TP10 Modélisation, simulation et vérification du « Priority Inheritance Protocol » en Kind2

1 Plan

En suivant l'exemple de Jahier, Halbwachs et Raymond [2], nous allons modéliser quelques éléments des protocoles « Priority Inheritance » et « Priority Ceiling » [3] avec le langage synchrone Lustre. Contrairement aux analyses réalisées avec les assistants de preuve PVS [1] et Isabelle [5], nous n'allons vérifier que quelques scénarios particuliers par Model-Checking en utilisant l'outil Kind2.

Kind2 traduit un programme Lustre, annoté avec des propriétés de sûreté, en une formule logique et y applique des manipulations symboliques sophistiquées (Bounded-Model Checking, k-induction et IC3/PDR) et un solveur SMT (dans notre cas : le solveur Z3 de Microsoft Research) pour essayer de démontrer les propriétés ou, à défaut, trouver un contre-exemple. Le fonctionnement de Kind2 nous importe peu ici, on se servira de l'outil comme d'un oracle [1].

La simulation est un point de départ indispensable pour comprendre le comportement d'un modèle dynamique. On pourra utiliser les outils de Lustre v4 pour compiler et visualiser les programmes:

`http://www-verimag.imag.fr/The-Lustre-Toolbox.html`

Ces outils sont plutôt des projets de recherche. Il est donc inévitable de se débattre un peu avec eux ...

2 Préparation

Les outils Kind2, Z3 et Lustre v4 ont déjà été téléchargés sur les ordinateurs fixes (dossier `/‘hostname‘`). Ils sont également téléchargeable aux adresses mentionnées ci-dessous:

`http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/linux64/lustre-v4-III-c-linux64.tgz`

`https://github.com/Z3Prover/z3`

`http://kind2-mc.github.io/kind2/`

Pour utiliser les outils de Lustre (`lustre` et `luciole`), il faut commencer par exécuter les instructions suivantes dans un shell:

```
export LUSTRE_INSTALL=/‘hostname‘/lustre-v4-III-c-linux64
source $LUSTRE_INSTALL/setenv.sh
```

Les fichiers du TP sont disponibles sur la page du cours :

`http://www.di.ens.fr/~pouzet/cours/systeme/tp10/tp-10.tgz`

3 Modélisation d'un système temps réels à trois processus

Considérons un système temp réels composé des trois processus présentés dans la figure 1 et des deux ressources partagées suivantes:

1. Pour en savoir plus, vous pouvez suivre le cours du MPRI « Parallélisme Synchrone ».

process 0 *(haute priorité)* déclenché par interruption exectime = 8		**process 1** *(moyen priorité)* décalage: 4 période: 20 exectime = 7		**process 2** *(bas priorité)* en continu (idle) exectime = 11	
0	work	0	work	0	work
1	**lock(0)**	1	**lock(1)**	1	work
2	work	2	work	2	**lock(0)**
3	**lock(1)**	3	work	3	work
4	work	4	work	4	work
5	**release(1)**	5	**release(1)**	5	work
6	**release(0)**	6	sleep	6	work
7	sleep			7	work
				8	**release(0)**
				9	work
				10	loop

FIGURE 1 – Les trois processus.

ressource 0 (*serial port*)
ressource 1 (*gps*)

Le processus 0 est déclenché par une interruption. Il a besoin des deux ressources pour faire son calcul et dort entre deux interruptions. Il a la plus haute priorité (0), les deux autres processus ne doivent donc pas être exécuté si son exécution est possible ou déjà déclenchée.

Le processus 1 est activé tous les 20 ticks après un décalage initial de 4 ticks. Il n'a besoin que du 'gps' (ressource 1) pour faire son calcul et dort entre deux activations.

Le processus 2 s'exécute en continu avec la plus basse priorité, il ne s'exécute donc que quand les deux autres processus dorment. Il n'a besoin que du 'serial port (ressource 0) pour faire son calcul et boucle au lieu de dormir.

Chaque processus est modélisé par un nœud Lustre. Voici le modèle du processus 1:

```
node process1 (cpu : bool)
returns (pc : int; sleep : bool; asks_cs1 : bool);
let
    pc = counter_modulo (cpu, exectime[1]);
    sleep = cpu and (pc = exectime[1] - 1);
    asks_cs1 = false -> pre (1 <= pc and pc < 5);
tel;
```

Il n'y a qu'une entrée booléenne `cpu`. Le processus ne s'exécute que lorsque `cpu` est vraie (le système est mono processeur). Il y a trois sorties:

`pc`	le compteur ordinal ('program counter')
`sleep`	un signal pour demander la mise en sommeil auprès de l'ordonnanceur
`asks_cs1`	un signal pour demander la ressource 1 ('cs' = 'critical section')

Le compteur ordinal est à 0 avant qu'un processus ne s'exécute pour la première fois, sinon c'est la valeur associée à l'instruction qui s'est exécutée dans l'instant courant. Une exécution typique du modèle est présentée dans la figure 2.

Le fichier `sched.lus` contient les nœuds principaux du modèle:

`dispatcher`	la logique du déclenchement des processus.
`scheduler`	l'ordonnanceur qui détermine quel processus exécuter à chaque tick.
`system`	Le système globale pour simuler/vérifier.

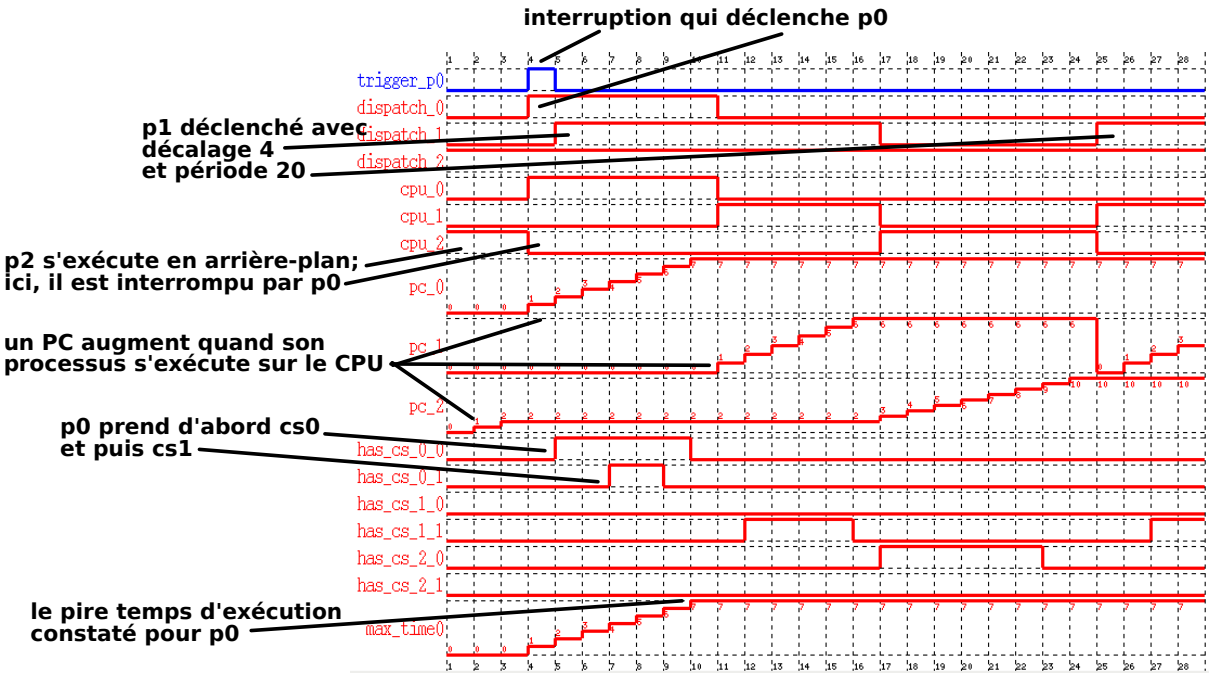

FIGURE 2 – Une exécution typique du modèle.

Question 1. Complétez les définitions des processus 0 et 2 et simulez le modèle avec `luciole`.

Le programme `luciole` fait partie des outils de Lustre v4. Il fournit une interface graphique simple pour compiler et simuler des programmes Lustre. Pour démarrer une simulation, tapez la commande:

```
luciole sched.lus system
```

Le premier argument est le nom du fichier, le deuxième est le nom d'un nœud. Faites attention aux messages d'erreur, parfois `luciole` s'exécute même si la compilation échoue. L'interface de `luciole` est rudimentaire, mais suffisante pour nos expériences. Il y a deux boutons:

`Step`	exécuter un tick normal
`trigger_p0`	exécuter un tick avec interruption du processus 0

Les autres éléments affichent les valeurs des sorties du nœud `system`. Il est possible d'afficher un chronogramme en cliquant sur le menu `Tools` et puis `sim2chro`.

En regardant la simulation de près, vous pouvez constater que le statut de la ressource 1 dans le processus 1, `has_cs_1_1` n'est pas correctement défini.

Question 2. Corrigez les définitions `p1_had_cs1` et `p1_has_cs1` dans le nœud `scheduler` d'après les autres définitions données.

L'ordonnanceur actuel exécute toujours le processus de plus haute priorité, mais il ne respecte pas les sections critiques. Il est donc possible pour deux processus d'utiliser la même ressource en même temps.

Question 3. Définissez la propriété d'exclusion mutuelle, `mutual_exclusion`, dans le nœud `scheduler`.

Il y a quelques annotations en commentaire dans le fichier, comme la suivante:

```
--%PROPERTY mutual_exclusion;
```

Il s'agit des déclarations pour `kind2` qui essaie de vérifier qu'elles sont toujours vraies pour n'importe quelle exécution.

Question 4. Lancez `kind2` en tapant:

```
/‘hostname‘/kind2 --z3_bin /‘hostname‘/z3 sched.lus
```

Il devrait trouver que la propriété `cpu_good` est vérifiée et donner un contre-exemple pour la propriété `mutual_exclusion`.

4 Implémentation du blocage

Question 5. Faites en sorte que les sections critiques soient respectées. (Pensez à définir les prédicats `p0_blocks_p1`, `p1_blocks_p0`, `p0_is_blocked` et ainsi de suite, et de les ajouter à la définition de `cpu` dans `scheduler`.)

Question 6. Vérifiez à nouveau la propriété `mutual_exclusion` avec `kind2`.

5 Implémentation du « Priority Inheritance Protocol »

Pour cette partie, nous allons changer le programme exécuter par le processus 0 pour qu'il ne demande pas le gps (mettez `asks_c1 = false` dans la définition de `process0`).

Le mélange des priorités et des ressources partagés peut induire un effet délétère qui s'appelle l'inversion de priorité. Le processus 0 ne peut pas s'exécuter si le processus 2 est en train d'utiliser la ressource 0. C'est normal. Mais, maintenant si le processus 1 est déclenché, il s'exécutera alors même que le processus 0 est en attente.

Question 7. Pour détecter l'inversion de priorité, ajouter un prédicat au nœud `system`:

```
no_priority_inversion = not (dispatch[0] and cpu[1]);
--%PROPERTY no_priority_inversion;
```

Lancez `kind2` pour générer une trace qui montre l'inversion de priorité. Reproduisez cette trace avec `luciole`.

Le Priority Inheritance Protocol est fait pour empêcher ce genre de scénario. Quand un processus de haute priorité est en attente d'un processus de plus basse priorité, le premier « prête » sa priorité au second pour qu'il ne puisse pas être interrompu par d'autres processus.

Question 8. Implémentez la logique de ce protocole dans le modèle donné et vérifiez avec `kind2` qu'il n'y a plus d'inversion de priorité.

Jahier et autres [2, §3.3] montrent comment modéliser ce protocole pour un modèle arbitraire. Ils notent que *the difficulty is to translate [the] "dynamic" condition into a Boolean condition.* Si vous souhaitez suivre leur article, notez que dans le cas que nous considérons, il n'est pas nécessaire de calculer la clôture transitive de la relation de blocage.

6 Implémentation du « Priority Ceiling Protocol »

Pour cette partie, nous reprenons la version originale du processus 0 où les les deux ressources sont demandés.

L'inversion n'est ici plus possible, mais le comportement temporisé du système a toujours un défaut. Normalement, on s'attend à ce que le temps d'exécution pire cas (« Worst-Case Execution Time (WCET) ») du processus 0 soit le temps d'exécution normal plus le maximum des temps d'exécution des sections critiques des processus 1 et 2 (dans le cas où il faut attendre un ou l'autre pour avoir une ressource). Dans notre exemple, on s'attend donc à un délai maximal de 13 ticks (8+5) entre l'interruption qui déclenche le processus 0 et la terminaison de ce processus.

Question 9. Ajoutez une proposition pour vérifier cette attente:

```
high_priority_wcet = max_time0 <= exectime[0] + longest_cs;
--%PROPERTY high_priority_wcet;
```

Essayez de vérifier cette propriété. Que se passe-t-il (regardez avec `luciole`) ?

Le problème est que, dans le pire des cas, le processus 0 doit attendre les deux processus pour avoir les ressources nécessaires à compléter son travail. Le « Priority Ceiling Protocol » est fait pour éviter ce genre de scénario.

Question 10. Dans notre exemple, il suffit d'empêcher que les deux ressources soient prises en même temps par les processus 1 et 2. Modifiez le nœud `scheduler` pour implémenter cette règle. Vérifiez la propriété de WCET avec `kind2`. Il est vraisemblable que `kind2` n'y arrive pas. Vous pouvez utiliser `ctrl-c` pour le tuer après quelques minutes.

Question 11. Le problème est que l'espace d'états des ordonnancements possibles est très large à cause du non-déterminisme introduit par l'entrée d'interruption. Que peut-on faire ? L'outil `kind2` est censé exploiter les autres propriétés déclarées dans le programme comme invariants. Y'a-t-il des invariants qui peuvent aider `kind2`? (Je n'en ai pas encore trouvé...)

Question 12. *Pour aller plus loin:* Implémentez une version générale du protocole du plafond de priorité [2, §3.4] et expérimentez avec plus de processus et de ressources.

Références

[1] B. Dutertre. Formal analysis of the priority ceiling protocol. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 151–160, Orlando, FL, USA, Nov. 2000. IEEE Computer Society.

[2] E. Jahier, N. Halbwachs, and P. Raymond. Synchronous modeling and validation of priority inheritance schedulers. In M. Chechik and M. Wirsing, editors, *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 140–154, York, UK, Mar. 2009. Springer.

[3] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[4] V. Yodaiken. Against priority inheritance. Technical report, Finite State Machine Labs, Sept. 2004.

[5] X. Zhang, C. Urban, and C. Wu. Priority inheritance protocol proved correct. In L. Beringer and A. Felty, editors, *Proc. 3rd Int. Conf. on Interactive Theorem Proving (ITP 2012)*, volume 7406 of *Lecture Notes in Computer Science*, pages 217–232, Princeton, NJ, USA, Aug. 2012. Springer.