Vérification d'applications temps-réel basées sur le paradigme de Logical Execution Time (LET)

Fabien Siron, Dumitru Potop-Butucaru, Robert de Simone, Damien Chabrol, Amira Methni

Vérification d'applications temps-réel basées sur le paradigme de Logical Execution Time (LET)

Fabien Siron*†, Dumitru Potop-Butucaru†, Robert de Simone†, Damien Chabrol* et Amira Methni*
*Krono-Safe - Massy, France
†INRIA - Paris/Sophia-Antipolis, France
*firstname.lastname@krono-safe.com
†firstname.lastname@inria.fr

***Résumé*—Le design de logiciel de contrôle/commande embarqué dépend de contraintes temporelles strictes. Pour cela, des formalismes et des théories basés sur la notion de temps logique permettent d'abstraire les durées temps-réel qui ne sont pas toujours connues au niveau conception. Cet article propose une comparaison des formalismes synchrones et du paradigme *Logical Execution Time (LET)* dans le but d'adapter au langage industriel *PsyC*, qui est proche du *LET*, des méthodes de vérification issues des langages synchrones.**

I. Introduction

Le design de logiciel de contrôle/commande embarqué, intéragissant massivement avec leur environnement, dépend de contraintes temporelles strictes. La correction temporelle du système doit être assurée au plus tôt dans le cycle logiciel, en particulier dans le contexte de systèmes critiques. Cependant, les durées temps-réel ne sont pas toujours connues au moment de la conception.

Plusieurs formalismes et théories basés sur la notion de temps logique ont été introduits afin de palier à ce problème. Le design est ainsi divisé en deux phases : d'une part la spécification, la programmation et la vérification basés sur le temps logique imposant des hypothèses sur les temps d'exécution et d'autre part, la validation de ces hypothèses vis à vis de temps physique lié à l'implémentation. Cet article propose une comparaison des formalismes synchrones et du paradigme *Logical Execution Time (LET)* (tous les deux issus de la notion de temps logique), dans le but d'adapter au langage industriel *PsyC*, qui est proche du *LET*, des méthodes de vérification issues des langages synchrones. En effet, bien qu'il y ait eu beaucoup de travaux sur la vérification formelle de langages synchrones [1], à notre connaissance, il n'en existe pas pour les langages complexes basés sur le *LET* tel que *PsyC*.

Dans la première section, nous détaillerons la notion de langage synchrone ainsi que ses différentes variantes. Nous détaillerons ensuite dans la deuxième section le paradigme *LET* ainsi que ses différentes implémentations et nous le comparerons aux approches synchrones. Nous donnerons ensuite une description informelle du langage *PsyC*, qui est une implémentation du paradigme *LET* par la société *Krono-Safe*. Nous terminerons par montrer comment le paradigme synchrone peut être utilisé pour la vérification d'applications basées sur le paradigme *LET*, tel que le *PsyC*.

II. Langages Synchrones

Les langages synchrones permettent d'atteindre les propriétés de déterminisme et de concurrence au travers de calculs élémentaires qui réagissent simultanément et instantanément à chaque tick d'une horloge globale commune. Dans ces langages, le temps n'est pas exprimé explicitement, mais est vu comme une séquence ordonnée de réactions atomiques (ou *instants*) du système ; la notion de temps est donc remplacée par la notion d'ordre, on parle alors de temps logique. L'hypothèse synchrone garantie alors, que si tous les calculs sont effectués avant la prochaine réaction, alors le temps physique peut être ignoré de manière sûre [2].

On peut classer les langages synchrones en deux grandes familles :

- les langages synchrones *flot de données* (e.g. tel que *Lustre* [3]) qui sont généralement déclaratifs, formés d'équations temporelles sur des flots auquels sont associés des horloges ;
- et les langages synchrones *flot de contrôle* (e.g. tel qu'*Esterel* [4]) qui sont généralement impératifs, permettant la concurrence impérative (i.e. l'existence de plusieurs pointeurs d'instructions différents coexistants au même instant) ainsi que la préemption qui permet d'interrompre une exécution.

Il convient cependant de nuancer les avantages des langages synchrones. Leur compilation est assez complexe et ne permet pas, généralement, de générer du code parallèle. De plus, les temps d'exécution physiques sont généralement limités au temps de réaction du système, appelé pire temps de réaction (WCRT). Cela complique la modélisation de système multi-périodiques, qui pourtant, aujourd'hui, deviennent de plus en plus nombreux.

Des approches plus récentes permettent toutefois de palier à ces problèmes en affaiblissant le paradigme synchrone tout en maintenant sa propriété de déterminisme. On peut citer l'algèbre synchrone *CoReA* de Boniol (*Communicating Reactive Automata* [5], inspiré de *SCCS* [6]), qui développe un modèle synchrone dit *faible* (en opposition au modèle synchrone classique, dit *fort*). Dans ce modèle, les calculs réagissent aux ticks d'une horloge globale, comme dans le modèle *synchrone fort*, mais produisent le résultat de leur réaction au prochain instant. Plus récemment, les travaux de Forget [7] introduisent un langage synchrone déclaratif multi-périodique, appelé *Prélude*. Ce langage, basé sur *Lustre*, développe l'idée d'une hypothèse synchrone *relachée*, c'est à dire que chaque flot, lors de sa réaction, doit finir son calcul

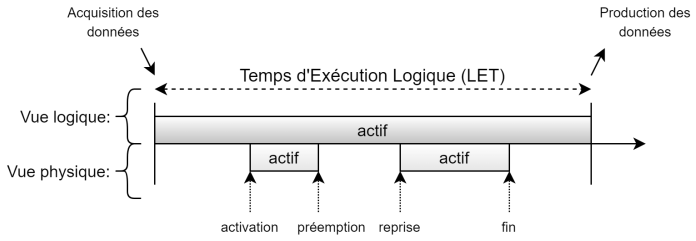

FIGURE 1. Modélisation d'un intervalle *LET*

avant sa prochaine activation. Cela implique que chaque flot à sa propre notion d'instant et permet ainsi la modélisation de système multi-périodiques. Affaiblir le modèle synchrone permet aussi d'éviter les problèmes de causalité, inhérents à ces approches [8].

III. LOGICAL EXECUTION TIME

Le paradigme *Logical Execution Time (LET)* [9] abstrait, de manière similaire au paradigme synchrone, le temps physique, via une succession d'instants logiques. Cependant, le calcul n'est pas considéré instantané mais durant un intervalle de temps logique (ou intervalle *LET*) défini par un instant de départ et de fin. Les communications sont ainsi effectuées sur ces deux instants : les entrées sont consommées sur l'instant de départ et les sorties sont produites sur l'instant de fin (voir figure 1). En d'autre terme, la durée effective d'une réaction est abstraite par le délai entre entrées et sorties. Cela se justifie du fait que le temps observable d'une fonction n'est pas réellement son temps effectif de calcul, mais le temps entre la prise en compte des entrées et la production des sorties.

En considérant que le *LET* étend le concept de temps logique avec la notion de durée, cela permet une plus grande variation des temps d'exécution. Cela permet ainsi de traiter la problèmatique des systèmes multi-périodiques et de faciliter la compilation vers du code parallèle (i.e. multi-tâche, multi-coeur ...). De plus, les propriétés de déterminisme et de concurrence issues du paradigme synchrone sont préservées du fait que toutes les communications sont effectuées sur des dates prédéfinies.

Bien que le *LET* soit généralement classé dans les modèles de calcul temporisés [10] du fait de sa proximité avec l'architecture timed-triggered (TTA) concue par Kopetz [11], les instants délimitants les intervalles sont relatifs à une l'horloge commune globale qui est logique. Ainsi, cette horloge peut aussi rythmer les réactions du système de manière évènementiel. Le paradigme *LET* est par conséquent basé sur un temps logique, tel que le paradigme synchrone, et n'est donc pas explicitement temporisé. *Giotto* [12] est une implémentation du *LET* avec des tâches simplement périodiques basées sur le temps physique. Il est étendu au travers de bases de temps évènementielles dans le langage *xGiotto* [13] supportant les tâches non périodiques. Ce langage propose une généralisation du paradigme *LET* acceptant des intervalles *LET* de taille variable. Le langage *PsyC* [14] supporte lui aussi des tâches non périodiques basées sur une base de temps logique. Il est développé dans la prochaine section.

On peut remarquer que le paradigme *LET* partage de

```
clock 2MS = 2 * MS;

temporal int tv_signal = 0 on MS;
temporal int tv_threshold = 0 on 2MS;

agent Threshold(uses realtime, starttime 2 MS)
{
  consult tv_signal $ 2;
  display tv_threshold;
  body start
  {
    tv_threshold = $[0]tv_signal > THRESHOLD ||
                   $[1]tv_signal > THRESHOLD;
    advance 1 with MS;
    /* boucle infinie sur le body start */
  }
}
```

Code 1. Exemple d'un agent *PsyC*

grandes similarités avec le paradigme synchrone. Tous deux permettent de préciser des instants en fonction d'une horloge globale commune aux différentes tâches. Dans le *LET*, contrairement au synchrone, les exécutions ne sont toutefois pas instantanées mais bornées par l'instant de fin de l'intervalle. Il convient cependant de noter qu'une exécution instantanée des calculs est une abstraction correcte du *LET* du fait que le déterminisme est assuré par le modèle de communication et non la durée des réactions. Le modèle de communication peut être abstrait par un délai systématique des communications produites par une tâche sur le prochain instant d'activation de celle-ci. Le paradigme *LET* peut ainsi être abstrait par le paradigme synchrone avec un délai systématique des communications, en d'autre termes, le paradigme synchrone dit *faible* tel que décrit par *CoReA*. Cette abstraction permet, par conséquent, de simplifier le modèle et de faciliter la vérification.

IV. PSYC

Le langage *PsyC* est une implémentation d'une forme généralisée du paradigme *LET* (similairement à *xGiotto*) dédiée aux applications temps-réel critiques. Il a été conçu comme un sur-ensemble du langage C afin de faciliter l'intégration d'applications conçues avec ce langage. Le langage est implémenté dans la suite *ASTERIOS* qui est produite par la société *Krono-Safe*. Cette suite outillée permet le développement d'applications aéronautiques critiques certifiées au plus haut niveau de criticité (DAL-A, DO-178C).

Une application *PsyC* est composée de tâches appelées *agents* qui sont des unités de calcul séquentielles formées d'une suite d'intervalles *LET*. Le contenu d'un agent est composé de code C où une instruction spéciale, `advance n c` dénote les bornes des intervalles *LET* en avançant la date logique de n ticks d'une horloge c dérivée d'une source temporelle unique. Le déterminisme est issu du principe de visibilité des communications inter-tâches. Les données sont timestampées à la date où elles sont émises. Ainsi, au sein d'un intervalle *LET*, les données externes visibles sont celles dont le timestamp est inférieur ou égal à la date logique courante, tandis que les données produites deviennent visibles

au prochain instant d'activation défini par la prochaine instruction `advance` (i.e. l'instant clôturant l'intervalle courant). Au niveau de l'application, les *agents* sont composés en parallèle de manière synchrone et communiquent entre eux principalement via un mode de communication échantillonné appelé *variable temporelle* (VT). L'écriture d'une VT se fait comme une affectation classique mais la lecture se fait via l'expression `$[n]tv` qui dénote le n$^{\text{ième}}$ dernier élément de la VT échantillonnée sur son horloge.

Le code 1 décrit un *agent PsyC* possédant une variable temporelle en entrée échantillonné sur la milliseconde (horloge MS) et une autre variable temporelle en sortie échantillonné sur une horloge 2 fois plus lente (horloge $2MS$) dénotant le dépassement d'un seuil. La déclaration d'horloge `clock` permet de spécifier l'horloge $2MS$ en fonction de l'horloge MS. Les deux variables temporelles sont ensuite déclarées avec une valeur d'initialisation ainsi que leurs horloges respectives. L'agent démarre à la date 2 ms et actualise la variable temporelle de sortie toutes les 1 ms en spécifiant si le seuil a été dépassé.

V. VÉRIFICATION

A. Observateurs

Nous nous intéressons ici aux propriétés de sûreté des systèmes temps-réel telles que des latences ou bien des motifs de cadencement. Classiquement, ces propriétés peuvent être modelisées via des logiques temporelles telles que *LTL* (logique temporelle linéaire) ou *CTL* (logique temporelle arborescente) [15]. Toutefois, issu des travaux sur les langages synchrones, les observateurs permettent de représenter les propriétés directement dans le formalisme utilisé par le système [1]. Les observateurs détectent les mauvaises exécutions du système (i.e. les exécutions partielles qui invalident une propriété du système). Cela permet, d'une part, d'avoir une homogénéité entre la représentation des propriétés et du système, et d'autre part, de vérifier les propriétés dynamiquement (via du *Runtime Monitoring* [16]) ou bien statiquement (via du *Model-Checking* [1] [15]).

Le code 2 propose un exemple d'observateur vérifiant, sur l'*agent* décrit dans le code 1 la propriété suivante : lorsque le signal dépasse un seuil, alors le signal de sortie est actif lors de l'instant suivant ou celui d'après. Cela se traduit en logique temporelle par la formule suivante [15] :

$$\Box(P \implies \bigcirc(Q \vee \bigcirc Q)) \tag{1}$$

où $P \stackrel{\text{def}}{=} tv\_signal > THRESHOLD$ et $Q \stackrel{\text{def}}{=} tv\_threshold = true$. Cette forme typique spécifie une latence entre l'évènement P et Q d'au minimum un instant et d'au maximum deux instants. Il faut le lire de la manière suivante : *il est toujours vrai que lorsque P est vrai alors Q est vrai à l'instant suivant ou celui d'après*. Le symbole $\Box$ dénote une propriété vraie sur tous les instants et le symbole $\bigcirc$ dénote une propriété vraie sur l'instant suivant. Il convient toutefois de préciser que cette équivalence ne prend pas en compte le *starttime* (i.e. la date de départ de l'*agent*).

```
agent Observer(uses realtime, starttime 2 MS)
{
  consult tv_signal $ 1;
  consult tv_threshold $ 1;
  body start {
    if ($[0]tv_signal > THRESHOLD) {
      advance 1 MS;
      if ($[0]tv_threshold != 1) {
        advance 1 MS;
        if ($[0]tv_threshold != 1)
          ast_error_raise(/* code d'erreur */);
      }
    } else {
      advance 1 MS;
    }
  }
}
```

Code 2. Exemple d'un observateur reprensenté sous forme d'agent *PsyC*

B. Vérification Dynamique via Runtime Monitoring

Les propriétés peuvent être vérifiées directement à l'exécution en considérant les observateurs comme des tâches standards. Le système doit toutefois être doté d'un mécanisme de gestion d'erreur à l'exécution. Dans le cas du *PsyC*, l'instruction `ast_error_raise` permet de lever une erreur au sein d'un *agent* en y associant une sanction (e.g. extinction de la tâche, extinction du système entier ...).

C. Vérification Statique via Model Checking

Les observateurs peuvent aussi servir à la vérification statique via du *model-checking*. En reprenant l'analogie décrite plus haut entre *LET* et synchrone, on peut proposer une traduction du modèle et de son observateur dans un formalisme synchrone. Le langage *PsyC* étant principalement impératif et flot de contrôle, *Esterel* est tout à fait adapté car il partage ces deux aspects.

Les codes 3 et 4 décrivent les traductions respectives de l'agent et de l'observateur. Tout deux partagent une structure similaire où l'instruction `advance` (ainsi que `starttime`) sont remplacées par des instructions `await`. La différence principale réside cependant au niveau des communications où leur émission est systématiquement retardées sur l'instant correspondant à la fin de l'intervalle *LET*. On considère ici que les variables temporelles (VT) sont traduites en signaux valuées (en *Esterel* la valuation des signaux est persistente). Pour une VT `temporal`, le signal *Esterel* `temporal` dénote le signal d'écriture (non échantillonné) et les signaux `temporal_n` dénotent les signaux échantillonnés d'index n accessibles par les lecteurs. Ainsi, dans le code 3, `tv_signal_0` dénote le dernier échantillon de `tv_signal` (échantillonné sur le dernier tick de l'horloge MS) et `tv_signal_1` dénote l'avant dernier échantillon. Par manque de place, les traductions des variables temporelles ne sont pas données. Il s'agit toutefois simplement de module d'échantillonnage classique.

L'erreur levée par l'observateur est remplacée par un signal dénotant un mauvais état de celui-ci (i.e. une erreur). Lors de la compilation *Esterel* permet de générer un automate modélisant tout le système (observateur compris). Il suffit alors d'une analyse d'accessibilité sur l'état d'erreur de l'observateur. Si

```
module Agent:
  input MS;
  input tv_signal_0 : integer;
  input tv_signal_1 : integer;
  output tv_threshold : boolean;
  await 2 MS;
  loop
    if ?tv_signal_0 > THRESHOLD or
       ?tv_signal_1 > THRESHOLD then
      await 1 MS;
      emit tv_threshold(true)
    else
      await 1 MS;
      emit tv_threshold(false)
    end if;
  end loop
end module
```

Code 3. Traduction de l'*agent PsyC* en *Esterel*

```
module Observer:
  input MS;
  input tv_signal_0 : integer;
  input tv_threshold_0 : boolean;
  await 2 MS;
  loop
    if ?tv_signal_0 > THRESHOLD then
      await 1 MS;
      if ?tv_threshold_0  else
        await 1 MS;
        if ?tv_threshold_0  else
          emit Error
        end if
      end if
    else
      await 1 MS
    end if
  end loop
end module
```

Code 4. Traduction de l'observateur de *PsyC* en *Esterel*

cet état est atteignable, alors il peut y avoir une exécution ne respectant pas la propriété spécifiée par l'observateur, sinon, cette propriété est garantie pour toute exécution.

Bien qu'il suffit d'un parcours sur l'automate généré afin de vérifier la propriété, cette approche souffre quand même du problème de l'explosion combinatoire. L'automate généré grandi exponentiellement vis à vis du nombre d'état de chaque tâche. Toutefois, la composition synchrone permet de limiter drastiquement l'explosion combinatoire. D'autre part, la compilation d'*Esterel* permet aussi d'obtenir l'automate sous forme implicite (i.e. via des systèmes d'équations) et permet ainsi la vérification dites "symbolique" qui ne présente pas ces problèmes.

VI. Conclusion

Le paradigme *LET* permet ainsi d'assurer déterminisme et concurrence tout en permettant une compilation simple et une analyse d'ordonançabilité fine. Toutefois, aujourd'hui il n'existe pas réellement d'approche de vérification formelle des langages implémentant ce paradigme. Les modèles synchrones peuvent être utilisés pour abstraire les langages basés sur le *LET* et les techniques de vérification actuelles peuvent être réutilisées. Dans l'exemple utilisé dans cet article, une analyse d'accessibilité de signaux d'erreurs est effectué sur l'automate (explicite) généré par le compilateur d'*Esterel*. La taille de celui-ci pouvant grossir rapidement, une perspective est l'utilisation de représentations implicites d'automate ce qui permettrait de traiter des applications de tailles supérieures. D'autre part, la comparaison entre synchrone et *LET* sera développée plus précisément dans un travail futur.

Références

[1] P. Raymond, "Synchronous Program Verification with Lustre/Lesar," 2010.

[2] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, pp. 1270 – 1282, 10 1991.

[3] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre : A declarative language for programming synchronous systems," in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, vol. 178, 1987, p. 188.

[4] F. Boussinot and R. De Simone, "The esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.

[5] F. Boniol, "CoReA : A synchronous calculus of parallel communicating reactive automata," in *PARLE'94 Parallel Architectures and Languages Europe*. Springer, 1994.

[6] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical computer science*, vol. 25, no. 3, pp. 267–310, 1983.

[7] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A Multi-Periodic Synchronous Data-Flow Language," in *11th IEEE High Assurance Systems Engineering Symposium*, Dec. 2008, p. 251.

[8] F. Boussinot and R. De Simone, "The sl synchronous language," *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 256–266, 1996.

[9] C. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer Berlin Heidelberg, 10 2012, pp. 103–120.

[10] E. A. Lee and S. A. Seshia, *Introduction to embedded systems : a cyber-physical systems approach*, second edition ed. Cambridge, Massachuetts : MIT Press, 2017.

[11] H. Kopetz, *Real-Time Systems : Design Principles for Distributed Embedded Applications*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[12] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto : a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[13] A. Ghosal, T. Henzinger, C. Kirsch, and M. Sanvido, "Event-driven programming with logical execution times," in *Hybrid Systems : Computation and Control*, vol. 2993, 03 2004, pp. 357–371.

[14] D. Chabrol, G. Vidal-Naquet, V. David, C. Aussagues, and S. Louise, "Oasis : A chain of development for safety-critical embedded real-time systems," 01 2004.

[15] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[16] A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. Della Monica, and A. Ingólfsdóttir, "A foundation for runtime monitoring," in *International Conference on Runtime Verification*. Springer, 2017, pp. 8–29.