Modélisation et validation formelle de systèmes globalement asynchrones et localement synchrones

Fatma Jebali, Mouna Tka Mnad, Christophe Deleuze, Frédéric Lang, Radu Mateescu, Ioannis Parissis

▶ **To cite this version:**

Fatma Jebali, Mouna Tka Mnad, Christophe Deleuze, Frédéric Lang, Radu Mateescu, et al.. Modélisation et validation formelle de systèmes globalement asynchrones et localement synchrones. Approches Formelles dans l'Assistance au Développement de Logiciels, Jun 2014, Paris, France. pp.97–102. hal-01007674

HAL Id: hal-01007674

https://hal.univ-grenoble-alpes.fr/hal-01007674

Submitted on 17 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modélisation et validation formelle de systèmes globalement asynchrones et localement synchrones

Fatma Jebali[2,1,3], Mouna Tka[1,4], Christophe Deleuze[1,4],
Frédéric Lang[2,1,3], Radu Mateescu[2,1,3], Ioannis Parissis[1,4]

[1] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2] Inria
[3] CNRS, LIG, F-38000 Grenoble, France
[4] Univ. Grenoble Alpes, LCIS, F-26000 Valence, France

Résumé

Les automatismes industriels et domestiques sont fréquemment mis en oeuvre au moyen de contrôleurs logiques programmables (CLP), qui exécutent en mode synchrone des applications embarquées interagissant avec leur environnement. En combinant plusieurs CLP qui opèrent indépendamment et communiquent à travers un réseau, il est possible de réaliser des automatismes plus élaborés, de type GALS (Globally Asynchronous, Locally Synchronous). Pour assurer une conception correcte des systèmes GALS, qui est difficile à cause de la présence simultanée des aspects synchrones et asynchrones, nous proposons dans cet article une méthodologie rigoureuse, basée sur des méthodes formelles et des techniques de validation automatique (test et vérification) issues des paradigmes synchrone et asynchrone.

1 Introduction

La mise en oeuvre classique des automatismes industriels et domestiques repose sur l'utilisation de contrôleurs logiques programmables (CLP), qui exécutent en mode synchrone des applications embarquées interagissant avec leur environnement (capteurs et actionneurs). Des automatismes plus élaborés peuvent être réalisés au moyen de plusieurs CLP qui opèrent indépendamment et communiquent de manière asynchrone à travers un réseau. Ces systèmes, qui appartiennent à la classe des GALS (*Globally Asynchronous, Locally Synchronous*) [2], combinent des aspects synchrones et asynchrones, ce qui rend leur conception difficile à cause du non-déterminisme des communications.

Dans cet article, nous proposons une méthodologie de conception rigoureuse des systèmes GALS, à base de méthodes formelles et de techniques de validation

automatique issues des paradigmes synchrone et asynchrone. Au niveau synchrone, la validation des CLP individuels est effectuée par génération de tests destinés à couvrir le comportement des applications embarquées. Au niveau asynchrone, les réseaux de CLP sont modélisés au moyen d'un langage pivot dédié à la description des GALS, qui est traduit ensuite vers un langage formel asynchrone équipé d'outils de vérification par équivalences et logiques temporelles. Les deux types de validation sont interdépendants : les scénarios de tests décrits au niveau synchrone peuvent être rejoués sur le modèle asynchrone (projeté sur les CLP individuels), et des séquences peuvent être générées automatiquement à partir du modèle asynchrone afin d'alimenter la génération de tests au niveau synchrone.

2 Validation synchrone

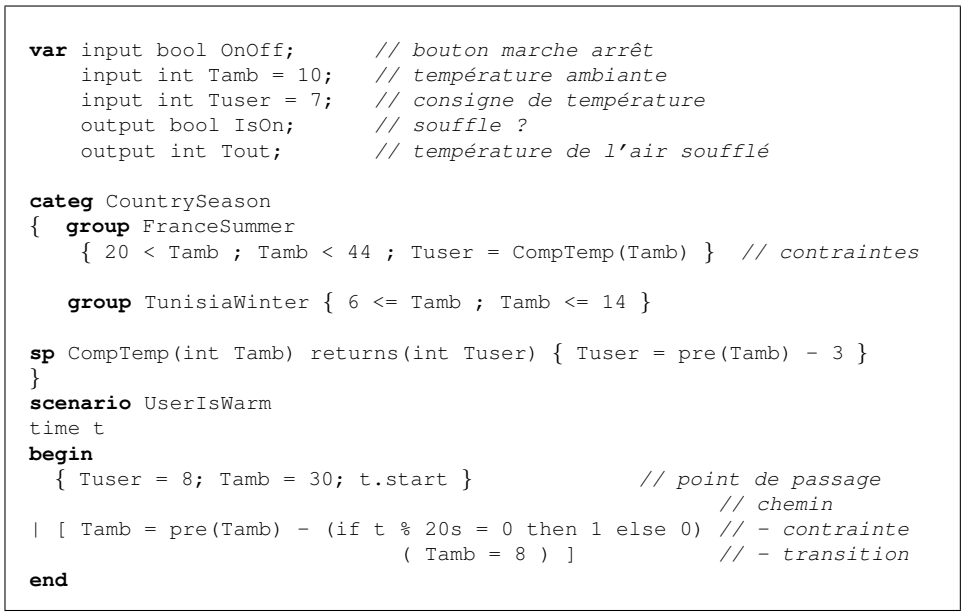
Dans le cadre du test d'un système synchrone, nous souhaitons offrir la possibilité aux concepteurs de spécifier des tests de telle sorte que leur génération soit automatisée. De manière similaire à des approches antérieures sur le test synchrone [5, 6], un générateur de données de test est branché au système et, en s'appuyant sur les spécifications de test fournies, génère des valeurs pour les entrées du système, puis observe les sorties produites. En fonction de ces dernières, il procède à une nouvelle génération de valeurs d'entrée et ainsi de suite. Un oracle peut vérifier la conformité des sorties par rapport aux spécifications du système.

Nous proposons un nouveau langage, SPTL (*Synchronous Programs Testing Language*), pour spécifier les tests. SPTL permet de définir des modèles de test décrivant des *contraintes* sur le comportement de l'environnement et des *scénarios* de test conformes à ces contraintes. SPTL a pour objectif d'être utilisable par des non spécialistes du test. Il adopte le paradigme *flot de données*, comme beaucoup de langages de programmation d'automates. Les entrées et sorties du système sont représentées par des variables typées. Un ensemble de contraintes relient ces variables : elles définissent les valeurs possibles pour les variables d'entrée en fonction des valeurs passées des variables d'entrée et de sortie.

Une spécification SPTL contient des déclarations de variables, des contraintes réparties en *groupes* et *catégories* (que nous discutons plus loin) et des scénarios. La figure 1 montre un exemple de spécification pour un système de régulation de climatisation. Les contraintes peuvent utiliser les opérateurs arithmétiques et logiques, ainsi que deux opérateurs spécifiques :

- `prob(`e`,`p`)` est une expression dont la valeur est e avec la probabilité p, et une autre valeur (parmi les valeurs possibles pour le type) sinon,
- `pre(`v`)` est la valeur de la variable (ou plus généralement expression) v au cycle précédent. La valeur utilisée lors du premier cycle est donnée dans la déclaration de la variable (cas de `Tamb` ici).

Des sous-programmes (`CompTemp` dans la figure) permettent de factoriser des expressions complexes pouvant apparaître dans les contraintes.



```
var input bool OnOff;      // bouton marche arrêt
    input int Tamb = 10;   // température ambiante
    input int Tuser = 7;   // consigne de température
    output bool IsOn;      // souffle ?
    output int Tout;       // température de l'air soufflé

categ CountrySeason
{  group FranceSummer
    { 20 < Tamb ; Tamb < 44 ; Tuser = CompTemp(Tamb) }  // contraintes

   group TunisiaWinter { 6 <= Tamb ; Tamb <= 14 }

sp CompTemp(int Tamb) returns(int Tuser) { Tuser = pre(Tamb) - 3 }
}
scenario UserIsWarm
time t
begin
  { Tuser = 8; Tamb = 30; t.start }             // point de passage
                                                        // chemin
| [ Tamb = pre(Tamb) - (if t % 20s = 0 then 1 else 0) // - contrainte
                            ( Tamb = 8 ) ]              // - transition
end
```

FIGURE 1 – Exemple de spécification SPTL (extrait d'un système de climatisation)

Profil Un système peut être utilisé dans des environnements variés. Décrire l'environnement le plus général reviendrait à fortement sous-spécifier le test. SPTL introduit la notion de *profil*, qui permet de décliner l'environnement en plusieurs variantes. Pour le système de climatisation, on pourrait avoir un profil correspondant à une habitation individuelle l'hiver en Tunisie, un autre correspondant à un bâtiment public l'été en France, chacun exprimant des contraintes différentes. Pour simplifier le travail d'élaboration des profils, SPTL permet de découper l'environnement en un petit nombre de *catégories*. Pour chaque catégorie, on définit un ensemble de *groupes*, chacun contenant des contraintes déterminées par cet aspect de l'environnement. Dans notre exemple, on a défini une catégorie `CountrySeason` avec deux groupes `FranceSummer` et `TunisiaWinter`. Une autre catégorie pourrait être le lieu d'utilisation (batiment public ou résidence individuelle).

Scénario Un scénario permet de spécifier une évolution de l'environnement dans le temps, de façon à amener le système dans un état précis. Il peut par exemple représenter une séquence d'actions d'un utilisateur. Un scénario s'exécute dans le cadre d'un profil et spécifie un ensemble de contraintes supplémentaires qui varient au cours de l'exécution. Un scénario est une séquence comprenant deux types d'éléments : des points de passage et des chemins. Un point de passage indique un ensemble de contraintes qui doivent s'appliquer sur un cycle de l'exécution. Un chemin comprend un ensemble de contraintes et une condition de transition.

Les contraintes s'appliquent à chaque cycle jusqu'à ce que la condition de transition, calculée à la fin de chaque cycle, soit vraie. À ce moment le scénario passe à l'étape suivante. Les scénarios permettent aussi de prendre en compte l'écoulement du temps avec des variables de type `time`. Ces variables sont des chronomètres qui peuvent être déclenchés dans une étape ponctuelle, et utilisés dans contraintes. Ceci permet d'utiliser le temps dans le déroulement du scénario (dans notre exemple, `Tamb` est décrémentée toutes les 20 secondes).

3 Validation asynchrone

Nous proposons un langage textuel formel, nommé GRL (*GALS Representation Language*) [4], pour la description des systèmes GALS composés de plusieurs composants synchrones en interaction permanente avec leur environnement et communiquant via des médiums de communication asynchrones. Un programme GRL est structuré en plusieurs types d'entités :

- les *types*, prédéfinis (entiers, booléens, etc.) ou définis par l'utilisateur (énumérés, intervalles, structures)
- les *blocs* (figure 2), qui constituent les briques d'exécution synchrones
- les *médiums* (figure 3) et les *environnements*, qui modélisent respectivement le réseau et des contraintes physiques ou logiques sur les blocs, et dont l'exécution est guidée par l'interaction avec les blocs
- les *systèmes* (figure 4), qui décrivent l'assemblage et les interactions entre blocs, médiums et environnements
- les *modules*, qui structurent l'ensemble et permettent de construire des bibliothèques réutilisables

Pour les blocs synchrones, GRL s'inspire des diagrammes de flot de données. Chaque bloc est décrit sous la forme d'un programme déterministe qui combine les structures de contrôle classiques des langages algorithmiques (affectations, if-then-else, case, etc.) et les instanciations hiérarchiques de blocs. Ce programme déterministe définit une boucle synchrone dont l'exécution est atomique : (1) lecture des entrées (2) réception de données du réseau (3) calcul des sorties (code impératif déterministe) (4) envoi de données au réseau (5) écriture des sorties.

Chaque instance de bloc dispose de sa propre mémoire, constituée de variables temporaires (dont les valeurs sont locales à un cycle d'exécution de la boucle synchrone) ou permanentes (dont les valeurs, qui persistent entre les cycles d'exécution de la boucle synchrone, définissent l'état courant du bloc).

Par rapport aux blocs, les médiums et les environnements s'exécutent de manière "passive", c'est-à-dire que leur comportement est guidé par l'occurrence de signaux induits (implicitement) par la réception ou l'envoi de données par un bloc (médiums) ou par la lecture ou l'écriture d'une entrée/sortie (environnements). De plus, ce comportement est non-déterministe, ce qui permet de modéliser des comportements complexes au bon niveau d'abstraction.

```
block AileronSystem (in spo : bool; out cpo : nat)
                    {receive lock, up, down : bool; send apo : nat} is
  perm pos : nat := 0

  if not(lock) and spo then
      if up then pos := pos + 1 elsif down then pos := pos − 1 end if
  end if;
  cpo := pos; apo := pos
end block
```

FIGURE 2 – Exemple de bloc GRL (extrait d'un système de contrôle de vol)

```
medium AccessScheduler {receive apo : nat | send lock, up, down : bool |
            receive lp, up, dp : bool | send app : nat |
            receive ls, us, ds : bool | send aps:nat } is

  perm lock_buff : bool := true; up_buff, down_buff : bool := false, apo_buff : nat := 0
  select
      on lp, up, dp  →  lock_buff := lp; up_buff := up; down_buff := dp
  []
      on ls, us, ds  →  lock_buff := ls; up_buff := us; down_buff := ds
  []
      on apo  →  apo_buff := apo
  []
      on ?app  →  app := apo_buff
  []
      on ?aps  →  aps := apo_buff
  []
      on ?lock, ?up, ?down  →  lock := lock_buff; up := up_buff; down := down_buff
  end select
end medium
```

FIGURE 3 – Exemple de médium GRL (extrait d'un système de contrôle de vol)

Syntaxiquement, GRL est inspiré du langage LNT [1], lui-même issu de la lignée des algèbres de processus, étendu avec des données et les structures de contrôle classiques des langages de programmation algorithmiques. D'une part, ce choix nous offre expressivité et concision pour la description des aspects synchrones et asynchrones et, d'autre part, il permet d'envisager à moindre coût des passerelles entre les modèles écrits en GRL et les outils de vérification de systèmes concurrents asynchrones diffusés au sein de la boîte à outils CADP [3], qui utilisent LNT comme langage d'entrée.

4 Conclusion et travaux en cours

La méthodologie proposée pour modéliser et analyser formellement des systèmes GALS est destinée à être intégrée dans un flot de conception industriel des automatismes à base de réseaux de PLC. Le langage GRL vise à établir une passerelle entre les diagrammes de flot de données décrivant les applications em-

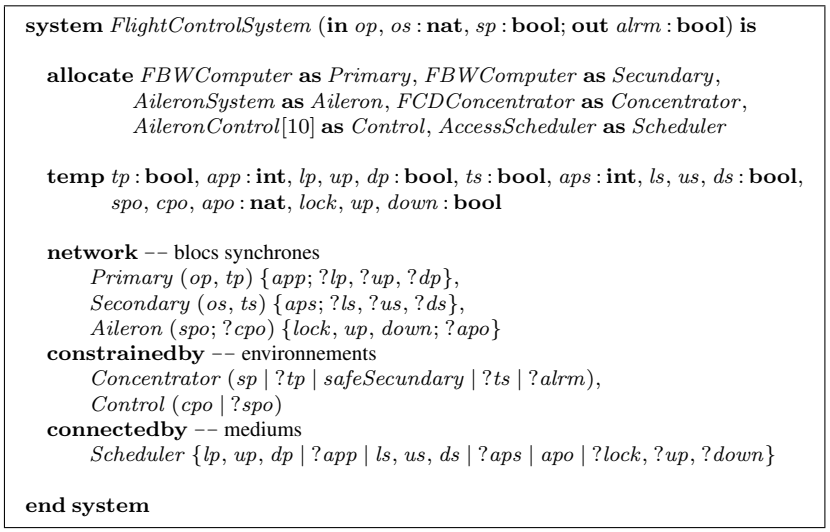


FIGURE 4 – Exemple de système GRL (extrait d'un système de contrôle de vol)

barquées sur les PLC individuels et la description formelle du comportement des PLC connectés en réseau. Une première connexion avec des outils de validation formelle sera obtenue par traduction du langage GRL vers LNT. A plus long terme, les fonctionnalités de validation seront déployées comme services sur des nuages de calcul, afin de promouvoir une approche de conception rigoureuse d'automatismes dans l'internet des objets.

Références

[1] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (version 5.4). INRIA/VASY, 149 pages, September 2011.

[2] D. M. Chapiro. *Globally-asynchronous Locally-synchronous Systems*. PhD thesis, Stanford, CA, USA, 1985.

[3] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011 : A toolbox for the construction and analysis of distributed processes. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 15(2) :89–107, 2013.

[4] F. Jebali, F. Lang, and R. Mateescu. GRL : A specification language for globally asynchronous locally synchronous systems. Research Report RR-8527, Inria, 2014. `http://hal.inria.fr/hal-00983711`.

[5] L. Madani, V. Papailiopoulou, and I. Parissis. Towards a testing methodology for reactive systems : A case study of a landing gear controller. In *ICST*, pages 489–497, 2010.

[6] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *RTSS*, pages 200–209, 1998.