Réécriture d'un modèle MCSE à l'aide du langage FIACRE

Clément Filleau, Pierre-Emmanuel Hladik

Réécriture d'un modèle MCSE à l'aide du langage FIACRE

Clément Filleau[1] *et Pierre-Emmanuel Hladik*[1,2]

[1] *CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France*

[2] *Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France*

—

11 octobre 2013

Projet ANR RESPECTED, programme ARPEGE

Résumé

Le travail présenté dans ce document s'inscrit dans une démarche de conception des systèmes embarqués. Il se focalise sur les règles de réécriture d'un modèle exprimé en MCSE dans le langage FIACRE. Cette réécriture vise à permettre une vérification formelle des propriétés du modèle tout en garantissant le respect de la sémantique du langage de haut-niveau MCSE. Les règles exposées couvrent la totalité de l'expressivité du modèle d'entrée et sont illustrées sur des exemples simples.

1 Introduction

Afin de maîtriser la complexité des systèmes embarqués, de nombreuses méthodologies proposent des outils pour passer du cahier des charges à une solution opérationnelle. Ces méthodes, dont MCSE (Méthodologie de Conception des Systèmes Embarqués) fait partie, requièrent un grand nombre d'étapes qui peuvent être complexes. Elles permettre d'améliorer la rapidité de la démarche, la qualité des systèmes créés et la compréhension au sein des différentes équipes travaillant sur un même projet. Cependant, il ne s'agit là que de méthodes, ce qui n'évite pas les erreurs de conception et ne garantit donc en rien le bon fonctionnement du système complet. Erreurs intrinsèques au système, problèmes d'interactions entre les différentes fonctions, contraintes temporelles non respectées, ne sont que quelques exemples parmi tous les problèmes pouvant intervenir au sein d'un système embarqué. Il apparaît donc un réel besoin de vérifier ces systèmes et d'assurer un fonctionnement strictement adapté à l'application.

Le travail présenté dans ce document s'inscrit dans ce contexte et propose une méthode de réécriture d'un modèle de conception MCSE avec le langage formel FIACRE. Ce nouveau modèle peut ainsi être vérifié par des outils dédiés afin d'en faire la vérification sur des propriétés comportementales. Ce travail impose de s'assurer que l'ensemble des propriétés du système original sont conservées lors de la transformation.

Les deux premières parties de ce rapport présentent la méthodologie MCSE et le langage FIACRE. La troisième partie introduit la transformation de MCSE en FIACRE à l'aide d'exemples simples.

Ce travail a été mené dans le cadre du projet ANR RESPECTED, program ARPEGE.

2 Présentation de la méthodologie MCSE

La méthodologie MCSE (Méthodologie de Conception des Systèmes Embarqués) est utilisée pour la spécification, la conception et la réalisation des systèmes embarqués. Sa spécification offre des modèles et des méthodes pour concevoir de tels systèmes. Elle est principalement utilisée pour réaliser des systèmes dédiés, c'est-à-dire de petite taille et devant être réalisés sur une période courte.

Cette partie fait référence au livre de J.-P. Calvez *Spécification et conception des systèmes : une méthodologie* [5]. Nous ne présentons pas ici la méthodologie, mais uniquement le modèle utilisé lors de la conception d'un système. Ce modèle servira d'entrée à la réécriture en langage FIACRE du système.

Le modèle MCSE est décomposé en deux niveaux :
– le niveau fonctionnel qui représente les fonctionnalités du système à un niveau d'abstraction plus ou moins élevé,
– le niveau comportemental qui décrit de manière séquentielle les activités de chaque fonction.

2.1 Le modèle fonctionnel

2.1.1 Les fonctions

Un modèle fonctionnel décrit, à l'aide d'une organisation hiérarchique, un système par des sous-fonctions connectées entre elles par des moyens de communication. Il donne une vision globale du système grâce à sa représentation schématique claire, dont les principaux éléments sont présentés ci-dessous. La figure 1 fournit un exemple trivial de système de type producteur-consommateur modélisé en MCSE.

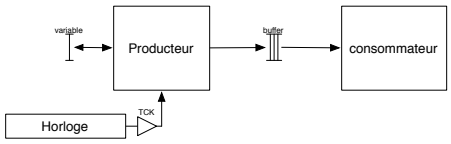


Fig. 1: Exemple simple d'un système en MCSE

Chaque fonctionnalité du système est représentée par un bloc rectangulaire. Le nom donné au bloc explicite le rôle de la fonction au sein de l'ensemble auquel elle appartient. La composition hiérarchique de fonctions est possible. Plusieurs fonctions peuvent

être regroupées pour former une macro-fonction. Grâce à ce mécanisme, le système peut être rafiné à différents niveaux. Un système est entièrement modélisé lorsque chacune des fonctions est décrite de façon séquentielle (voir 2.1.3).

2.1.2 Synchronisation et échange de données

MCSE propose trois moyens de synchroniser ou d'échanger des données entre fonctions. La première se présente sous la forme d'une **variable d'état** (représentée par la figure 2) accessible par plusieurs fonctions en lecture et en écriture. Cet accès exclue les accès simultanés à la variable en considérant que les opérations de lecture et d'écriture sont atomiques : l'accés à la variable est bloqué pendant l'utilisation de cette dernière par la fonction.

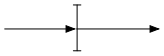


Fig. 2: Représentation MCSE d'une variable d'état

Le deuxième moyen de communication est un transfert d'information par l'intermédiaire de **buffers** (figure 3). Il s'agit d'un échange de messages entre fonctions avec une relation de type producteur-consommateur. La taille des buffers est supposée infinie : la file de messages n'est jamais pleine et il est toujours possible de venir lire ou écrire une donnée. Les accès au buffer se font de manière totalement asynchrone, c'est-à-dire que l'écriture et la lecture peut se faire à tout moment et n'est pas bloquante.

Nous verrons dans les parties suivantes que nous ne respectons pas ces hypothèses du modèle MCSE, mais que les buffers sont de taille finie et qu'ils sont bloquants en écriture si la file est pleine et en lecture si la file est vide.

En pratique, le cas d'une file pleine en écritre est un cas d'erreur et l'application doit réagir en conséquence. Nous ferons donc l'hypothèse que ce cas est toujours pris en considération.

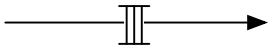


Fig. 3: Représentation MCSE d'un buffer

Enfin, le troisième moyen pour synchroniser les fonctions du système se fait au moyen d'une **synchronisation par événement** (figure 4.2.3). Une fonction productrice de l'évènement de synchronisation est reliée à une ou plusieurs fonctions synchrones. L'évènement produit est diffusé vers l'ensemble des fonctions reliées qui peuvent, à compter de la réception du signal, débuter ou continuer leur exécution. Le modèle MCSE spécifie simplement que toutes les fonctions en réception d'un événement sont sensibilisés. Pour offrir une sémantique plus stricte du comportement de MCSE, nous dirons que toutes les transitions (voi ci-après) en attentes de la synchronisation sont sensibilisées. De plus, les événements ne sont pas comptés, c'est-à-dire que si un événement se produit plusieurs fois avant d'être consommé, le nombre d'occurrences n'est pas mémorisé.

En pratique, la diffusion d'événement à plusieurs récepteur n'est pas autorisé et seul les événement 1-à-1 sont utilisés.

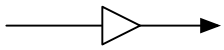


Fig. 4: Représentation MCSE d'un synchronisation par événement

2.1.3 Le modèle comportemental

Le modèle comportemental décrit l'exécution d'une fonction élémentaire d'un système, c'est-à-dire une fonction dont le comportement est entièrement décomposée de manière séquentielle. Afin de décrire le comportement des fonctions, nous utilisons une représentation graphique (proposée par See4Sys) proche de l'annexe comportementale de AADL. La figure 5 fournit le comportement pour la fonction producteur de l'exemple précédent. Les principaux éléments graphiques sont décrits ci-dessous.

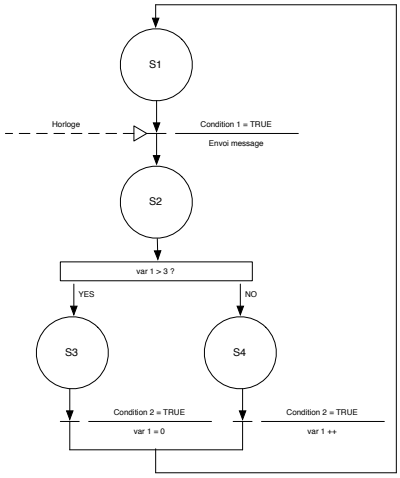
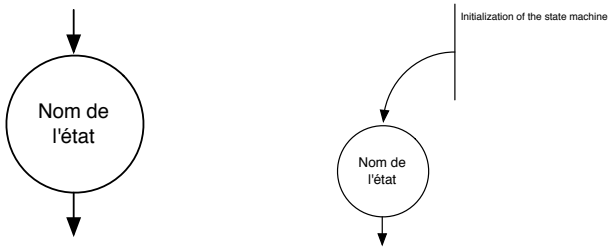


Fig. 5: Modèle comportemental de la fonction producteur

État : décrit l'état d'une fonction et est représentée par un cercle (figure 6(a)). Un état initial doit être spécifié pour chaque fonction et est noté à l'aide du symbole de la figure 6(b))



(a) Un état (b) État initial

Transition bloquante : décrit la transition d'un état source à un état destination. La transition est gardée par une condition booléenne et des actions peuvent y être attachées. Ces actions peuvent être un appel à un sous-programme, l'envoi d'un message ou une affectation de valeurs. Dans le cas de l'envoi d'un message par buffer, l'écriture n'est jamais bloquante. Par contre, une erreur est automatiquement produite s'il y a une tentative d'écriture dans un buffer plein. Les actions sont attachées à la transition et non à l'état : si

une transition est sensibilisée, les actions sont réalisées et l'état courant devient l'état de destination.

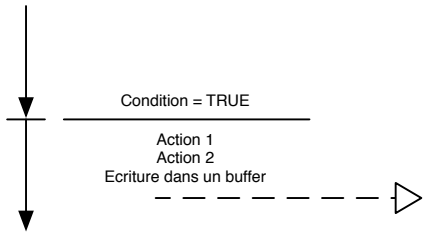


Fig. 6: Test bloquant

Transition événementielle bloquante : identique à la transition bloquante avec action, mais la garde peut aussi comprendre un événement. La transition est sensibilisée si le test est vrai et si l'événement a été produit. L'événement est consommé quand la transition est franchie. Dans le cas d'une attente sur un événement sur une communication par buffer, la lecture ainsi que la libération d'une place du buffer est immédiate et atomique.

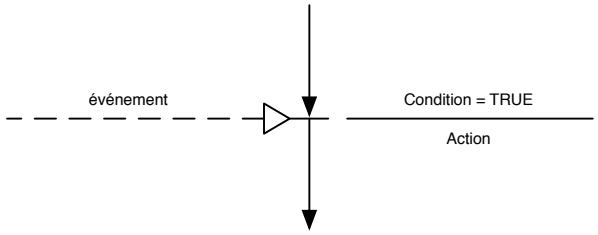


Fig. 7: Test bloquant sur événement

Transition non bloquante avec test conditionnel : décrit la transition d'un état source vers plusieurs états destinataires. En fonction de la condition un seul état destinataire devient l'état courant.

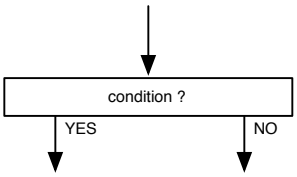


Fig. 8: Test conditionnel

Transition non bloquante à choix multiples : identique au test conditionnel, mais le choix de prochain état courant dépend d'un ensemble de valeurs.

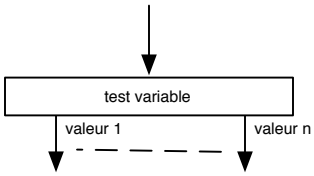


Fig. 9: Test à choix multiple

2.1.4 Gestion du temps

Certaines fonctions peuvent être périodiques. Dans le modèle MCSE initial cette périodicité est modélisée par une fonction dédiée qui génère un évènement périodiquement sur lequel se synchronise la fonction périodique. Nous utiliserons cette représentation, tout en distinguant clairement les fonctions de traitement de celles dédiées aux horloges.

Une évolution souhaitable du langage MCSE serait de faire apparaître clairement des blocs Horloges qui ont uniquement pour but de produire des événements périodiques que l'on pourrait éventuellement suspendre.

Aucune annotation n'est prévue dans le modèle pour apporter des informations sur le temps consommé par les actions. Nous ajoutons donc dans le modèle des informations temporelles sur ces actions sous la forme d'annotations.

3 Le langage FIACRE

Cette partie présente succinctement le langage FIACRE [4] ainsi que l'outillage existant autour. Pour une présentation détaillée le lecteur peut consulter la documentation en ligne de FIACRE [1].

La démarche que nous suivons est similaire à celle exposée dans [3] où une réécriture de modèle AADL (Architecture Analysis and Design Language) vers FIACRE a été réalisée. Il est intéressant de noter les motivations des travaux qui ont conduits à la définition de FIACRE et qui restent toujours valables dans le cadre de notre étude :

- besoin de créer un langage de vérification fonctionnelle des systèmes,
- besoin de traduire les langages de modélisation des systèmes dans ce langage de vérification pour assurer un fonctionnement sûr et maîtrisé des systèmes critiques,
- disposé d'un langage pivot de vérification comme entrée pour différents langages de modélisation.

3.1 Description du langage FIACRE

Le langage FIACRE est un langage formel de modélisation de systèmes qui permet leur simulation et leur vérification comportementale. La syntaxe est proche des langages de programmation haut niveau. La description des systèmes en langage FIACRE se veut facile à appréhender. Nous y retrouvons de nombreux points communs, notamment avec le langage C, tels que :

- la présence de tests conditionnels (« *if ... then ... else* », « *case ... of* »),
- la description séquentielle des actions,
- des processus et composants avec passages de paramètres,
- la structuration du code (en-tête de fonction, déclarations, actions).

Les fonctions primaires d'un système sont décrites en FIACRE par l'intermédiaire de processus. Les processus sont composés, à la manière des réseaux de Petri, d'états et de conditions d'évolution entre les états, l'ensemble des tests et des actions étant évalué de manière séquentielle. De plus, les états des processus sont atomiques. des indications temporelles peuvent être aussi spécifiée.

Les fonctions primaires peuvent être regrouper sous forme de composants. De même, plusieurs composants peuvent être assemblés entre eux, assurant ainsi une décomposition hiérarchique. Il est impératif d'indiquer, à chaque regroupement de fonctions ou de composants, les liens de communication qui les relient entre eux.

Il existe de trois types de communication entre processus :

– les synchronisation par port : relation bloquante de toutes les fonctions reliées au port jusqu'à ce qu'elles soient toutes prêtes à recevoir le signal de synchronisation pour continuer leur évolution,
– les variables partagées qui peuvent êtres lues ou modifiées de manière asynchrone : il est à noter que l'éxécution atomique d'un état empêche deux fonctions d'accéder simultanément à une même variable,
– les buffers qui sont un regroupement de dimension réglable de plusieurs variables de même type : chaque buffer peut être accédé de manière asynchrone en lecture ou en écriture en vérifiant impérativement qu'il n'est pas vide ou plein selon le cas d'utilisation.

Afin de faire la vérification d'un modèle en FIACRE, celui-ci est traduit en réseau de Petri avec franchissement conditionnel. De plus, de la glue syntaxique est ajoutée pour synchronisée toutes les machines à états.

3.2 Exemple

Afin d'illustrer l'utilisation du langage FIACRE, un exemple simple est présenté ici. Il s'agit d'un système possédant deux fonctions avec un accès partagé à une variable. Les deux fonctions élémentaires sont décrites à l'aide de processus et admettent un paramètre *variable* de type booléen (ligne 34). Leur contenu est trivial : elles complémentent la valeur de *variable* (lignes 5 à 11 et 21 à 27) puis attendent un temps compris entre 4 et 10 (lignes 14 et 30). Pour modéliser le système complet, un composant est créé. Il implémente les deux processus et possède une même variable interne booléenne qui est passée en paramètre aux deux fonctions : c'est la variable partagée. Le code FIACRE et le réseau de Petri correspondant sont donnés par le listing 1 et la figure 10. Le fichier .c décrivant l'évolution du réseau de Petri n'est pas fourni, mais fait partie intégrante du modèle.

3.3 Conception des modèles de vérification

Le processus de vérification du modèle d'un système est présenté par la figure 11. Le système à vérifier est dans un premier temps décrit dans le langage FIACRE. Cette description se veut comportementale, c'est-à-dire que seule l'évolution dynamique de l'état du système est représentée. L'outil `frac` compile ce code dans le format Time Transition System (TTS). Une fois ce fichier obtenu, l'utilisation de la boîte à outils TINA (voir 3.4) permet d'en faire la vérification.

Listing 1: Un exemple de code FIACRE

```
process PROC1 (&variable : read write bool) is // En-tête du processus PROC1
  states Etat1, Etat2     // Déclaration des états du processus

  from Etat1     // Comportement du processus
    if variable = true then
      variable := false;
      to Etat2
    else
      variable := true;
      to Etat2
    end

  from Etat2
    wait[4,10];
    to Etat1

process PROC2 (&variable : read write bool) is // En-tête du processus PROC2
  states Etat1, Etat2

  from Etat1
    if variable = true then
      variable := false;
      to Etat2
    else
      variable := true;
      to Etat2
    end

  from Etat2
    wait[4,10];
    to Etat1

component Main is // Déclaration du système
  var    VAR : bool := false  // Déclaration de la variable partagée

  par   PROC1(&VAR)     // Composition des composants
     || PROC2(&VAR)
  end
```

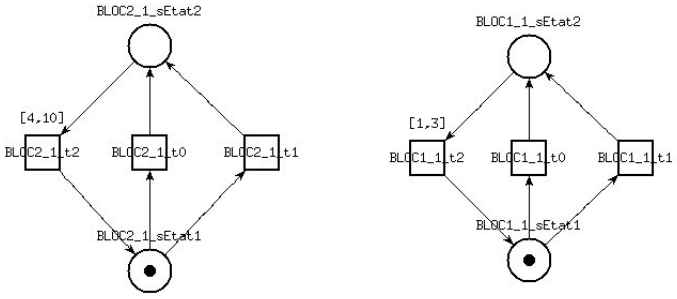


Fig. 10: Réseau de Petri généré à partir du code FIACRE

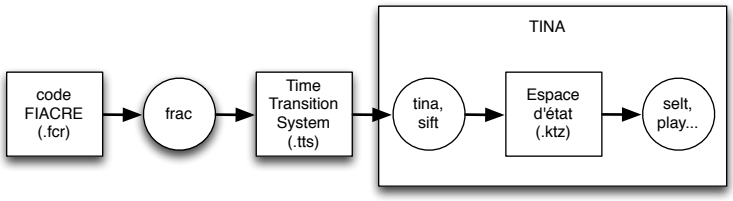


Fig. 11: Chaîne de vérification pour un modèle FIACRE avec la boîte à outils TINA

3.4 Boîte à outils TINA

La boîte à outils TINA [2] permet de model-checker des propriétés sur un système décrit par un Time Transition System (TTS). Elle offre la possibilité de tester les spécificités structurelles et comportementales d'un système afin de délimiter ses frontières d'utilisation. La liste ci-dessous explicite les principaux outils disponibles et leur utilité :

- `tina` permet de générer l'espace d'état du modèle,
- `sift` est analogue à tina avec moins d'options, mais permettant de considérer des espaces d'états plus grands et de manière plus efficace,
- `selt` vérifie les propriétés LTL sur un modèle,
- `struct` analyse la structure du réseau de Petri généré (présence d'invariants, de flows ou semiflows),
- `pathto` recherche les chemins d'un état initial vers un état final définis par l'utilisateur,
- `play` permet de faire évoluer le réseau de Petri pas à pas pour étudier manuellement son évolution.

4 Réécriture d'un modèle MCSE en FIACRE

Cette partie expose point par point les règles de réécriture d'un modèle MCSE dans le langage FIACRE.

4.1 Transformation du comportement d'une fonction élémentaire

Le comportement d'une fonction élémentaire est réécrite en FIACRE en suivant les règles suivantes :

- la fonction MCSE est représentée par un *processus* FIACRE,
- chaque état du comportement devient un état (*state*) en FIACRE,
- les actions de type affectation sont directement codées en FIACRE,
- les transitions bloquantes sont écrites à l'aide d'une structure « *if ... then ...* »,
- les transitions événementielle bloquante utilisent mot-clef « *on* » en FIACRE,
- les transitions non bloquantes avec tests conditionnels ont la structre « *if ... then ... else ...* »,
- les transitions non bloquantes à choix multiples utilisent leur structure « *case ... of ...* » de FIACRE.

Enfin, la composition est naturellement exprimée à l'aide des composants (*component*) du langage FIACRE qui permettent de regrouper plusieurs processus dans un même composant.

4.2 Réécriture des échanges de données

4.2.1 Les variables partagées

La réécriture des variables partagées se fait sans difficulté en FIACRE en déclarant une variable accessible par les processus. Dans les deux langages, la variable est accessible en lecture et/ou en écriture et l'accès est atomique et asyncrhone.

Il est cependant nécessaire de faire attention à l'utilisation de variables de types compteur. Une borne sur leur plage d'utilisation est impératif sous peine de créer un nombre de marquages infinis.

La figure 12 présente un modèle MCSE et le listing 2 son code FIACRE. Cet exemple consiste simplement en un système trivial d'écrivain-lecteur sur une variable partagée. L'écriture (ligne 29) et la lecture (ligne 11) se font de manière asynchrone.

Listing 2: Un exemple de variable partagée en FIACRE

```
type T is union // Déclaration du type de la variable partagée
    ETAT1
  | ETAT2
  | ETAT3
end

process LECTEUR (&produit : read T) is

  states Attente, Etat1, Etat2, Etat3
  from L1
    case produit of    // Test sur la valeur de la variable
          ETAT1 → to Etat1
        | ETAT2 → to Etat2
        | ETAT3 → to Etat3
    end

  from L2
    to L1

  from L3
    to L1

  from L4
    to L1

process ECRIVAIN (&produit : read write T) is
  states Attente, Attente2, Attente3
  from E1
    produit := ETAT1;    // Ecritures dans la variable partagée
    to E2

  from E2
    produit := ETAT2;
    to E1

component Main is
  var    tmp_produit : T := ETAT1     // Initialisation de la variable

  par     LECTEUR(&tmp_produit)
       || ECRIVAIN(&tmp_produit)
  end
Main
```

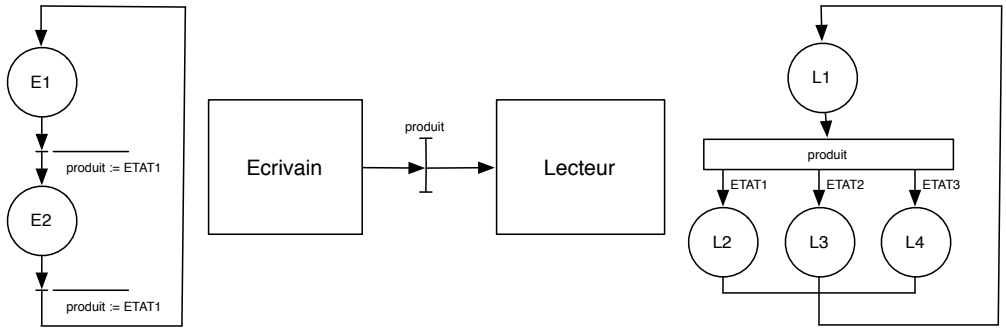


Fig. 12: Modèle MCSE de l'essai sur les variables partagées

4.2.2 Les buffers

Pour décrire un buffer en FIACRE nous utilisons une *queue*. Comme présenté dans la partie sur le modèle MCES, la principale différence entre le buffer MCSE et la file en FIACRE est que cette dernière a une taille bornée. Il est donc nécessaire de le prendre en considération.

L'hypothèse de MCSE que les buffers sont de taille infinie n'est pas réaliste. Il serait pertinent d'introduire dans le modèle MCSE la taille du buffer et de définir clairement le protocole d'écriture et de lecture. Nous supposerons ici que l'écriture produit une erreur si la queue est pleine et que la lecture est bloquante si la queue est vide. Notons que l'écriture dans un buffer plein provoque « naturellement » une erreur en FIACRE.

La figure 13 et le listing 3 montrent respectivement le modèle MCSE et le code FIACRE d'un système de producteur consommateur avec un buffer de taille 3 (ligne 1).

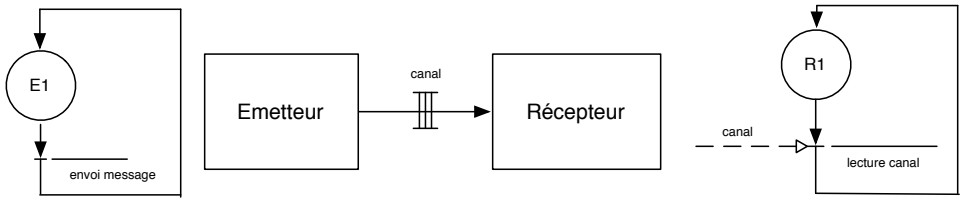


Fig. 13: Modèle MCSE avec un buffer

Listing 3: Un exemple de code FIACRE pour modéliser un buffer

```
1  const MAX_BUFFER : nat is 3 // Taille du buffer
2
3  type T is union
4    MESSAGE
5  end
6  type TBUFF is queue MAX_BUFFER of T
7
8  process EMETTEUR (&canal : read write TBUFF) is
9    states Emission,Test, Envoi_erreur, Erreur
10
11   from E1
12     on not (full(canal));
13     canal := enqueue(canal, MESSAGE); // Envoie du message
14     to E1
15
16 process RECEPTEUR (&canal : read write TBUFF) is
17   states Reception
18
19   from R1
20     on not (empty(canal)); // Attente d'un message
21     canal := dequeue canal; // Lecture du buffer
22     to R1
23
24 component Main is
25   var   tmp_canal : TBUFF := {||}   // Déclaration du buffer et initialisation (vide)
26
27   par  EMETTEUR(&tmp_canal)
28     || RECEPTEUR(&tmp_canal)
29   end
```

4.2.3 Synchronisation par événement

La sémantique des synchronisation par événement en MCSE étant trop faible pour en faire une interprétation correcte en FIACRE, il est nécessaire de commencer par mieux la définir. MCSE indique simplement qu'une fonction est sensibilisée quand un événement est produit.

Nous supposerons que l'occurrence d'un événement est mémorisée (sans compteur) pour chaque consommateur de l'événement. La production d'un événement n'est pas bloquant. À chaque fois que l'événement est utilisé dans une garde pour franchir une transition, nous supposons que l'événement est consommé.

Pour représenter cela, nous ne pouvons pas utilisé un port en FIACRE qui est une synchronisation de type rendez-vous. En place, nous utilisons des variables de type « flag », c'est-à-dire un booléen signalant la présence ou non de l'événement. Pour chaque processus qui se synchronise sur un événement nous créons une variable. La production de l'événement consiste à passer à vrai tous les flags liés à cet événement. La synchronisation se fait simplement en utilisant le mot-clef **on** de FIACRE, puis par une remise à faux du flag utilisé. Remarquons que la synchronisation et la remise a faux sont atomique en FIACRE, ce qui est cohérent avec le comportement MCSE.

L'exemple suivant illustre ce principe : deux processus concurrents sont synchronisés sur la même horloge et attendent la production du top d'horloge pour pouvoir s'exécuter. La figure 14 et le listing 4 présentent respectivement le modèle MCSE et le code FIACRE de cet exemple. La production de l'évènement est faite lignes 30 et 31. La synchronisation des processus sur cet événement est décrite lignes 4 et 15.

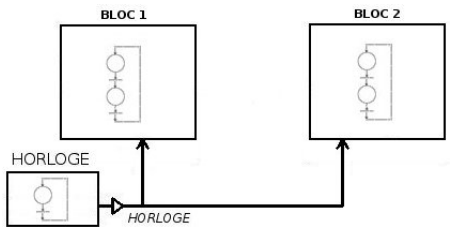


Fig. 14: Modèle MCSE de synchronisation par événement

4.3 Gestion du temps

Afin de modéliser le temps consommé par une action, le mot-clé **wait** est utilisé suivi de l'intervalle de temps représentant le temps consommé par le traitement. Le listing 4 fait

Listing 4: Un exemple de code FIACRE de synchronisation par événement

```
process BLOC1 (&activation_bloc1 : read write bool) is
  states B1.1, B1.2
  from B1.1
    on activation_bloc1;      // Syncrhonisation sur activiation_bloc1
    activation_bloc1 := false;      // Réinitialisation du flag
    to B1.2

  from B1.2
    wait[1,3];
    to B1.1

process BLOC2 (&activation_bloc2 : read write bool) is
  states Etat1, Etat2
  from Etat1
    on activation_bloc2;      // Syncrhonisation sur activiation_bloc2
    activation_bloc2 := false;      // Réinitialisation du flag
    to Etat2

  from Etat2
    wait[5,8];
    to Etat1

process HORLOGE (
  &activation_bloc1 : read write bool,
  &activation_bloc2 : read write bool) is
  states Attente

  from Attente
    wait[50,50];                // Mise à jour des flags d'activation des fonctions
    activation_bloc1 := true;   // toutes les 50 unités de temps
    activation_bloc2 := true;
    to Attente

component Main is
  var   tmp_activation1 : bool := false,    // Initialisation des flags
        tmp_activation2 : bool := false

  par   BLOC1(&tmp_activation1)
    || BLOC2(&tmp_activation2)
    || HORLOGE(&tmp_activation1, &tmp_activation2)

  end
```

apparaître une telle temporisation ligne 9 dans l'état B1.2 du processus B1.

Pour modéliser les horloges plusieurs approches ont été envisagées :
- créer un processus pour chaque horloge avec une boucle et un délai (fonction *wait*) égal à la période. L'inconvénient majeur de ce mécanisme est qu'il engendre une explosion de l'espace d'état, d'autant plus grande que le nombre d'horloges est important. Cette approche n'est pas viable.
- avoir un seul processus périodique avec un pas égal au plus petit commun diviseur de toutes les horloges et qui reproduit le cycle d'apparition des différents événements. Cette solution permet de réduire l'espace d'état tout en conservant le comportement souhaité. Nous privilégions cette approche.

En ce qui concerne les fonctions synchronisées, l'attente du passage d'un flag à « vrai » se fait à l'aide du mot-clef **on**. Ce test bloquant permet d'éviter une scrutation continuelle de l'état du flag qu'entraîne l'utilisation d'une structure *if ... then ... else* et permet une diminution de la taille de l'espace d'état.

Modéliser l'ordonnancement est un problème que FIACRE ne permet pas de résoudre entièrement. Le seul ordonnancement pouvant être modélisé est un ordonnancement non préemptif à priorités fixes monoprocesseur. Pour cela, le mécanisme de flags est repris pour modéliser la ressource processeur. Une variable booléenne partagée, CPU, est créée et chaque processus vérifie la disponibilité de CPU avant de commencer son exécution. Si la ressource est libre, le processus la consomme en plaçant le flag à faux, si elle n'est pas libre, il attend. A la fin de son exécution, le processus rend la main en plaçant le flag à vrai. Un exemple de code est fourni par le listing 6. Il est cependant nécessaire de noter quelques points importants :
- la désactivation du flag CPU doit impérativement se faire juste après avoir pris la ressource CPU de façon à ce que le flag ne puisse pas être remis à vrai entre-temps et ainsi donner la possibilité à une autre tache de venir s'éxécuter en parallèle (le listing 5 offre un exemple de mauvaise utilisation). Pour y remédier, il est nécessaire :
  - soit que les transitions sensibilisées soient franchies immédiatement en ajoutant une garde $[0, 0]$,
  - soit de placer le flag à faux dès que la décision d'exécution d'une tâche a été validé, avant la transition(voir l'exemple du listing 6).
- si un blocage intervient lors de l'exécution d'un état la ressource CPU n'est jamais repassé à vrai, ce qui entraîne un blocage complet du système,
- il est possible d'ajouter des priorités sur les transitions pour modéliser un ordonnancement non-préemptif à priorités fixes.

Listing 5: Un exemple de problème de gestion de l'ordonnancement

```
process BLOC1 (&activation_bloc1 : read write bool, &CPU : read write bool) is
  states Etat1, Etat2, Execution

  from Etat1
    on activation_bloc1;
    to Etat2

  from Etat2
    on CPU;
    activation_bloc1 := false;
    to Execution  // Lors du passage entre les deux états, un autre
            // processus peut prendre la main et consommer la
            // ressource CPU

  from Execution
    CPU := false;  // La mise à jour du flag au début de l'exécution
    wait[1,6];     // n'assure pas le non-parallélisme
    CPU := true;
    to Etat1
```

Listing 6: Un exemple de modélisation d'un ordonnancement non-préemptif

```
process BLOC1 (&activation_bloc1 : read write bool, &cpu : read write bool) is
  states Etat1, Etat2, Execution

  from Etat1
    on activation_bloc1;
    to Etat2

  from Etat2
    on cpu;
    cpu := false;  // Prise du cpu
    activation_bloc1 := false;
    to Execution

  from Execution
    wait[1,6];
    cpu := true;  // Le cpu est relâché
    to Etat1

process BLOC2 (&activation_bloc2 : read write bool, &cpu : read write bool) is
  states Etat1, Etat2, Execution

  from Etat1
    on activation_bloc1;
    to Etat2

  from Etat2
    on cpu;
    cpu := false;  // Prise du cpu
    activation_bloc2 := false;
    to Execution

  from Execution
    wait[1,6];
    cpu := true;  // Le cpu est relâché
    to Etat1

process HORLOGE (&activation_bloc1 : write bool, &activation_bloc2 : write bool) is
  states Attente

  from Attente
    wait[50, 50];  // Les processus sont activés périodiquement
    activation_bloc1 := true;
    activation_bloc2 := true;
    to Attente

component Main is
  var activation1 : bool := false,
      activation2 : bool := false,
      flag_cpu : bool := true  // Le cpu est initialement libre

  par   BLOC1(&activation1, &flag_cpu)
      || BLOC1(&activation2, &flag_cpu)
      || HORLOGE(activation1, activation2)
  end
```

5 Conclusion

Cette étude montre la faisabilité de la réécriture d'un modèle MCSE en FIACRE. L'ensemble des éléments du modèle a été pris en considération et nous avons montré comment effectuer la transformation. Nous avons aussi mis en avant le manque de sémantique de certaines parties de MCSE en particulier en ce qui concerne les échanges de données et la synchronisation. Ce manque a été comblé en étendant le modèle pour nos besoins.

La prochain étape du travail consiste à outiller cette transformation afin de la rendre automatique. Pour cela il semble judicieux d'utiliser les méthodes issues de l'ingénirie dirigée par les modèles et donc de passer par une première phase de méta-modélisation des langages avant d'en définir formellement les règles de transformation.

Un travail complémentaire à celui présenté dans ce document serait de proposer une méthodes pour générer automatiquement une application exécutable à partir du modèle MCSE qui respecte la même sémantique.

Un autre problème n'a pas été explicitement abordé dans ce document à savoir la modélisation de l'environnement et comment représenter les scénarios de test. En effet, tout système réagit à son environnement, il est donc nécessaire d'en modéliser son comportement. De même, les propriétés vérifiables ne sont pas clairement explicitées et mériteraient une attention particulière.

Références

[1] Documentation fiacre : http ://projects.laas.fr/fiacre/papers.php.

[2] Documentation tina : http ://projects.laas.fr/tina/papers.php.

[3] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, M. Filali, M. Pantel, and F. Vernadat. Langage intermédiaire et transformations de modèles pour le développement de systèmes temps-réel : retour d'expérience sur la chaÃőne de vérification formaelle fiacre. Technical report, CNRS/LAAS/IRIT/Université de Toulouse, 2010.

[4] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre : an intermediate language for model verification in the topcased environment. In *Embedded Real Time Software and Systems*, Toulouse, 2008. ERTS.

[5] J.-P. Calvez. *Spécification et conception des systèmes : une méthodologie.* Editions Masson, 1990.