Preuve formelle de la propriété DRF-SC dans le modèle RC11

Quentin Ladeveze

Équipe Cambium
Inria Paris

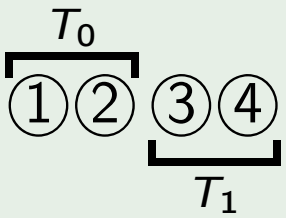
29 mars 2021

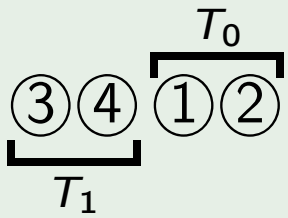
Consistance séquentielle et mémoire faible

Quel comportement attend-t-on d'un programme concurrent ?
↪ Séquentiellement Consistant **SC** : entrelacement des instructions.

Exemple : double écriture et double lecture sur x86

T_0	T_1
`1 : x = 1;`	`3 : y = 1;`
`2 : print(y);`	`4 : print(x);`





y=0 et x=1

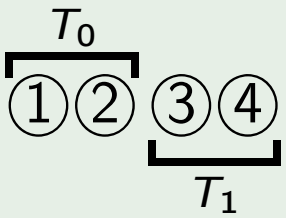
y=1 et x=0

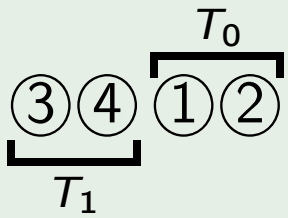
Consistance séquentielle et mémoire faible

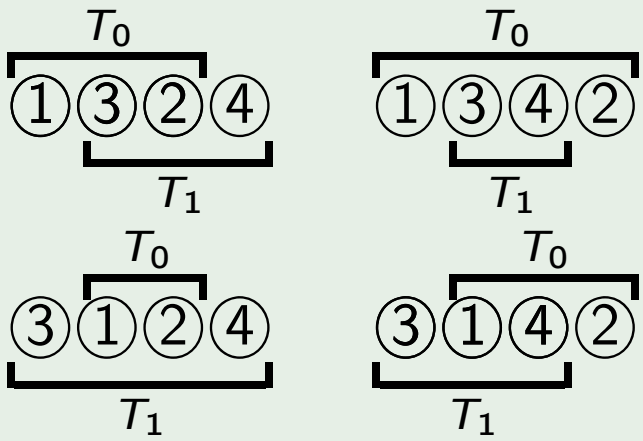
Quel comportement attend-t-on d'un programme concurrent ?
$\hookrightarrow$ Séquentiellement Consistant **SC** : entrelacement des instructions.

Exemple : double écriture et double lecture sur x86

T_0	T_1
`1 : x = 1;`	`3 : y = 1;`
`2 : print(y);`	`4 : print(x);`







y=0 et x=1

y=1 et x=0

y=x=1

Consistance séquentielle et mémoire faible

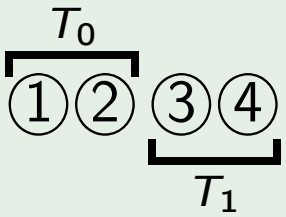
Quel comportement attend-t-on d'un programme concurrent ?
↪ Séquentiellement Consistant **SC** : entrelacement des instructions.

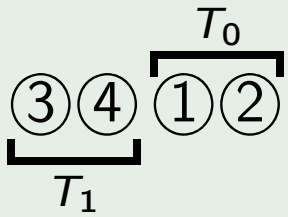
Exemple : double écriture et double lecture sur x86

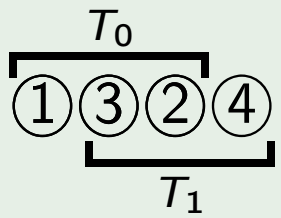
T_0	T_1
`1 : x = 1;`	`3 : y = 1;`
`2 : print(y);`	`4 : print(x);`

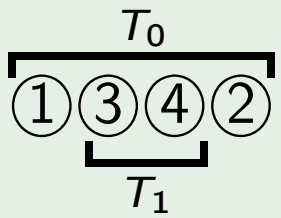
Résultat observé

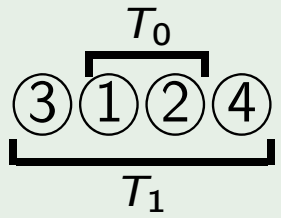
y = x= 0

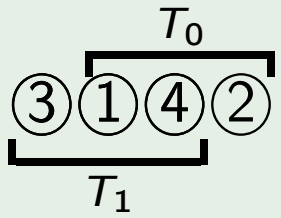












y=0 et x=1

y=1 et x=0

y=x=1

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

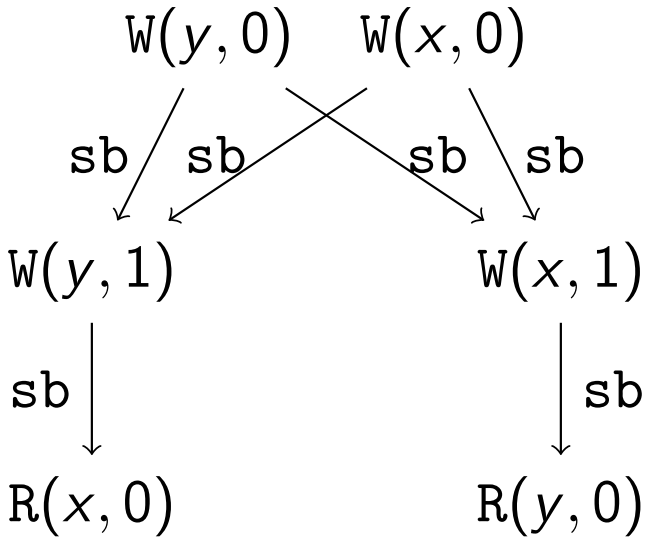
$x = y = 0$

On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$x = y = 0$



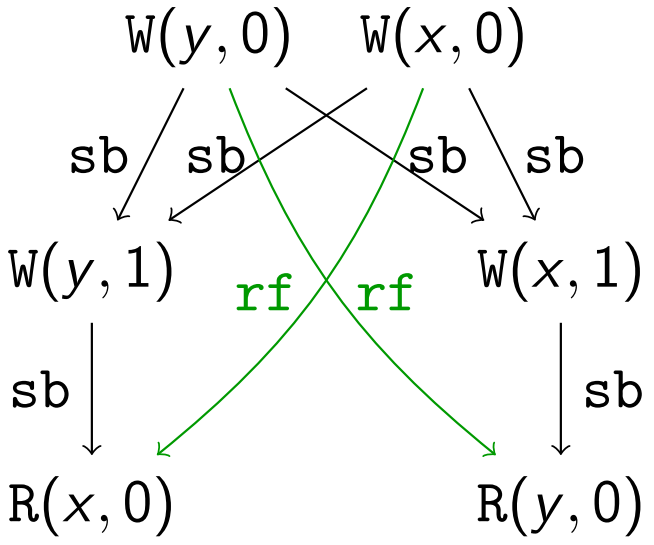
On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

1. **Évènements mémoire** : lectures (`R`), écritures (`W`) et leur ordre dans le programme `sb`.

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$x = y = 0$



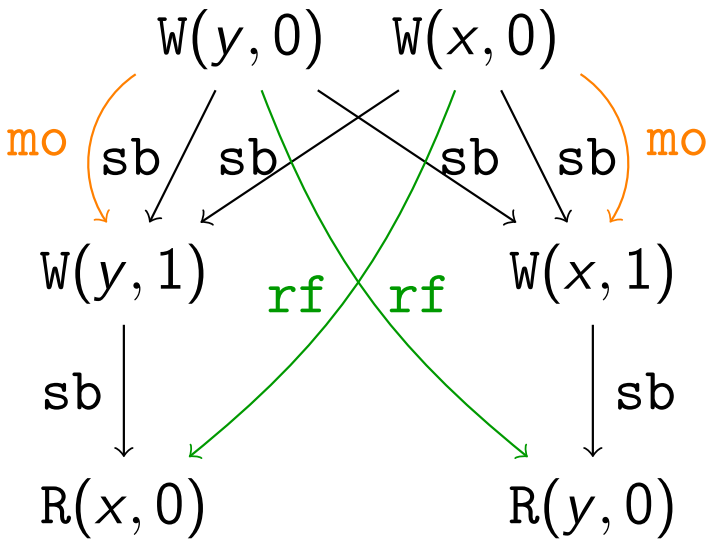
On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

1. **Évènements mémoire** : lectures (`R`), écritures (`W`) et leur ordre dans le programme `sb`.
2. `rf` (*reads-from*) : quand est-ce qu'une valeur lue a été écrite

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$x = y = 0$



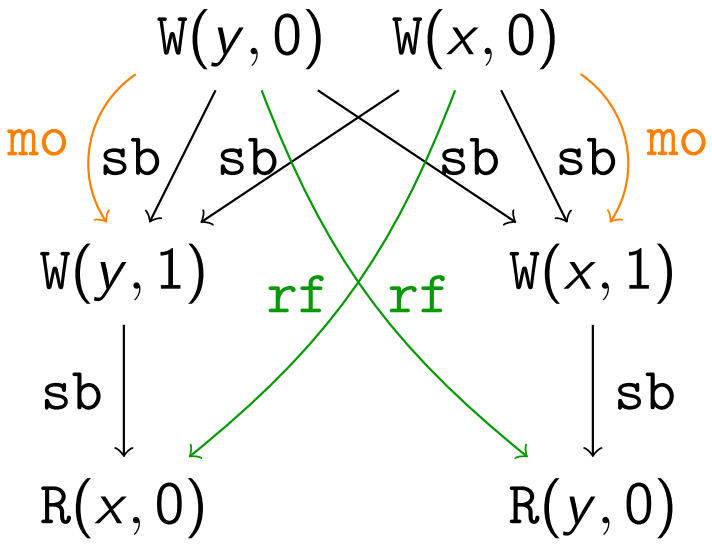
On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

1. **Évènements mémoire** : lectures (`R`), écritures (`W`) et leur ordre dans le programme `sb`.
2. `rf` (*reads-from*) : quand est-ce qu'une valeur lue a été écrite
3. `mo` (*modification-order*) : dans quel ordre les écritures ont affecté la mémoire partagée

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$$x = y = 0$$



On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

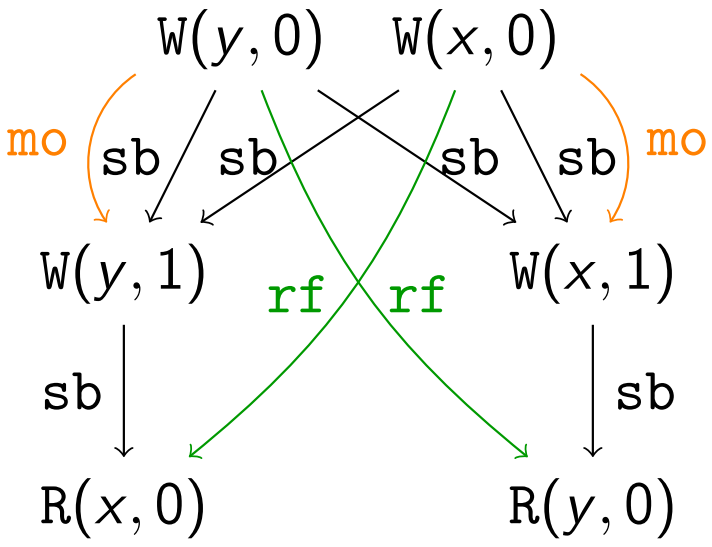
1. **Évènements mémoire** : lectures (`R`), écritures (`W`) et leur ordre dans le programme `sb`.
2. `rf` (*reads-from*) : quand est-ce qu'une valeur lue a été écrite
3. `mo` (*modification-order*) : dans quel ordre les écritures ont affecté la mémoire partagée
4. `rb` (*reads-before*) : quand est-ce qu'une valeur lue est écrasée par une nouvelle écriture

$$\texttt{rb} =$$

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$$x = y = 0$$



On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

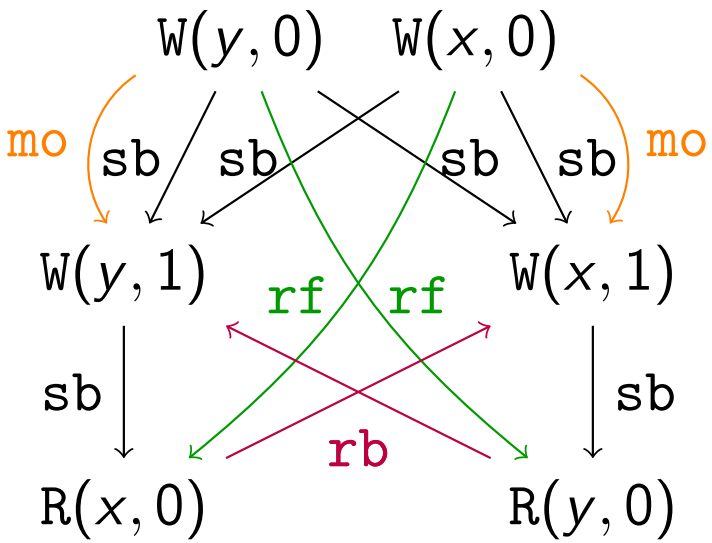
1. **Évènements mémoire** : lectures (`R`), écritures (`W`) et leur ordre dans le programme `sb`.
2. `rf` (*reads-from*) : quand est-ce qu'une valeur lue a été écrite
3. `mo` (*modification-order*) : dans quel ordre les écritures ont affecté la mémoire partagée
4. `rb` (*reads-before*) : quand est-ce qu'une valeur lue est écrasée par une nouvelle écriture

$$\mathtt{rb} = \mathtt{rf}^{-1}$$

Représenter une exécution

T_0	T_1
`x = 1;`	`y = 1;`
`print(y);`	`print(x);`

$$x = y = 0$$



On veut une abstraction des interactions avec la mémoire partagée dans une exécution :

1. **Évènements mémoire** : lectures (R), écritures (W) et leur ordre dans le programme sb.
2. rf (*reads-from*) : quand est-ce qu'une valeur lue a été écrite
3. mo (*modification-order*) : dans quel ordre les écritures ont affecté la mémoire partagée
4. rb (*reads-before*) : quand est-ce qu'une valeur lue est écrasée par une nouvelle écriture

$$\mathtt{rb} = \mathtt{rf}^{-1}; \mathtt{mo}$$

Définir les exécutions possibles

Modèle mémoire

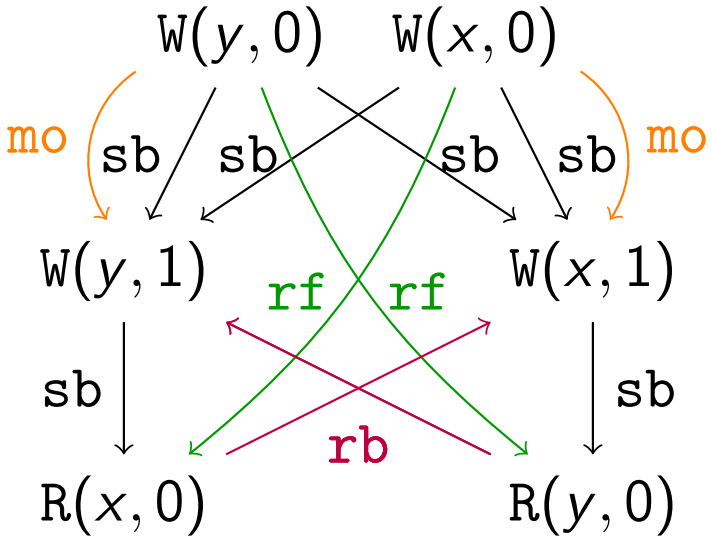
Un modèle mémoire définit quelles exécutions sont *possibles* en définissant des contraintes sur les relations.

Exemple : le modèle mémoire SC

1. `rf`, `mo` et `rb` : relations de *communication* qui imposent un ordre dans l'exécution des évènements.
2. Dans SC, on veut que ces relations soient cohérentes avec `sb`.

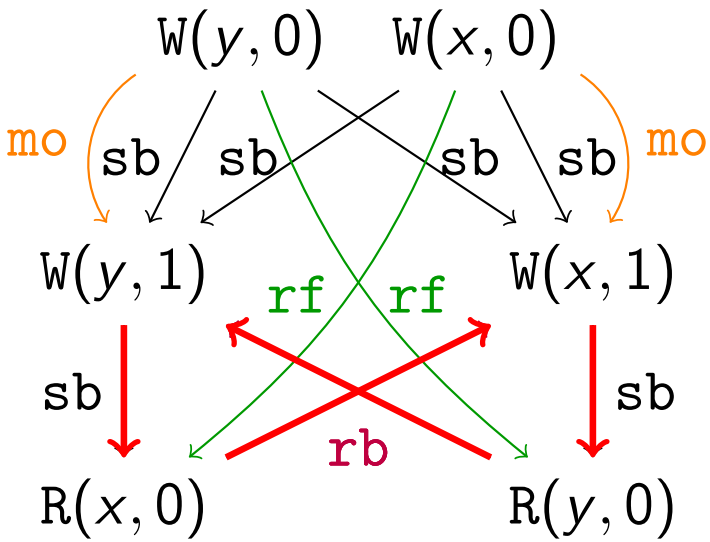
$$\mathit{acyclic}\ (\texttt{sb} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{rb})$$

Exemple d'exécution non-SC



1. Les deux lectures sont effectuées avant les deux écritures

Exemple d'exécution non-SC



1. Les deux lectures sont effectuées avant les deux écritures
2. Cela crée un cycle dans $(\mathtt{sb} \cup \mathtt{rb})$

Le standard C/C+11

- Chaque architecture matérielle a son modèle mémoire.
- Langages de haut niveau : il faut un modèle mémoire unique et portable
- Peut-on utiliser SC ? Rend impossible beaucoup d'optimisations.
- Le standard C/C++11 décrit un modèle et définit des opérations de *synchronisation*

Synchronisation en C/C++11

1. Des accès atomiques
2. Des barrières mémoire
3. Une opération *read-modify-write* atomique

Niveaux d'atomicité

Niveaux d'atomicité

- Attributs sur les accès à la mémoire
- Permet de restreindre les relations de communication
- Plus la restriction est forte, plus le coût est important

- **SC** : Si tous les accès sont de niveau SC, l'exécution est séquentiellement consistante.
- ***Release-acquire*** : Permettent d'implémenter les *locks*.
- **Relâchés** : Compilés en accès normaux, mais peuvent interagir avec les autres atomiques

Une idée du modèle mémoire

- **Évènements** : Chaque évènement est annoté par un niveau d'atomicité.
- La relation hb : ordre d'exécution imposé par les *release-acquire*

$$\texttt{hb} \approx (\texttt{sb} \cup \texttt{rf}_{\texttt{RA}})^{+}$$

- La relation psc : condition SC restreinte aux évènements SC

$$\texttt{psc} \approx (\texttt{sb} \cup \texttt{rf}_{\texttt{sc}} \cup \texttt{mo}_{\texttt{sc}} \cup \texttt{rb}_{\texttt{sc}})$$

Une idée du modèle mémoire

- **Évènements** : Chaque évènement est annoté par un niveau d'atomicité.
- La relation $\texttt{hb}$: ordre d'exécution imposé par les *release-acquire*

$$\texttt{hb} \approx (\texttt{sb} \cup \texttt{rf}_{\texttt{RA}})^{+}$$

- La relation $\texttt{psc}$: condition SC restreinte aux évènements SC

$$\texttt{psc} \approx (\texttt{sb} \cup \texttt{rf}_{\texttt{sc}} \cup \texttt{mo}_{\texttt{sc}} \cup \texttt{rb}_{\texttt{sc}})$$

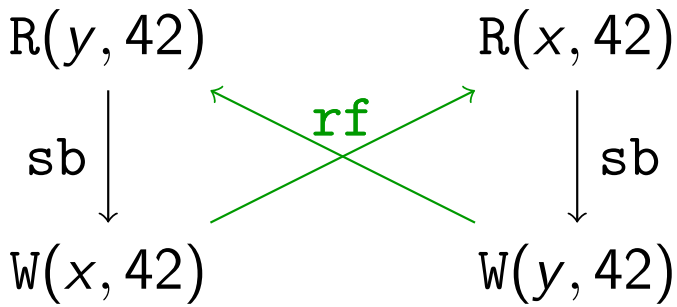
Condition de validité d'une exécution dans le modèle C11 :

- $\texttt{hb}; (\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^{*}$ est irréfléxif. L'exécution doit être compatible avec l'ordre imposé par les *release-acquire*.
- $\texttt{psc}$ est acyclique. Si on ne considère que les évènements SC, l'exécution doit être séquentiellement consistante.

Exclusion des exécutions *out-of-thin-air*

- Exécution *out-of-thin-air* : valeur qui vient "de nulle part"
- Pour l'éviter, on impose que $(\texttt{sb} \cup \texttt{rf})$ soit acyclique
- C'est un problème complexe. Cette condition est simple, mais elle exclut certaines optimisations

T_0	T_1
`a = x;`	`b = y;`
`y = a;`	`x = b;`



C11, c'est compliqué ...

Points positifs

- Contrôle fin de la synchronisation et de ses coûts
- Schémas de compilation corrects vers plusieurs architectures (x86, Power, ARM)
- Permet d'appliquer des optimisations aux programmes C/C++11

⚠ Problèmes

- Modèle plus complexe que présenté : interactions entre différents atomiques, *read-modify-write*, barrières de plusieurs niveaux ...
- Modèle assez complexe pour qu'il soit facile de se tromper sur le comportement d'un programme.
- Comment le rendre utilisable plus simplement ?

La propriété DRF-SC

Idée : Un programme C/C++11 sera séquentiellement consistant si il ne comporte pas de *race* : deux évènements conflictuels qui peuvent s'exécuter simultanément.

DRF-SC faible

Si une exécution n'est pas conforme à SC, elle contient une *race*

DRF-SC fort

Si toutes les exécutions SC du programme sont dépourvues de *races*, alors toutes les exécutions conformes à C11 sont également SC.

$\hookrightarrow$ Cette version est utilisable par quelqu'un qui ne connait pas le modèle. C'est celle que nous avons vérifié.

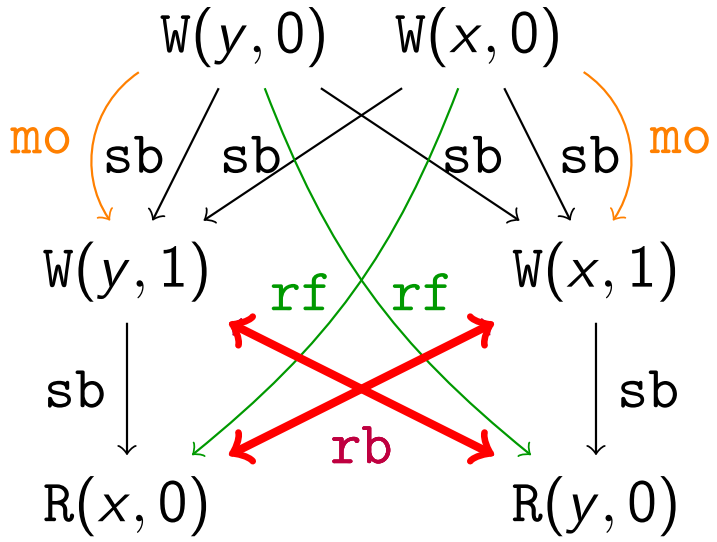
La notion de *race*

Des évènements forment une *race* si :

1. Ils accèdent au même emplacement mémoire
2. L'un d'entre eux au moins est une écriture
3. L'un d'entre eux au moins n'est pas un atomique SC
4. Ils ne sont pas ordonnés par $\texttt{hb}$ ou $\texttt{hb}^{-1}$

Intuition

- Conditions ① et ② : caractérisent les évènements potentiellement conflictuels
- Conditions ③ et ④ : si rien n'impose un ordre, les évènements peuvent être simultanés



Obstacle à la preuve

Exécutions d'un programme

La formulation de DRF-SC mentionne "toutes" les exécutions d'un programme. Qu'est-ce que cela signifie ?

- Première solution : avoir une sémantique complète des exécutions possibles d'un programme C/C++11 $\Rightarrow$ (trop) compliqué
- Deuxième solution : on définit axiomatiquement des transformations qui garde l'exécution dans le même programme $\Rightarrow$ mieux

Exécutions d'un même programme

On peut, en restant dans le même programme :

1. Prendre le *préfixe* d'une exécution : un sous-ensemble de ses évènements clos par rapport à $(\texttt{sb} \cup \texttt{rf})$. Aucune valeur modifiée.
2. Changer la relation `mo` d'une exécution. Aucune valeur modifiée.
3. Changer la valeur lue par une lecture à la fin d'un *thread* (sb-finale). Une seule valeur modifiée, mais ne peut pas changer le reste de l'exécution.

Principe de la preuve

DRF-SC Fort

- Si toutes les exécutions SC d'un programme sont dépourvues de *races*, alors toutes les exécutions possibles dans C11 du programme seront également SC.
- **Contraposée** : Si une exécution possible dans C11 n'est pas SC, alors il existe une autre exécution du même programme, SC et contenant une *race*.

Grandes lignes de la preuve

1. On suppose une exécution C/C++11 mais pas SC
2. On identifie une arête du cycle qui rend cette exécution non-SC
3. On transforme le programme avec les axiomes : on casse le cycle et trouve une *race*

Identifier une arête du cycle

- On utilise une propriété proche de DRF-SC faible :

Super-race

- Ils concernent le même emplacement mémoire
- Au moins l'un d'entre eux est une écriture
- Au moins l'un d'entre eux n'est pas SC
- Ils ne sont pas ordonnés par $(\mathtt{sb} \cup \mathtt{rf}|_{\mathtt{sc}})$ ou $(\mathtt{sb} \cup \mathtt{rf}|_{\mathtt{sc}})^{-1}$

- Notion très proche de celle de *race*, mais plus large $(\mathtt{sb} \cup \mathtt{rf}|_{\mathtt{sc}} \subseteq \mathtt{hb})$

SRF-SC

Si une exécution C11 ne contient pas de *super-race*, alors elle est SC.

Supprimer le cycle

On suppose une exécution possible en C/C++11, mais pas SC :

- Elle doit contenir une *super-race* (elle serait SC dans le cas contraire)
- On prend le plus petit préfixe de l'exécution qui contient une *super-race*
- Le premier évènement k qu'on ne peut pas supprimer fait partie de la *super-race* et du cycle qui rend l'exécution non-SC
- En fonction de la nature de k, on applique les axiomes pour :
  - Supprimer le cycle et rendre l'exécution SC
  - Transformer la *super-race* en *race*

Formalisation en Coq

- Cette preuve existe déjà, mais pas formalisée dans un assistant de preuves (**Repairing Sequential Consistency in C/C++11**. Lahav, Vafeiadis, Kang et Al. PLDI17)
- Formalisation de cette preuve : 8000 lignes de code

Points forts de Coq

- Traduction naturelle des définitions et des énoncés de théorèmes
- Sûreté des raisonnements non triviaux et non détaillés
- Existence d'outils pour automatiser des raisonnements sur les relations

Sûreté des raisonnements avec Coq

- Transformations de programmes axiomatisés qui produisent des exécutions cohérentes
- Explicitation de certains raisonnements :

Exemple

- **Lemme** : La transformation effectuée dans un cas particulier donne bien une exécution SC
- Dans la preuve papier : "*It is easy to see that [the execution] is SC-consistent.*"
- Dans la preuve Coq : 218 cas à prendre en compte. La certitude apportée par Coq est la bienvenue.

Travailler avec les relations en Coq

- `relation-algebra` : procédure de décision pour les (in)équations dans KAT (Kleene Algebra with Tests)
  - Kleene Algebra : relations binaires + union + composition + clôtures réfléxive et/ou transitive
  - Tests : tests booléens qu'on peut combiner avec les relations
- Adapté pour les modèles mémoires : contraintes sur les relations (type d'évènements, niveaux d'atomicité) exprimées par des tests.

  Exemple : $\texttt{rf}_{\texttt{RA}} = \texttt{[M Rel];[W];rf;[R];[M Acq]}$

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
```

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
  apply antisym; [|kat].
  apply itr_ind_l1; [kat|].
```

1. `kat` **résout** une direction et le cas de base de l'induction.

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
  apply antisym; [|kat].
  apply itr_ind_l1; [kat|].
  ra_normalise;
  rewrite mo_trans2;
  ra_normalise.
  repeat (try (apply leq_cupx));
```

1. `kat` **résout** une direction et le cas de base de l'induction.
2. `ra_normalise` **normalise** le but et on le décompose

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
  apply antisym; [|kat].
  apply itr_ind_l1; [kat|].
  ra_normalise;
  rewrite mo_trans2;
  ra_normalise.
  repeat (try (apply leq_cupx));
  try (elim_conflicting_rw);
  try (elim_conflicting_wr);
```

1. `kat` **résout** une direction et le cas de base de l'induction.
2. `ra_normalise` **normalise** le but et on le décompose
3. On **réécrit** pour rajouter de l'information et `kat` **résout** les cas absurdes (`[W];[R]` et `[R];[W]`)

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
  apply antisym; [|kat].
  apply itr_ind_l1; [kat|].
  ra_normalise;
  rewrite mo_trans2;
  ra_normalise.
  repeat (try (apply leq_cupx));
  try (elim_conflicting_rw);
  try (elim_conflicting_wr);
  unfold rb;
  try (mrewrite mo_trans2);
  try (mrewrite rfrfinv);
  kat.
```

1. `kat` **résout** une direction et le cas de base de l'induction.
2. `ra_normalise` **normalise** le but et on le décompose
3. On **réécrit** pour rajouter de l'information et `kat` **résout** les cas absurdes (`[W];[R]` et `[R];[W]`)
4. On **réécrit** et **résout** les cas restant avec `kat`

Travailler avec les relations en Coq

```
Lemma eco_rfmorb_seq_rfref:
  (rf + mo + rb)^+ =
  rf + ((mo + rb);rf?).
Proof.
  apply antisym; [|kat].
  apply itr_ind_l1; [kat|].
  ra_normalise;
  rewrite mo_trans2;
  ra_normalise.
  repeat (try (apply leq_cupx));
  try (elim_conflicting_rw);
  try (elim_conflicting_wr);
  unfold rb;
  try (mrewrite mo_trans2);
  try (mrewrite rfrfinv);
  kat.
Qed.
```

1. `kat` **résout** une direction et le cas de base de l'induction.
2. `ra_normalise` **normalise** le but et on le décompose
3. On **réécrit** pour rajouter de l'information et `kat` **résout** les cas absurdes (`[W];[R]` et `[R];[W]`)
4. On **réécrit** et **résout** les cas restant avec `kat`

Bilan

Preuve avec uniquement des réécritures et une procédure automatique

Peut-on adapter la preuve à d'autres modèles

Quels sont les obstacles si on souhaite adapter cette preuve à d'autres modèles mémoire ?

- La preuve utilise la notion de *super-race* qui est spécifique à C11

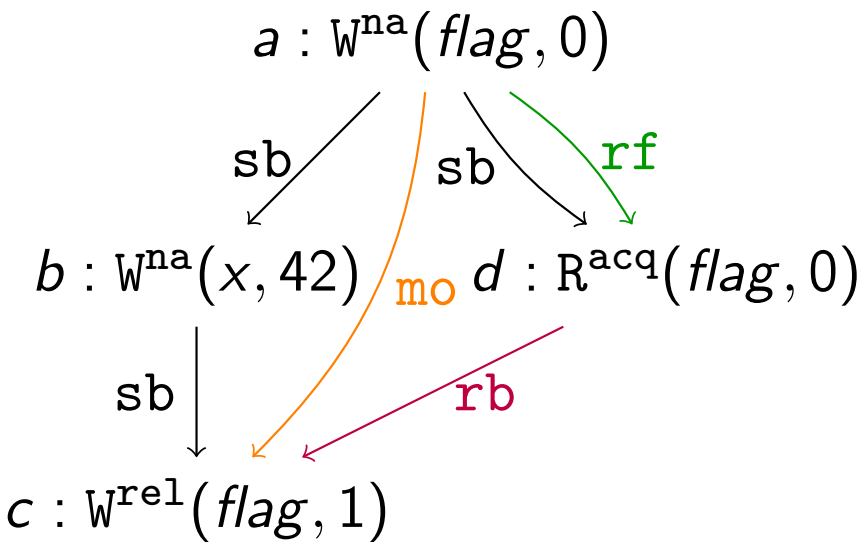
Hypothèse

La preuve peut fonctionner de la même façon si on remplace les *super-race* par des *races* et si on utilise le théorème DRF-SC faible

- Division en deux problèmes. Quelles propriétés un modèle doit avoir pour :
  1. vérifier la propriété DRF-SC faible ?
  2. la propriété DRF-SC faible implique DRF-SC fort ?

Le critère d'absence de *race*

$$x :=_{\mathtt{na}} 42 \quad \| \quad r_1 :=_{\mathtt{acq}} flag$$
$$flag :=_{\mathtt{rel}} 1 \quad \| \quad if\,(r_1)\ \ r_2 :=_{\mathtt{na}} x$$



- Exemple de *message passing*.
- Quand l'acquisition échoue, *race* entre c et d.
- Pourtant, l'exécution est SC
- Est-ce que l'absence de *race* n'est pas une condition trop forte pour DRF-SC ?

Conclusion

- Dans le modèle C11, si on synchronise correctement les *races* potentielles, on retrouve des exécutions séquentiellement consistantes
- Nous l'avons vérifié en Coq
- Cette preuve pourait être généralisée, et découpée en deux propriétés indépendantes.
- Imposer l'absence de *race* est peut-être trop fort pour restaurer une sémantique SC. Une preuve générique et certifiée par Coq serait utile pour explorer cette possibilité.

`https://github.com/qladevez/rc11`

Le modèle C11 complet : hb

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

1. On relie toujours un *release* ou un SC à un *acquire* ou un SC.
2. Des barrières peuvent jouer le rôle d'un *release* ou d'un *acquire*
3. On peut avoir une chaîne d'évènements au même emplacement après le *release*
4. La *communication* se fait entre des atomiques au moins relâchés.
5. La lecture du *release* par le *acquire* peut être précédée par une chaîne de *read-modify-write*
6. Le *release* lit la valeur écrite par le *acquire*

Le modèle C11 complet : psc

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

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

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

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