibles pour les accès qui suivent la barrière *acquire*.
   - lorsqu'une barrière *release* précède une écriture atomique donc la valeur est lue par une lecture *acquire*, les effets des accès précédant la barrière *release* sont visibles pour les accès qui suivent la lecture *acquire*.
   - lorsque la valeur d'une écriture *release* est lue par une lecture atomique suivie d'une barrière *acquire*, les effets des accès précédant la lecture *release* sont visibles pour les accès qui suivent la barrière *acquire*.
2. Une barrière SC interdit tout réordonnancement entre des accès effectués par les instructions qui la précèdent et celles qui la suivent.

Enfin, le standard définit une opération `atomic_exchange` qui permet de lire la valeur stockée à un emplacement, puis d'y écrire une nouvelle valeur. On nomme ces opérations des accès *read-modify-write*. Le standard garantit que la valeur inscrite à cet emplacement mémoire ne sera pas modifiée entre la lecture et l'écriture qui correspondent à cette opération.

2.2 Notations

Avant de décrire l'ensemble des exécutions associées à un programme, nous introduisons quelques notations. On note R^+, $R^?$ et R^* respectivement la clôture transitive, réflexive et réflexive-transitive d'une relation binaire R. On note R^{-1} l'inverse de la relation R. On note $R_1; R_2$ la composition de deux relations R_1 et R_2. L'opérateur de composition est prioritaire par rapport à l'opérateur d'union. Une relation R est acyclique si R^+ est irréflexive.

On note $[A]$ la relation identité restreinte à l'ensemble A. On note $\mathsf{dom}(R)$ et $\mathsf{codom}(R)$ respectivement le domaine et le codomaine de la relation R.

Pour une fonction f, on note $=_f$ l'ensemble des paires d'éléments dont l'image par f est identique et $\neq_f$ l'ensemble des paires d'éléments dont l'image par f est différente. On note $R|_f$, la relation $(R \cap =_f)$ et $R|_{\neq f}$ la relation $(R \cap \neq_f)$.

Si R est un ordre partiel strict, $R|_{imm}$ désigne les *liens immédiats* dans R, c'est à dire les paires $\langle a, b\rangle \in R$ telles que pour tout c, $\langle a, c\rangle \in R$ implique $\langle b, c\rangle \in R^?$ et $\langle c, b\rangle \in R$ implique $\langle c, a\rangle \in R^?$. L'intuition derrière cette notion de lien immédiat est de dire qu'il n'y pas d'éléments "*entre*" a et b dans R quand $\langle a, b\rangle \in R|_{imm}$.

On suppose également un ensemble Loc d'emplacements mémoire et un ensemble Val de valeurs. On utilise comme méta-variables x, y, z pour des emplacements mémoires et v pour des valeurs.

On note `na`, `rlx`, `acq`, `rel`, `acqrel` et `sc` les différents modes d'accès à la mémoire, respectivement les accès non-atomiques, relâchés, *acquire*, *release*, *release-acquire* et SC. La figure 2 montre la façon dont les accès à la mémoire sont ordonnés par $\sqsubset$.

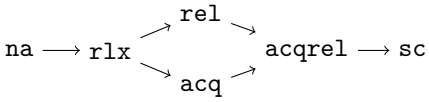

FIGURE 2 – Ordonnancement des modes d'accès par $\sqsubset$

2.3 L'ensemble des exécutions

2.3.1 Évènements

La première composante d'une exécution est un ensemble fini `E` d'évènements. La figure 3 présente la définition en COQ d'un évènement. On peut voir que le type `Event` peut représenter une lecture, une écriture ou une barrière.

```
Inductive Event : Type :=
| Read (eid: nat) (m: Mode) (l: Loc) (v: Val)
| Write (eid: nat) (m: Mode) (l: Loc) (v: Val)
| Fence (eid: nat) (m: Mode).
```

FIGURE 3 – La définition en COQ d'un évènement

Chaque évènement a un identifiant, représenté par un entier naturel et un mode d'accès (non-atomique ou atomique d'un certain type). Les lectures et les écritures sont également

constituées d'un emplacement mémoire et d'une valeur écrite à ou lue de cet emplacement. Dans cette formalisation, les emplacements mémoire et les valeurs sont représentés par des entiers naturels. L'ensemble $\mathtt{E}$ des évènements est bien formé si tous ses éléments ont un identifiant différent, et si les modes d'accès sont cohérents (une écriture ne peut pas avoir pour mode `rel` par exemple). Ces deux conditions sur l'ensemble des évènements sont rassemblées dans la définition `valid_evts`.

On peut noter que cette formalisation est presque identique à la définition d'origine dans le modèle RC11 [7, section 3.1]. La seule différence réside dans l'encodage des identifiants. La définition d'origine représente les évènements d'une exécution par une fonction qui associe à chaque identifiant un label, composé d'un mode d'accès, d'une valeur et d'un emplacement. Cette définition est équivalente à la définition en Coq que l'on a présentée, associée à la condition que chaque évènement a un identifiant différent.

Six fonctions $\mathtt{typ}$, $\mathtt{mod}$, $\mathtt{id}$, $\mathtt{loc}$, $\mathtt{val_r}$ et $\mathtt{val_w}$ permettent d'accéder respectivement au type ($\mathtt{R}$, $\mathtt{W}$ ou $\mathtt{F}$), au mode d'accès, à l'identifiant, à l'emplacement affecté et à la valeur lue ou écrite d'un évènement. Les fonctions $\mathtt{val_r}$, $\mathtt{val_w}$ et $\mathtt{loc}$ sont partielles, car elles ne s'appliquent pas aux barrières. De même, $\mathtt{val_r}$ ne s'applique qu'aux lectures et $\mathtt{val_w}$ qu'aux écritures.

On utilise a, b, c comme méta-variables pour des évènements. Pour chaque type d'évènement $\mathtt{T} \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$, $\mathtt{T}$ désigne également l'ensemble $\{a \in \mathtt{E} \mid \mathtt{typ}(e) = \mathtt{T}\}$. Les ensembles d'évènements peuvent également être annotés par un emplacement en indice et par un exposant pour le mode. Par exemple, $\mathtt{W}_x^{\sqsupseteq \mathtt{rlx}}$ désigne l'ensemble des écritures à l'emplacement x dont le mode est au moins relâché.

L'ensemble $\mathtt{E}$ contient un ensemble d'évènements d'initialisation $\mathtt{E}_0 = \{\mathtt{W}^{\mathtt{na}}(i, x, 0) \mid x \in \mathsf{Loc}\}$ où les identifiants i sont choisis tels que $\mathtt{E}$ est bien formé.

2.3.2 Relations

Une exécution est également constituée de quatre relations entre évènements :

1. $\mathtt{sb}$ (*sequenced-before*) est une relation d'ordre strict (parfois aussi notée $\mathtt{po}$ pour *program order*), telle que $\mathtt{sb} \subseteq \mathtt{E} \times \mathtt{E}$. Cette relation ordonne les évènements d'un même *thread* dans l'ordre dans lequel ils apparaissent dans le programme. Elle doit également ordonner les évènements d'initialisation avant tous les autres évènements (on a $\mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0) \subseteq \mathtt{sb}$). Ces conditions de validité sont rassemblées dans la définition `valid_sb`.
2. $\mathtt{rf}$ (*reads-from*) est une relation binaire qui associe à une écriture a, les lectures qui lisent de la mémoire la valeur écrite par a. Cette relation doit satisfaire quatre conditions :
   (a) $\mathtt{rf} \subseteq [\mathtt{W}]; =_{\mathtt{loc}}; [\mathtt{R}]$.
   (b) Pour tout $\langle a, b\rangle \in \mathtt{rf}$, on a $\mathtt{val_w}(a) = \mathtt{val_r}(b)$.
   (c) $\langle a_1, b\rangle, \langle a_2, b\rangle \in \mathtt{rf}$ implique $a_1 = a_2$.
   (d) $\mathtt{R} \subseteq \mathsf{codom}(\mathtt{rf})$.

   Ces conditions imposent que la relation lie une écriture à une lecture, que ces deux évènements affectent un même emplacement et qu'elles lisent ou écrivent la même valeur. De plus, $\mathtt{rf}$ doit associer une unique écriture à chaque lecture, signifiant qu'une valeur lue en mémoire doit nécessairement y avoir été écrite. Ces conditions de validité sont rassemblées dans la définition `valid_rf`.
3. $\mathtt{mo}$ (*modification order*) est une relation d'ordre strict sur $\mathtt{W}$ (parfois aussi notée $\mathtt{co}$ pour *coherence order*). Elle relie des évènements affectant le même emplacement mémoire. Pour tout $x \in \mathit{Loc}$, la restriction de $\mathtt{mo}$ aux évènements lisant de ou écrivant sur x (notée $\mathtt{mo}_x$) est une relation d'ordre total sur $\mathtt{W}_x$. Cette relation décrit l'ordre dans lequel les effets des écritures en mémoire deviennent visibles à tous les *threads* du programme. Ces conditions de validité sont rassemblées dans la définition `valid_mo`.

4. $\texttt{rmw}$ (*read-modify-write*) est une relation binaire qui permet de traduire les accès *read-modify-write*. Notre formalisation traduit ces opérations en une lecture suivie immédiatement par une écriture dans $\texttt{sb}$. On a $\texttt{rmw} \subseteq [\texttt{R}]; (\texttt{sb}|_{imm} \cap =_{loc}); [\texttt{W}]$. Pour toute paire d'évènements $\langle a, b\rangle \in \texttt{rmw}$, on a :

$$\langle \texttt{mod}(a), \texttt{mod}(b)\rangle \in (\langle \texttt{rlx}, \texttt{rlx}\rangle, \langle \texttt{rel}, \texttt{rlx}\rangle, \langle \texttt{rlx}, \texttt{acq}\rangle, \langle \texttt{rel}, \texttt{acq}\rangle, \langle \texttt{sc}, \texttt{sc}\rangle)$$

On note $\texttt{At}$ le sous-ensemble des évènements de $\texttt{E}$ reliés par $\texttt{rmw}$. Ces conditions de validité sont rassemblées dans la définition `valid_rmw`.

On notera respectivement $G.\texttt{sb}$, $G.\texttt{rf}$, $G.\texttt{mo}$ et $G.\texttt{rmw}$ les relations *sequenced-before*, *reads-from*, *modification order* et *read-modify-write* d'une exécution G. De plus, on note $G.\texttt{rf}|_{\texttt{sc}}$ la relation $([G.\texttt{E}^{\texttt{SC}}]; G.\texttt{rf}; [G.\texttt{E}^{\texttt{SC}}])$. La notation équivalente existe pour $\texttt{sb}$, $\texttt{mo}$ et $\texttt{rmw}$.

Dans cet article, nous considérons uniquement les programmes concurrents qui sont un ensemble fini de programmes séquentiels, exécutés en parallèle chacun sur un *thread*. La construction d'une exécution d'un programme est alors définie inductivement sur la structure des programmes séquentiels qui le composent.

Bien sûr, cette définition dépend du langage source, mais elle suit toujours le même schéma. À l'ensemble des accès mémoire réalisés lors d'une exécution d'un programme, on ajoute une relation $\texttt{sb}$ qui encode l'ordre des instructions ayant généré ces accès dans le programme. Puis on ajoute des relations $\texttt{rf}$, $\texttt{mo}$ et $\texttt{rmw}$ arbitraires, mais respectant les conditions ci-dessus.

On peut retrouver les définitions des évènements et des relations d'une exécution dans le fichier `exec.v` du développement Coq.

2.4 La conformité d'une exécution

La validité d'une exécution se décompose en quatre conditions à vérifier. Pour les exprimer de façon compréhensible, nous allons d'abord introduire d'autres relations, dérivées de celles qui composent une exécution.

2.4.1 Les relations rb et eco

$$\texttt{rb} \triangleq \texttt{rf}^{-1}; \texttt{mo} \qquad (\textit{reads-before})$$

La relation $\texttt{rb}$ (*reads-before*, parfois notée $\texttt{fr}$ pour *from-read*) relie les lectures d'une valeur en mémoire aux écritures qui sont effectuées au même emplacement après cette lecture.

$$\texttt{eco} \triangleq (\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^{+} \qquad (\textit{extended coherence order})$$

La relation $\texttt{eco}$ contient l'ensemble des paires d'évènements telles que l'effet du premier évènement est visible pour le second évènement. En effet, l'effet d'une écriture est visible pour les lectures qui lisent cette valeur, ce qui est exprimé par $\texttt{rf}$. Par définition, $\texttt{mo}$ décrit l'ordre dans lequel les écritures sont visibles les unes pour les autres. Et enfin, $\texttt{rb}$ (*reads-before*) assure que la lecture d'une valeur en mémoire soit effectuée avant que cette valeur ne soit changée par une autre écriture.

2.4.2 La relation hb

La relation $\texttt{hb}$ (*happens-before*) est utilisée pour exprimer la sémantique des atomiques *release-acquire*. Une paire $\langle x, y\rangle \in \texttt{hb}$ signifie que l'effet de l'accès mémoire x doit être visible de l'évènement y, et ce à cause d'une paire d'évènements *release-acquire*. La relation $\texttt{hb}$ est définie à partir de deux relations plus basiques :

$$\texttt{rs} \triangleq [\texttt{W}]; \texttt{sb}|^{?}_{\texttt{loc}}; [\texttt{W}^{\sqsupseteq\texttt{rlx}}]; (\texttt{rf}; \texttt{rmw})^{*} \qquad (\textit{release-sequence})$$

$$\texttt{sw} \triangleq [\texttt{E}^{\sqsupseteq\texttt{rel}}]; ([\texttt{F}]; \texttt{sb})^{?}; \texttt{rs}; \texttt{rf}; [\texttt{R}^{\sqsupseteq\texttt{rlx}}]; (\texttt{sb}; [\texttt{F}])^{?}; [\texttt{E}^{\sqsupseteq\texttt{acq}}] \qquad (\textit{synchronizes-with})$$

$$\texttt{hb} \triangleq (\texttt{sb} \cup \texttt{sw})^{+} \qquad (\textit{happens-before})$$

On peut voir que hb ordonne les paires d'éléments appartenant à sw, c'est à dire les paires d'éléments *release-acquire* qui se *synchronisent* l'un avec l'autre. Il ordonne également tous les évènements qui précèdent le premier élément d'une paire dans sw avec ceux qui suivent le second élément de cette même paire.

$$\underbrace{[\texttt{E}^{\sqsupseteq\texttt{rel}}]}_{①}; \underbrace{([\texttt{F}]; \texttt{sb})^{?}}_{②}; \underbrace{[\texttt{W}]; \texttt{sb}|^{?}_{\texttt{loc}}}_{③}; \underbrace{[\texttt{W}^{\sqsupseteq\texttt{rlx}}]}_{④}; \underbrace{(\texttt{rf}; \texttt{rmw})^{*}}_{⑤}; \underbrace{\texttt{rf}}_{⑥}; \underbrace{[\texttt{R}^{\sqsupseteq\texttt{rlx}}]}_{④}; \underbrace{(\texttt{sb}; [\texttt{F}])^{?}}_{②}; \underbrace{[\texttt{E}^{\sqsupseteq\texttt{acq}}]}_{①}$$

La figure ci-dessus correspond à la définition de sw dans laquelle on a déplié la définition de rs et numéroté les parties importantes de la définition. Dans sa forme la plus simple, la relation sw relie deux évènements *release-acquire* (①) par une relation rf (⑥). Mais il y a des complications : le premier évènement de la paire peut être une écriture *release* (④), une écriture *release* qui précède une écriture atomique au même emplacement (③), ou une barrière *release* qui précède une écriture atomique (②). Le second évènement de la paire peut être une lecture *acquire* (④) ou une barrière *acquire* (②).

Quelle que soit la situation, il existe une paire composée d'une écriture et d'une lecture atomique qui entrent en jeu dans sw (④). La lecture atomique peut lire la valeur écrite par l'écriture atomique (⑥). Elle peut également lire la valeur écrite par le dernier évènement d'une chaîne de *read-modify-write* si le premier évènement de cette chaîne a lu la valeur écrite par l'écriture atomique (⑤).

2.4.3 La relation psc

La relation psc est utilisée pour exprimer la condition des atomiques SC. Intuitivement, on souhaite qu'il y ait un ordre total entre tous les évènements sc, qui correspond à l'ordre dans lequel ils sont exécutés et donc perçus par les autres *threads*.

Cet ordre total ne doit pas contredire ni l'ordre du programme (sb), ni l'ordre dans lequel des variables doivent être visibles les unes par rapport aux autres (eco). L'ordre imposé par hb doit également être pris en compte. Enfin, la sémantique des barrières sc doit être exprimée.

Pour qu'un ordre total qui respecte ces conditions existe, il est suffisant qu'un ordre partiel qui respecte ces conditions et qui soit acyclique existe [3]. Cela est dû au fait qu'un ordre partiel acyclique sur un ensemble peut toujours être étendu en un ordre total sur ce même ensemble.

La relation psc décrit cet ordre partiel sur les évènements sc.

$$\texttt{scb} \triangleq \texttt{sb} \cup (\texttt{sb}|_{\neq\texttt{loc}}; \texttt{hb}; \texttt{sb}|_{\neq\texttt{loc}}) \cup \texttt{hb}|_{\texttt{loc}} \cup \texttt{mo} \cup \texttt{rb}$$

$$\texttt{psc}_{base} \triangleq ([\texttt{E}^{\texttt{sc}}] \cup ([\texttt{F}^{\texttt{sc}}]; \texttt{hb}^{?})); \texttt{scb}; ([\texttt{E}^{\texttt{sc}}] \cup (\texttt{hb}^{?}; [\texttt{F}^{\texttt{sc}}]))$$

La relation $\texttt{psc}_{base}$ est l'équivalent d'un ordre partiel sur les évènements sc qui est cohérente avec (sb ∪ eco) et hb. rf a été exclue de eco, car lorsque deux éléments sc sont reliés par rf, ils le sont également par hb.

La raison pour laquelle on remplace hb par $(\texttt{sb} \cup \texttt{sb}|_{\neq\texttt{loc}}; \texttt{hb}; \texttt{sb}|_{\neq\texttt{loc}} \cup \texttt{hb}|_{\texttt{loc}})$, est qu'imposer que cet ordre partiel respecte strictement l'ordre imposé par hb rend la compilation correcte vers l'architecture Power impossible [7, section 2.1].

Cette relation $\texttt{psc}_{base}$ impose également que les effets d'un évènement qui doit s'exécuter après une barrière `sc` ne doivent pas pouvoir être observés par les évènements dont les effets doivent s'exécuter avant la barrière.

$$\texttt{psc}_{\texttt{F}} \triangleq [\texttt{F}^{\texttt{sc}}]; (\texttt{hb} \cup \texttt{hb}; \texttt{eco}; \texttt{hb}); [\texttt{F}^{\texttt{sc}}]$$

Cependant, la relation $\texttt{psc}_{base}$ n'exprime pas l'intégralité de la sémantique des barrières `sc`. On souhaiterait qu'un programme dans lequel tous les évènements sont séparés par des barrières `sc` soit conforme à SC. Or, l'acyclicité de $\texttt{psc}_{base}$ ne suffit pas à garantir que cette propriété est respectée [7, section 2.2]. La relation $\texttt{psc}_{\texttt{F}}$ représente les contraintes imposées par ces barrières.

$$\texttt{psc} \triangleq \texttt{psc}_{base} \cup \texttt{psc}_{\texttt{F}}$$

La relation `psc` est simplement l'union de ces deux relations et représente l'ensemble des contraintes qu'un ordre total sur les évènements `sc` doit respecter.

2.4.4 Les quatre conditions de validité

Une exécution est conforme au modèle RC11 à quatre conditions :

1. $\texttt{hb}; \texttt{eco}^?$ est irréflexif
2. $\texttt{rmw} \cap (\texttt{rb}; \texttt{mo}) = \varnothing$
3. `psc` est acyclique
4. $\texttt{sb} \cup \texttt{rf}$ est acyclique

La condition 1 garantit que plusieurs comportements seront évités. Tout d'abord, elle garantit que la relation $(\texttt{hb}; \texttt{eco})$ est irréflexive. Cette condition exprime la sémantique des accès *release-acquire*, en excluant les exécutions où les effets de certains évènements sont observés (exprimé par `eco`) dans un sens interdit par une paire d'atomiques *release-acquire* (exprimé par `hb`). Cependant, `hb` n'exprime pas uniquement l'ordre imposé par ces accès atomiques, puisqu'elle contient également `sb`. Le fait que la relation $(\texttt{sb}; \texttt{eco})$ soit irréflexive signifie qu'un évènement ne peut pas observer les effets des évènements de son propre *thread* dans un ordre différent de celui du programme. Enfin, la condition garantit que la relation `hb` est irréflexive.

La condition 2 exprime la sémantique des paires *read-modify-write*. Elle exclut les exécutions dans lesquelles la valeur contenue dans un emplacement mémoire est modifiée entre la lecture d'une paire *read-modify-write* à cet emplacement et son écriture.

La condition 3 impose que l'ordre partiel `psc` sur les évènements `sc` est acyclique, et donc qu'il peut être étendu en un ordre total.

La condition 4 résout radicalement un problème majeur du modèle mémoire du standard C11, à savoir les comportements *out-of-thin-air*.

$$\begin{array}{l} a :=_{\texttt{rlx}} x \\ y :=_{\texttt{rlx}} a \end{array} \;\Big\|\; \begin{array}{l} b :=_{\texttt{rlx}} y \\ x :=_{\texttt{rlx}} b \end{array}$$

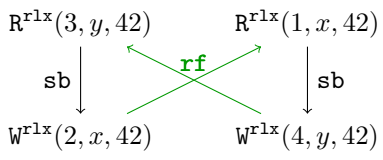

FIGURE 4 – Exécution (à droite) *out-of-thin-air* d'un programme (à gauche)

La figure 4 présente un exemple de programme qui peut donner lieu à une exécution contenant un cycle dans $(\texttt{sb} \cup \texttt{rf})$. Dans ce programme, les valeurs écrites aux emplacements x et y dépendent de valeurs lues des mêmes emplacements. Dans l'exécution associée, ces lectures peuvent lire des valeurs qui viennent "de nulle part", puisqu'il y a une dépendance cyclique entre les 4 accès mémoire de l'exécution. Bien sûr, aucune implémentation du standard C11

ne présente ce genre de comportement. Cependant, formaliser précisément ce qui constitue un comportement *out-of-thin-air* sans interdire des optimisations connues des concepteurs de processeurs et de compilateurs est notoirement difficile [5,6]. Des solutions existent, mais elles sont complexes [9,10]. Imposer l'acyclicité de $(\texttt{sb} \cup \texttt{rf})$ est une solution simple à ce problème, mais qui implique de moins bonnes performances lorsque l'on compile vers certaines architectures (par exemple Power [7, section 5]). On peut retrouver la définition du modèle RC11 dans le fichier `rc11.v` du développement Coq.

3 Vérification du théorème DRF-SC

Maintenant que notre modèle mémoire est rigoureusement défini, nous présentons la preuve qu'il respecte la propriété DRF-SC. Nous souhaitons donc montrer que si toutes les exécutions conformes au modèle SC ne contiennent pas de *races*, alors toutes les exécutions de ce programme qui sont conformes au modèle RC11 sont également conformes au modèle SC.

Trois définitions nous manquent pour que cet énoncé soit précis. D'abord une définition axiomatique du modèle mémoire SC, puis une définition précise de ce qu'est une *race* et enfin une définition de l'ensemble des exécutions associées à un programme. Nous présenterons donc ces trois définitions, suivies de la preuve en elle-même.

3.1 Le modèle SC

Dans le modèle SC, on souhaite que tous les accès à la mémoire, qu'ils soient atomiques ou non, soient observés dans l'ordre dans lequel ils apparaissent dans le programme. On souhaite donc que la relation $\texttt{eco}$ ne soit jamais en contradiction avec $\texttt{sb}$. On souhaite également que l'atomicité des opérations *read-modify-write* soit préservée. On caractérise donc le modèle SC par ces deux conditions :

1. $\texttt{rmw} \cap (\texttt{rb}; \texttt{mo}) = \varnothing$
2. $(\texttt{sb} \cup \texttt{eco})$ est acyclique.

On peut retrouver la définition de la conformité à SC dans le développement Coq. On peut également noter deux propriétés importantes relatives aux modèles SC et RC11 :

Lemme 1 (). *Toutes les exécutions conformes au modèle SC sont également conformes à RC11*

Cette propriété nous assure que RC11 n'exclut aucun comportement SC, et ne fait qu'en admettre des nouveaux. Cela garantit que le modèle est "réaliste", et qu'il n'exclut pas de comportements auxquels un programmeur pourrait légitimement s'attendre.

Lemme 2 (). *Si tous les accès d'un programme sont des atomiques de type* `sc`*, toutes ses exécutions conformes à RC11 sont également conformes à SC*

3.2 Les *races*

```
Definition race (ex: Execution) : rlt Event :=
  fun x => fun y =>
    (is_write x \/ is_write y) /\
    x <> y /\ get_loc x = get_loc y /\
    ~((hb ex) x y) /\ ~((hb ex) y x).
```

FIGURE 5 – La définition en Coq d'une *race*

Jusqu'à présent, nous avons décrit deux évènements formant une race comme deux accès au même emplacement mémoire, pouvant être exécutés simultanément, et dont au moins un est une écriture.

Nous appelons *évènements en conflit* les paires distinctes d'accès à un emplacement mémoire dont au moins un est une écriture. Comment peut-on exprimer que deux de ces éléments peuvent s'exécuter simultanément ?

Une autre façon de dire que deux instructions peuvent s'exécuter simultanément est de dire que le modèle mémoire n'impose pas que les effets d'une des instructions doivent être visibles pour l'autre accès sur la mémoire, et donc n'impose pas que leurs effets affectent la mémoire dans un ordre précis. Dans le cas de RC11, cet ordre peut être imposé grâce aux atomiques, dans deux cas :

1. lorsque une première instruction est reliée à une seconde par `hb`. Les effets de la première instruction sur la mémoire doivent alors être visibles pour la seconde.
2. lorsque deux évènements sont `sc`, l'acyclicité de `psc` impose que leurs effets soient vus dans le même ordre par tous les *threads* du programme.

Les évènements x et y forment donc une race, ce qu'on note $\langle x, y\rangle \in$ `race`, lorsque x et y sont en conflit, qu'ils ne sont pas reliés par `hb` (ni dans un sens, ni dans l'autre), et que l'un d'entre eux n'est pas un atomique `sc`.

On peut remarquer dans la figure 5 que la définition de *race* de notre formalisation Coq n'est pas exactement la même, puisqu'elle n'exclut pas les paires d'évènements `sc`. Dans l'article original, les auteurs du modèle RC11 définissent une *race* de cette façon, et formulent la garantie DRF-SC de façon légèrement différente de ce que nous avons présenté jusqu'à présent. Le théorème tels qu'ils le formulent [7, théorème 4, section 8], stipule que si dans toutes les exécutions conformes à SC, les évènements qui forment une *race* sont tous les deux `sc`, alors toutes les exécutions du programme conformes à RC11 sont conformes à SC.

Pour que la formalisation en Coq de la preuve de la garantie DRF-SC soit la plus directe possible, nous avons conservé la formulation initiale dans la formalisation Coq, comme on peut le voir dans les hypothèses dans le lemme final `drf_sc_final`.

Ajouter le fait que deux évènements `sc` ne peuvent pas former une *race* directement dans la définition d'une *race* ou le préciser dans les hypothèses de notre théorème final sont deux façons équivalentes de prouver la même chose. Nous pensons que la première façon expose plus clairement la signification de ce qu'est une *race*, et c'est pourquoi nous avons introduit la notion de cette façon dans cet article.

3.3 L'ensemble des exécutions

Comme nous l'avons mentionné dans la partie 2.3.2, on ne dispose pas d'une façon unique de transformer un programme en un ensemble d'exécutions, puisque cette transformation dépend du langage source que l'on considère. Cela pose un problème lorsque l'on veut formuler notre théorème DRF-SC, puisqu'il nous faut considérer l'ensemble des exécutions conformes à SC et RC11.

Pour contourner cette difficulté, nous définissons de façon axiomatique ce que signifie pour deux exécutions le fait d'être issues d'un même programme. Nous expliquerons pourquoi ces axiomes sont raisonnables et correspondent à une idée intuitive de ce que sont deux exécutions issues d'un même programme.

3.3.1 Préfixes d'exécution

Une exécution G' est un préfixe d'une autre exécution G lorsque l'ensemble des évènements de G' est un sous-ensemble $\mathtt{E}$ des évènements de G tel que :

— Les évènements de G' contiennent l'ensemble $\mathtt{E}_0$ des évènements d'initialisation de G.

— Pour tout évènement $b \in G'.\mathtt{E}$, si un évènement a est tel que $\langle a, b\rangle \in G.(\mathtt{sb} \cup \mathtt{rf})$, alors $a \in G'.\mathtt{E}$. $G'.\mathtt{E}$ est donc clos par rapport à $G'.(\mathtt{sb} \cup \mathtt{rf})^{-1}$.

Intuitivement, le préfixe d'une exécution correspond à une exécution incomplète. Pour chaque *thread*, on choisit un des évènements qu'il contient et on ne garde que les évènements qui le précèdent dans le programme. Le sous-ensemble choisi n'est pas clos seulement par rapport à $\mathtt{sb}^{-1}$, mais aussi par rapport à $\mathtt{rf}^{-1}$ pour éviter que les choix de découpage pour chaque *thread* soient incohérents entre eux, et que certaines lectures lisent en mémoire des valeurs qui n'y ont jamais été écrites.

Dans le cadre de la preuve de la propriété DRF-SC, nous considérons une formulation différente mais équivalente de la notion de préfixe [1]. On considère qu'une exécution est *cohérente* si les identifiants uniques associés à chacun des évènements de l'exécution sont tels que pour toute paire d'évènements $\langle x, y\rangle \in (\mathtt{sb} \cup \mathtt{rf})$, on a $\mathtt{id}(x) \leq \mathtt{id}(y)$.

Lorsque G est une exécution cohérente, G' est une *délimitation* de G par un entier naturel k si l'ensemble des évènements de G' est un sous-ensemble $\mathtt{E}$ des évènements de G tel que :

— Les évènements de G' contiennent l'ensemble $\mathtt{E}_0$ des évènements d'initialisation de G.

— Pour tout évènement $x \in G'.\mathtt{E}$, $\mathtt{id}(x) \leq k$.

Pour toute exécution cohérente et quel que soit l'entier k, une délimitation de l'exécution G par k, notée G_k est un préfixe de l'exécution. Il est à noter que pour toute exécution conforme à RC11 (ou SC par extension), il existe des identifiants qui font de l'exécution une exécution cohérente, et ce grâce à la condition qui interdit les comportements *out-of-thin-air* dans RC11. L'intérêt de cette définition alternative est qu'elle permet de comparer les préfixes d'exécutions. Pour j, k tels que $j \leq k$, G_j est un plus petit préfixe de G que G_k.

Pour comprendre pourquoi cet axiome est raisonnable, il est utile d'anticiper un tout petit peu sur la preuve en elle-même. Notre raisonnement consistera à montrer la contraposée de notre énoncé de DRF-SC, c'est à dire que si un énoncé est conforme à RC11 mais pas à SC, il existe une autre exécution du même programme, conforme à SC et qui contient une *race*. Dans ce cadre, il n'est pas nécessaire de prouver qu'il existe une autre exécution "complète" du programme. Puisque l'on cherche à montrer que cette autre exécution est conforme à SC et contient une *race*, il suffit de montrer qu'un préfixe d'exécution respectant ces conditions existe. En effet, si une exécution incomplète d'un programme conforme à SC contient une *race*, on peut imaginer qu'il suffit de continuer à exécuter les instructions du programme séquentiellement, pour obtenir une autre exécution complète, qui sera toujours conforme à SC, et dans laquelle la *race* sera toujours présente.

3.3.2 Changement d'une lecture sb-finale

Soit G une exécution qui contient une lecture $a \notin \mathsf{dom}(G.\mathtt{sb})$ (a est le dernier évènement de son propre *thread*). L'évènement a lit une valeur v de l'emplacement mémoire x. Cette valeur a été écrite à cet emplacement par une écriture b. Supposons maintenant qu'il existe dans G une écriture c qui écrit la valeur v' à l'emplacement x. Nous pouvons construire l'exécution G' dans laquelle la valeur lue par a est changée pour v', et où la paire $\langle b, a\rangle$ de $G.\mathtt{rf}$ est remplacée par une paire $\langle c, a\rangle$.

Nous considérons alors que G et G' sont deux exécutions d'un même programme. Considérer que ce changement résulte en une autre exécution d'un même programme est raisonnable, car la valeur lue par une lecture ne peut pas avoir une influence sur ce qui s'est passé avant la lecture dans le *thread*. Il est possible de modéliser précisément les évènements suivant la lecture qui peuvent être modifiés, grâce une relation de *dépendance*. Mais dans notre cas ce n'est pas nécessaire, car aucun évènement ne suit la lecture.

1. Pour la preuve de cette équivalence, voir le lemme Coq `bounded_exec_is_prefix`.

3.3.3 Égalité modulo mo

Si deux exécutions sont identiques à l'exception de leur relation mo, on considère que ce sont deux exécutions d'un même programme.

Cet axiome est également conforme à l'intuition que l'on peut avoir de deux exécutions d'un même programme. En effet, lorsqu'on ne change ni les instructions d'un programme, ni les valeurs qui sont lues ou écrites par ces instructions, le comportement du programme reste le même. Lorsque l'on change la relation mo, et donc l'ordre dans lequel sont perçues les écritures, seule la conformité de l'exécution avec un modèle mémoire, et donc la possibilité de l'existence d'une telle exécution, peut changer.

3.3.4 Formalisation en Coq

```
Inductive sameP (res ex: Execution) : Prop :=
  | sameP_pre : prefix res ex -> sameP res ex
  | sameP_res_chval : forall j k bound c v l,
      minimal_conflicting_pair ex bound j k ->
      numbering k > numbering j ->
      res_chval_k ex res bound j k c l v -> sameP res ex
  | sameP_mo : eq_mod_mo res ex -> sameP res ex
  | sameP_trans : forall c, sameP res c -> sameP c ex -> sameP res ex.
```

FIGURE 6 – La définition en Coq de deux exécutions d'un même programme

Les conditions dans lesquelles deux exécutions sont issues d'un même programme sont rassemblées dans un prédicat inductif `sameP` présenté dans la figure 6. Les conditions nécessaires pour que le changement d'une lecture `sb`-finale soit possible sont plus restreintes que ce que nous avons présenté dans la section 3.3.2.

Ces conditions ne prennent en compte que le changement de lectures `sb`-finales dans un cas particulier, qui est le cas précis qui se présentera dans la preuve. L'important est de noter que ce cas particulier implique que la lecture dont la valeur lue est modifiée est bien `sb`-finale dans l'exécution.

3.4 La preuve

```
Lemma drf_sc (e: Execution):
  complete_exec e /\ numbering_coherent e /\ numbering_injective e ->
  (forall e', sameP e' e /\ sc_consistent e' ->
              (forall a b, (race e') a b ->
                           (get_mode a = Sc /\ get_mode b = Sc))) ->
  (forall e', sameP e' e /\ rc11_consistent e' ->
              sc_consistent e').
```

FIGURE 7 – Énoncé final du théorème DRF-SC

Le principe général de la preuve est le suivant : nous allons prouver la contraposée de notre énoncé initial, tel qu'il est présenté dans la figure 7. Nous supposons une exécution G conforme à RC11, mais pas à SC, et nous cherchons à prouver qu'il existe une autre exécution du même programme qui est elle conforme à SC, mais qui contient une race (ou dont les éléments formant une *race* sont tous les deux `sc`. Voir 3.2).

Pour présenter cette preuve de façon claire et permettre au lecteur d'accéder aux lemmes importants du développement Coq, la figure 8 présente le raisonnement sous forme d'arbre.

Les nœuds de cet arbre représentent les différentes étapes de la preuve, et seront annotés par des liens vers les lemmes importants concernant l'exécution que l'on considère à cette étape de la preuve. Les flèches pleines représente un passage d'une exécution à une autre en restant dans les exécutions d'un même programme. Les flèches en pointillés représentent des embranchements dans le raisonnement lorsque l'on considère différents cas.

Aux feuilles de cet arbre, on trouvera les lemmes finaux, c'est à dire ceux qui prouvent que les exécutions que nous avons obtenues dans chaque cas sont conformes à SC et contiennent une *race*.

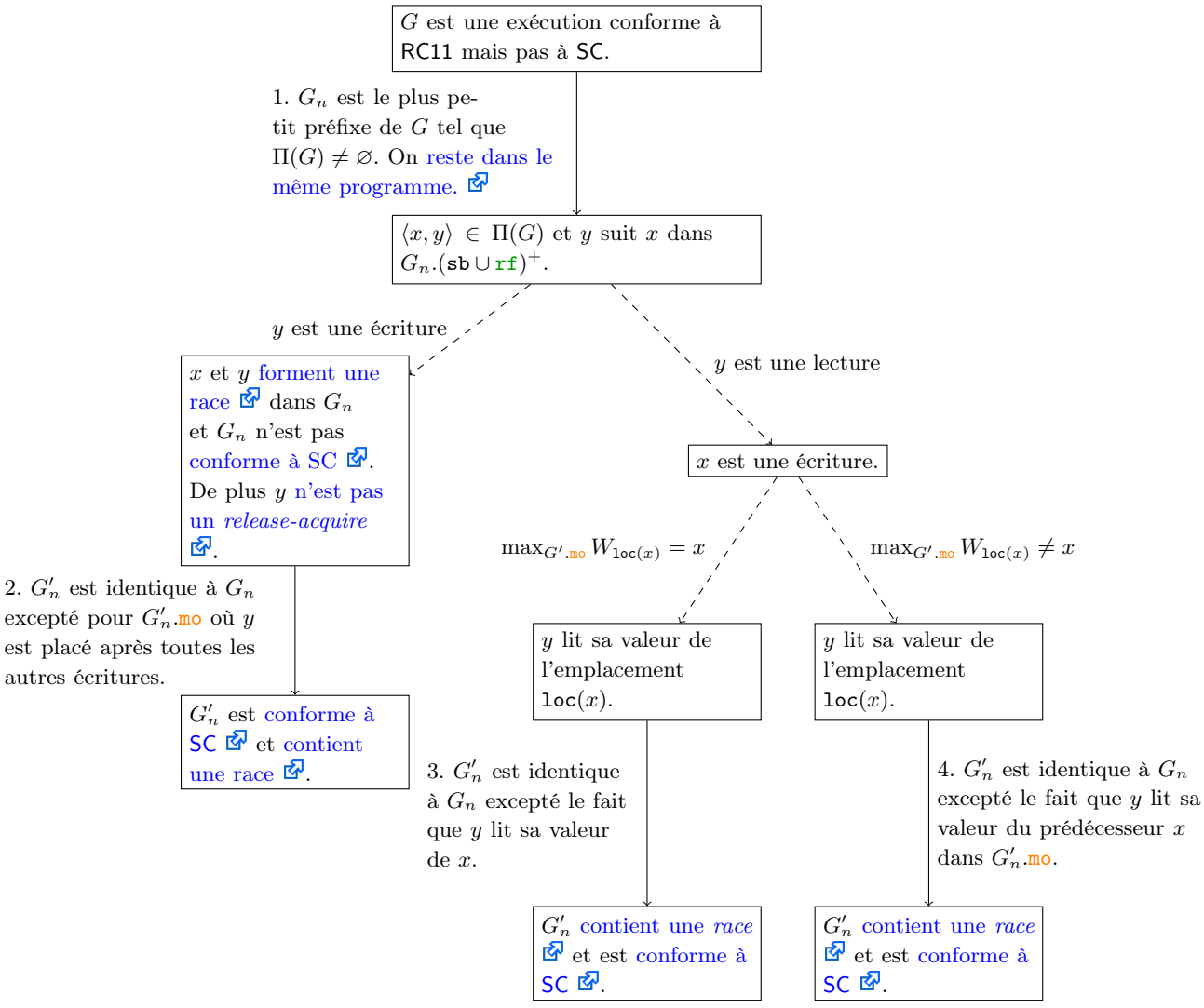

FIGURE 8 – Schéma résumant les différentes étapes du raisonnement de la preuve DRF-SC pour RC11

Avant d'exposer le raisonnement de la preuve un peu plus en détails, nous introduisons $\Pi(G)$ l'ensemble des paires d'évènements $\langle a, b\rangle$ telles que :

— a et b sont deux évènements de l'exécution ($a \in G.\mathtt{E}$ et $b \in G.\mathtt{E}$),

— a et b sont en conflit,

— a et b ne sont pas tous les deux des atomiques `sc`, et

— ni $\langle a, b\rangle$, ni $\langle b, a\rangle$ n'appartiennent à $(G.\mathtt{sb} \cup G.\mathtt{rf}|_{\mathtt{sc}})^+$.

Pour tout $i \in \mathbb{N}$ tel que G_i est une délimitation de G, si $\Pi(G_i) = \varnothing$, alors G_i est conforme à SC.

On suppose donc une exécution G conforme à RC11 mais pas à SC. Puisque G n'est pas conforme à SC, on a $\Pi(G) = \varnothing$. Il existe alors G_n, la plus petite délimitation de G (pour tout $i < n$, on a $\Pi(G_i) = \varnothing$), dont les évènements incluent une paire d'évènements $\langle x, y\rangle \in \Pi(G_n)$ telle que $\mathtt{id}(y) = n$. Cela correspond à l'étape 1 présentée dans la figure 8.

Puisque tous les préfixes de G_n sont conformes à SC, et que G_n respecte la condition d'atomicité des *read-modify-write*[2], G_n n'est pas conforme à SC à cause d'un cycle dans $G_n.(\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})$ dont y fait partie. On considère alors deux cas :

1. Si y est une écriture, on a x et y qui forment une *race* dans G_n. De plus, l'écriture y n'étant suivie par aucun évènement dans $G_n.\mathtt{sb}$, l'étape qui la suit dans le cycle ne peut faire partie que de la relation $G_n.\mathtt{mo}$. On modifie donc la relation $G_n.\mathtt{mo}$ pour obtenir l'exécution G'_n dans laquelle y n'est suivie par aucune écriture dans $G'_n.\mathtt{mo}$. On obtient alors une exécution conforme à SC et contenant une *race*. Cela correspond à l'étape 2 présentée dans la figure 8. Il est également important de noter que y ne pouvait pas être une écriture atomique, et que donc l'atomicité des *read-modify-write* n'est pas affectée par notre modification de la relation $\mathtt{mo}$.
2. Si y est une lecture, on a forcément x qui est une écriture. Puisque y est une lecture qui n'est suivie par aucun évènement dans $(G_n.\mathtt{sb} \cup G_n.\mathtt{rf})$, l'étape qui la suit dans le cycle ne peut faire partie que de la relation $G_n.\mathtt{rb}$. Cela signifie que y lit sa valeur depuis une lecture qui est suivie par un autre évènement dans $G_n.\mathtt{mo}$.
   On considère alors deux cas :
   (a) $\max_{G'.\mathtt{mo}} W_{\mathtt{loc}(x)} = x$. Dans ce cas, on prend le prédécesseur immédiat d de x dans $G_n.\mathtt{mo}$ est on remplacer la valeur lue par y par celle de d pour obtenir l'exécution G'_n. Cela correspond à l'étape 3 présentée dans la figure 8. Or, par définition, x n'est suivi par aucun évènement dans $G'_n.\mathtt{mo}$ et de plus il n'est suivi par aucun autre évènement dans $(G'_n.\mathtt{sb} \cup G'_n.\mathtt{rf})$. Le cycle est donc brisé est on obtient une exécution conforme à SC et contenant une *race*. Nous avons pris le prédécesseur de x car créer un lien dans $G'_n.\mathtt{rf}$ entre x et y, aurait brisé la *race* entre ces deux évènements.
   (b) $\max_{G'.\mathtt{mo}} W_{\mathtt{loc}(x)} \neq x$. Dans ce cas, on remplace la valeur lue par y par celle de $\max_{G'.\mathtt{mo}} W_{\mathtt{loc}(x)}$. Cela correspond à l'étape 4 présentée dans la figure 8. Puisque cette écriture n'est suivie par aucun évènement dans $G'_n.\mathtt{mo}$, le cycle est brisé et on obtient une exécution conforme à SC et contenant une *race*.

4 Limitation et généralisation

En dehors d'apporter un degré de confiance plus élevé, la mécanisation de cette preuve peut avoir selon nous deux intérêts majeurs.

Premièrement, nous pensons que c'est un bon point de départ pour repousser une limitation du théorème DRF-SC. Nous présentons cette limite, qui concerne un idiome de programmation courant, et nous posons quelques pistes pour étendre l'application de cette propriété à cet idiome.

Deuxièmement, nous voudrions abstraire cette preuve pour qu'elle ne dépende plus de détails du modèle RC11, mais seulement de propriétés, qui pourraient éventuellement être partagées par d'autres modèles mémoire.

2. Le préfixe conserve la propriété d'atomicité des exécutions et G respecte l'atomicité car conforme à RC11.

4.1 DRF-SC et le *message-passing*

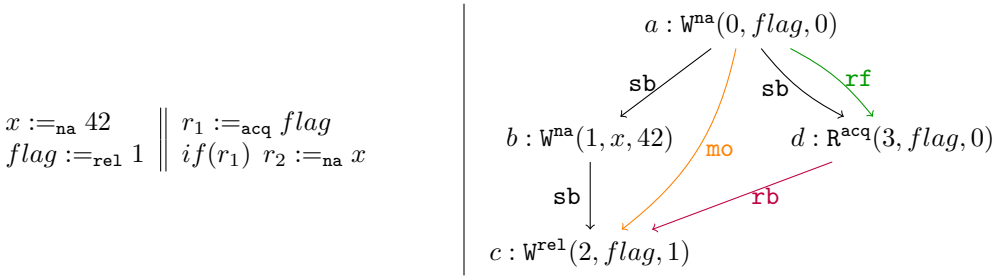

FIGURE 9 – Un programme de *message-passing* et son exécution contenant une *race*

Nous entendons par limitation l'existence de programmes qui n'ont que des comportements SC, mais sur lesquels la propriété DRF-SC ne s'applique pas. La figure 9 est un exemple d'un tel programme. Dans ce programme, on utilise un flag auquel on accède avec des lectures et écritures *release-acquire*, pour protéger l'accès à des données (ici le contenu de x). Toutes les exécutions de ce programme qui sont conformes à RC11 le sont également avec SC.

Mais l'exécution présentée, qui est conforme à SC, contient une *race* entre l'écriture du flag (c) et sa lecture dans le second *thread* (d). Cette *race* ne peut pas donner lieu à un comportement non conforme à SC car le *modification-order* mo est imposé dans cette exécution. La *race* pourrait donner lieu à un comportement non SC si l'écriture (c) pouvait être perçue comme affectant la mémoire avant l'écriture initiale (a). Or cela est impossible, car cela créerait un cycle dans ($\mathtt{hb}; \mathtt{eco}^?$), et donc l'exécution ne serait pas conforme à RC11. Il existe donc un ordre de perception des écritures par les autres *threads* imposé par RC11 et qui n'est pas pris en compte dans la notion de *race*.

Peut-on restreindre la définition de *race* pour que les situations de ce type ne soient pas considérées comme une *race* tout en conservant la propriété DRF-SC ? Nous pensons que la formalisation en Coq que nous venons de présenter sera un atout pour répondre à cette question, car elle permettra de faire évoluer la notion de *race* et la preuve de DRF-SC associée au modèle en parallèle, en étant sûr que cette preuve reste valide.

4.2 Généralisation

Nous pensons que cette preuve mécanisée pourrait être adaptée à d'autres modèles. Pour cela, nous pensons qu'il serait efficace d'encapsuler chaque modèle mémoire axiomatique dans une *typeclass*. Ce procédé permettrait de s'abstraire des différents types d'évènements que le modèle mémoire doit prendre en compte, ainsi que de l'implémentation de relations dérivées des relations de base sb, rf, mo et rmw.

En étudiant la preuve de la propriété DRF-SC pour le modèle RC11, nous pouvons constater plusieurs difficultés :

— la preuve dépend beaucoup de la présence d'atomiques sc dans le modèle RC11. L'ensemble $\Pi(G)$ notamment, est construit à partir de la relation $\mathtt{rf}|_{\mathtt{sc}}$. Pour pouvoir s'en abstraire, nous devons définir l'ensemble des évènements sc, comme un ensemble d'évènements partageant une certaine propriété.

— la preuve dépend également beaucoup des quatre conditions de conformité du modèle RC11. On ne peut donc utiliser la preuve que pour des modèles strictement plus forts que RC11, dont les conditions de conformité impliquent celles de RC11. Or, il existe peut-être des modèles plus relâchés, qui satisfont également la propriété DRF-SC. L'objectif est donc de trouver le modèle le plus relâché possible qui respecte la propriété DRF-SC.

Nous pensons que la première étape de cet effort d'abstraction, avant d'essayer d'adapter la preuve à un autre modèle, est de reformuler le modèle RC11, pour que la preuve se repose moins sur ces détails. Nous pensons que l'assistant de preuves sera un avantage pour garantir que cette reformulation du modèle est équivalente à celle que nous avons présentée.

5 Conclusion

Nous avons présenté dans cet article la formalisation d'un modèle, défini dans [7], qui décrit le comportement des programmes C/C++11 concurrents. Nous avons ensuite présenté la formalisation dans l'assistant de preuve Coq d'une preuve existante et issue du même article, stipulant que ce modèle respecte la propriété DRF-SC. Cette propriété permet aux programmeurs d'ignorer complètement les détails du modèle mémoire, qui peuvent être complexes et contre-intuitifs. Nous avons formalisé cette preuve dans l'assistant de preuves Coq.

La formalisation dans des assistants de preuves de modèles mémoire est un domaine actif, et des modèles mémoires proches de celui de C11 ont récemment été formalisés en Coq [4,12]. Cependant, ces deux modèles vérifient une variante plus «faible» de la propriété DRF-SC. Cette variante stipule que si toutes les exécutions d'un programme conformes à un modèle qui respecte DRF-SC "faible" ne contiennent pas de *races*, alors toutes les exécutions de ce programme sont conformes à SC. Cette propriété est moins intéressante, car elle demande au programmeur de raisonner sur toutes exécutions conformes au modèle mémoire, et pas seulement sur celles conformes à SC pour y détecter les éventuelles *races*.

Le développement Coq final comptabilise au final environ 8000 lignes de code. Nous signalons l'utilisation intensive de la librairie `relation-algebra` [11] au sein de ce développement. Cette librairie fournit une tactique `kat`, qui est une procédure de décision sur les énoncés d'inclusion (et d'équivalence par extension) entre relations (sous certaines conditions). Cette librairie s'est avérée être un outil particulièrement précieux pour travailler sur les modèles mémoires axiomatiques.

Si la preuve du fait que RC11 respecte cette propriété DRF-SC apparaissait déjà dans l'article présentant ce modèle [7], sa mécanisation n'a pas été un simple travail de traduction. Certaines parties non triviales du raisonnement n'étaient pas explicitées dans cette preuve. La preuve du lemme `change_val_eco_ac2` par exemple, nécessite de prendre en compte de très nombreux cas, mais n'est pas donnée dans la preuve non-mécanisée. Pour ces raisonnements non détaillés, la sûreté apportée par un assistant de preuves est d'autant plus importante.

En outre, nous pensons que cette mécanisation de la preuve sera un outil précieux dans la réalisation de deux objectifs : étendre cette propriété pour qu'elle s'applique à plus de programmes et abstraire cette preuve de RC11 pour qu'elle s'applique à des modèles différents. Cette preuve mécanisée pourrait également être utilisée dans le cadre d'une autre preuve formelle. Par exemple, il serait possible de vérifier la spécification d'un programme C11 en se basant sur une sémantique SC si on prouve que ses exécutions ne contiennent aucune *races* et d'obtenir par extension une preuve de cette spécification dans le modèle mémoire RC11.

Références

[1] Jade Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats : Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2) :1–74, 2014.

[3] Mark Batty, Alastair F Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 634–648, 2016.

[4] John Bender and Jens Palsberg. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA) :1–28, 2019.

[5] Hans-J. Boehm. P1217r0 : Out-of-thin-air, revisited, again, 2018.

[6] Hans-J Boehm and Brian Demsky. Outlawing ghosts : Avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–6, 2014.

[7] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++ 11. *ACM SIGPLAN Notices*, 52(6) :618–632, 2017.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9) :690–691, 1979.

[9] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0 : global optimizations in relaxed memory concurrency. In *PLDI*, pages 362–376, 2020.

[10] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *ACM SIGPLAN Notices*, 51(1) :622–633, 2016.

[11] Damien Pous. Kleene algebra with tests and Coq tools for while programs. In *International Conference on Interactive Theorem Proving*, pages 180–196. Springer, 2013.

[12] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. Repairing and mechanising the JavaScript relaxed memory model. *arXiv preprint arXiv :2005.10554*, 2020.