VÉRIFICATION DE SYSTÈMES GALS EN COMBINANT LANGAGES SYNCHRONES ET ALGÈBRES DE PROCESSUS

Damien THIVOLLE[1]

Un sistem GALS (Globally Asynchronous Locally Synchronous) este constituit dintr-o colecţie de componente secvenţiale şi deterministe care se execută în mod concurent şi care comunică utilizând canale lente sau defectuoase. Acest articol propune o metodologie generală pentru modelizarea şi verificarea sistemelor GALS utilizând o combinaţie de limbaje sincrone (pentru componentele secvenţiale) şi calcule de procese (pentru canalele de comunicaţie şi concurenţă asincronă). Această metodologie este ilustrată cu ajutorul unui studiu de caz industrial furnizat de Airbus: un protocol de comunicaţie TFTP/UDP între un avion şi baza terestră, modelizat cu atelierul Eclipse/TOPCASED pentru ingineria dirijată de modele, şi apoi analizat formal cu package-ul CADP pentru verificare şi evaluarea performanţelor.

A Gals (Globally Asynchronous Locally Synchronous) system typically consists of a collection of sequential, deterministic components that execute concurrently and communicate using slow or unreliable channels. This paper proposes a general approach for modelling and verifying Gals systems using a combination of synchronous languages (for the sequential components) and process calculi (for communication channels and asynchronous concurrency). This approach is illustrated with an industrial case-study provided by Airbus: a Tftp/Udp communication protocol between a plane and the ground, which is modelled using the Eclipse/Topcased workbench for model-driven engineering and then analysed formally using the Cadp verification and performance evaluation toolbox.

Un système GALS (Globalement Asynchrone Localement Synchrone) est normalement constitué d'une collection de composants séquentiels et déterministes qui s'exécutent de façon concurrente et communiquent au moyen de canaux lents et non sûrs. Cet article propose une approche générale pour la modélisation et la vérification des systèmes GALS en utilisant une combinaison de langages synchrones (pour les composants séquentiels) et d'algèbres de processus (pour les canaux de communication et le parallélisme asynchrone). Cette approche est illustrée par une étude de cas industrielle fournie par Airbus : une variante du protocole TFTP pour les communications entre un avion et le sol, qui est modélisée à l'aide de la plateforme Eclipse/TOPCASED puis analysée formellement au moyen de la boîte à outils de vérification et d'évaluation de performances CADP.

Mots-clés: vérification formelle, systèmes GALS, CADP, SAM, LOTOS NT

[1]PhD student, INRIA Grenoble-Rhône-Alpes, 655 avenue de l'Europe, 38 330 Montbonnot Saint Martin, France, University POLITEHNICA of Bucharest, Romania, e-mail: damien.thivolle@inria.fr

1. Introduction

Dans le domaine de l'électronique, la conception de circuits synchrones (i.e., circuits dont la logique est gouvernée par une horloge centrale) est depuis longtemps l'approche de choix. Dans le domaine du logiciel, les langages synchrones sont définis sur des concepts similaires. Quelque soit leur syntaxe concrète (flux de données ou formalisme d'automates), ces langages partagent un paradigme commun : un programme synchrone est formé de composants qui évoluent par des étapes discrètes et une horloge centrale garantit que tous les composants évoluent simultanément. Chaque composant est normalement déterministe, tout comme la composition de tous les composants. Cette hypothèse simplifie grandement la simulation, le test et la vérification des systèmes synchrones.

Ces vingt dernières années, les langages synchrones sont devenus la norme pour la programmation de systèmes critiques embarqués comme les contrôleurs que l'on peut trouver dans les avions, les voitures, les trains ou encore les centrales nucléaires. Ces langages ont aussi trouvé une place dans la conception des circuits électroniques. Comme exemples de langages synchrones, nous pouvons citer ESTEREL [1], LUSTRE/SCADE [2], SIGNAL/SILDEX [3] et ARGOS [4] . Nous invitons le lecteur à se référer à [5] pour une liste de succès récents dans l'application des techniques de vérification formelle à des systèmes avioniques complexes.

De plus en plus, les systèmes embarqués ne satisfont plus les propriétés des systèmes synchrones. Les approches récentes (modular avionics, X-by-wire...) introduisent un degré croissant d'asynchronisme et de non déterminisme. Cette situation est connue depuis longtemps dans l'industrie des circuits électroniques où le terme GALS (Globalement Asynchrone Localement Synchrone) est employé pour désigner les circuits qui consistent en un ensemble de composants synchrones (gouvernés par leur propre horloge) qui communiquent de façon asynchrone. Ces évolutions remettent en cause la position bien établie des langages synchrones dans l'industrie. En effet, l'introduction d'asynchronisme invalide les propriétés de non déterminisme et d'instantanéité des systèmes réactifs et rend donc caduques les techniques de vérification efficaces qui existent pour ces systèmes. Il devient alors nécessaire d'adapter les techniques de vérification existantes au cas des systèmes GALS.

Nous avons trouvé dans la littérature diverses tentatives visant à repousser les limites des langages synchrones pour les appliquer à l'étude des systèmes GALS. Suivant les résultats de Milner [6] qui ont montré que l'asynchronisme peut être encodé dans le modèle de calcul synchrone, nombre d'auteurs [7] [8] [9] [10] se sont efforcés de décrire les systèmes GALS à l'aide de langages synchrones ; par exemple, le non déterminisme est exprimé par l'ajout d'entrées

auxiliaires (oracles) dont la valeur est indéfinie. Le désavantage principal de ces approches est que l'asynchronisme et le non déterminisme ne sont pas reconnus comme des concepts de première classe donc les outils de vérification des langages synchrones n'ont pas d'optimisations spécifiques au parallélisme asynchrone (ordres partiels, minimisation compositionnelle...). D'autres approches étendent les langages synchrones pour permettre un certain degré d'asynchronisme, comme dans CRP [11], CRSM [12] ou encore multiclock ESTEREL [13], mais, à notre connaissance, de telles extensions ne sont pas (encore) utilisées dans l'industrie. Enfin, nous pouvons mentionner les approches [14] [15] dans lesquelles les langages synchrones sont compilés et distribués automatiquement sur un ensemble de processeurs s'exécutant en parallèle. Bien que ces approches permettent de générer directement des implémentations de systèmes GALS, elles ne traitent pas de la modélisation et de la vérification de ces systèmes.

Une approche totalement différente serait d'ignorer les langages synchrones et d'adopter des langages spécifiquement conçus pour modéliser le parallélisme asynchrone et le non déterminisme, et équipés de puissants outils de vérification formelle comme les algèbres de processus : CSP [16] , LOTOS [17] ou PROMELA [18]. Un tel changement de paradigme est aujourd'hui impensable pour des entreprises qui ont investi massivement dans les langages synchrones et dont les produits à cycles de vie extrêmement longs demandent une certaine stabilité en termes de langages de programmation et d'environnements de développement.

Dans cet article, nous proposons une approche intermédiaire qui combine les langages synchrones et les algèbres de processus pour modéliser, vérifier et simuler les systèmes GALS. Notre approche essaie de retenir le meilleur des deux paradigmes :

- Nous continuons à utiliser les langages synchrones et leurs outils pour spécifier et vérifier les composants synchrones d'un système GALS.
- Nous introduisons les algèbres de processus pour : (1) encapsuler ces composants synchrones ; (2) modéliser des composants additionnels dont le comportement est non déterministe, comme par exemple des canaux de communications non sûrs qui peuvent perdre, dupliquer et/ou permuter des messages ; (3) interconnecter tous ces composants d'un même système GALS grâce au parallélisme asynchrone. La spécification qui résulte est asynchrone et peut être analysée par les outils accompagnant l'algèbre de processus considérée.

Nous avons trouvé dans la littérature deux approches qui suivent une direction similaire. Dans [19], des spécifications CRSM [12] sont automatiquement traduites en PROMELA pour vérifier grâce au model checker SPIN des propriétés exprimées comme un ensemble d'observateurs. Notre

approche est différente car nous réutilisons les langages synchrones tels qu'ils sont, sans qu'il soit nécessaire d'introduire un nouveau langage synchrone/asynchrone comme CRSM.

Dans [20], plus similaire à notre approche, le compilateur SIGNAL est utilisé pour générer du code C à partir de programmes synchrones écrits en SIGNAL. Ce code est ensuite encapsulé dans des processus PROMELA qui communiquent par une abstraction d'un bus matériel. Enfin, le model checker SPIN est utilisé pour vérifier des formules de logique temporelle sur la spécification obtenue. L'approche que nous proposons suit le même principe mais présente des différences clés avec [20] dans la façon d'intégrer les programmes synchrones dans un environnement asynchrone:

- Le protocole de communication qui relie les deux programmes synchrones présentés dans [20] est implémenté dans [21] en LUSTRE et SIGNAL. Ce protocole présente un degré faible d'asynchronisme et aucun non déterminisme. Il est d'ailleurs prouvé que ce protocole équivaut à un canal FIFO sans perte à un élément. Notre approche est plus générale car la communication entre le programme synchrone et son environnement peut se faire soit directement à l'aide d'un canal de communication, soit par l'intermédiaire d'un processus asynchrone auxiliaire qui implémente un protocole donné.
- Le degré d'asynchronisme est encore limité par l'utilisation de la directive "atomic" de PROMELA qui assure le non entrelacement de la séquence d'actions qu'elle englobe avec les actions de l'environnement. Dans leur approche, cette directive englobe la totalité des actions de chacun des deux processus asynchrones qui encapsulent les deux programmes synchrones. De cette façon, la réception des entrées dans l'un des processus asynchrones, l'appel de la fonction C encodant le programme synchrone et l'envoi des sorties à l'environnement sont une séquence atomique d'actions. Les deux programmes synchrones ne peuvent donc pas s'exécuter de façon concurrente ce qui, pour nous, ne constitue pas un vrai exemple de système GALS. Au contraire, notre approche est complètement asynchrone et les exécutions des processus asynchrones qui encapsulent les programmes synchrones peuvent s'entrelacer. Notre approche est donc plus générale car : (1) elle permet une modélisation plus réaliste de la sémantique des systèmes GALS (elle ne requiert pas l'arrêt du système tout entier durant le calcul de la réaction de l'un des programmes synchrones) et (2) elle est applicable à un large panel d'algèbres de processus dont la plupart (contrairement à PROMELA) ne possède pas de directive "atomic" ; les seules contraintes pour ces algèbres de processus sont de permettre à l'utilisateur de définir des types et des

fonctions ainsi que d'avoir les primitives classiques pour exprimer le parallélisme asynchrone.

- Dans leur approche, les processus asynchrones qui encapsulent les programmes synchrones sont vides et ne font que transmettre les valeurs reçues par l'environnement au programme synchrone. Dans la réalité, ce schéma de systèmes GALS est trop restrictif car il arrive que des programmes synchrones ne spécifient que la partie "contrôle" d'une application et que ce soit les programmes asynchrones qui définissent le flux de données. Dans notre approche, nous prenons cela en compte et le degré de complexité du processus asynchrone encapsulateur peut varier selon le système GALS à modéliser.

Nous illustrons notre approche par une étude de cas industrielle fournie par Airbus dans le contexte du projet TOPCASED[2] : un protocole de communication entre un avion et le sol qui consiste en deux entités TFTP (*Trivial File Transfer Protocol*) qui s'exécutent en parallèle et qui communiquent par un canal UDP (*User Datagram Protocol*). Comme langage synchrone, nous considérons SAM [22], (similaire à ARGOS [4]) qui a été conçu par Airbus et qui est utilisé au sein de cette entreprise. Une suite logicielle pour SAM est distribuée avec la plateforme *open-source* TOPCASED basée sur Eclipse. Comme algèbre de processus, nous considérons LOTOS NT [23], une version simplifiée de la norme internationale E-LOTOS [24]. Un traducteur automatique qui transforme des spécifications LOTOS NT en spécifications LOTOS est développé au sein de la boîte à outils CADP [25]. Ces spécifications LOTOS générées peuvent alors être vérifiées et leurs performances évaluées.

Ce document est organisé comme suit. La section **2** explique les principes de notre approche et illustre son application aux langages SAM et LOTOS NT. La section **3** détaille l'étude de cas industrielle. La section **4** traite de la modélisation formelle de l'étude de cas en LOTOS NT. La section **5** détaille les techniques que nous avons employées pour générer les espaces d'états de nos spécifications et expose nos résultats de vérification. Enfin, la section **6** donne des remarques conclusives et nos perspectives concernant ces travaux.

2. Approche proposée

Dans cette section, nous détaillons notre approche pour la modélisation des systèmes GALS à l'aide des langages synchrones et des algèbres de processus. Nous présentons ensuite l'application de cette méthode aux langages SAM et LOTOS NT.

[2] http://www.topcased.org

2.1. Les programmes synchrones vus comme des fonctions de Mealy. Un *programme synchrone* est la composition synchrone d'un ou plusieurs *composants synchrones*. Un composant synchrone effectue une séquence d'itérations discrètes et maintient un état interne s. A chaque itération, il reçoit un ensemble de m valeurs d'entrée $i_1...i_m$ de son environnement, calcule instantanément une réaction, renvoie un ensemble de n valeurs $o_1...o_n$ à son environnement et se positionne sur son nouvel état s_0. Autrement dit, il peut être représenté par une *machine de Mealy* [26] qui est un quintuplet (S, s_0, I, O, f) où :

- S est un ensemble fini d'états,
- s_0 est l'état initial,
- I est un alphabet fini d'entrée,
- O est un alphabet fini de sortie,
- f □ S × I → S × O est une fonction de transition (aussi appelée une fonction de Mealy) qui associe à l'état courant et un symbole de l'alphabet d'entrée, l'état pour la réaction suivante ainsi qu'un symbole de l'alphabet de sortie : $f(s, i_1...i_m) = (s_0, o_1...o_n)$.

Lorsqu'un programme synchrone a plusieurs composants, ces composants peuvent communiquer les uns avec les autres grâce à des connexions entre les sorties de certains composants et les entrées d'autres composants. Par définition du parallélisme synchrone, à chaque itération, tous les composants réagissent simultanément. Par conséquent, la composition de plusieurs composants peut aussi être représentée par une machine de Mealy. Pour les langages synchrones ESTEREL et LUSTRE, un format intermédiare commun OC (Object Code) a été proposé pour représenter ces machines de Mealy.

2.2. Le langage SAM. Pour illustrer notre approche, nous considérons le cas du langage SAM, défini par Airbus et dont une description formelle est donnée dans [22]. Un composant synchrone en SAM est un automate qui a un ensemble de ports d'entrée et un ensemble de ports de sortie. Chaque port correspond à une variable booléenne. Un composant SAM est très proche d'une machine de Mealy. La seule différence réside dans le fait que les transitions sortant d'un même état ont des indices de priorité qui leur sont associés. Cela est nécessaire car en SAM, plusieurs transitions peuvent être activées par une même série de valeurs d'entrée.

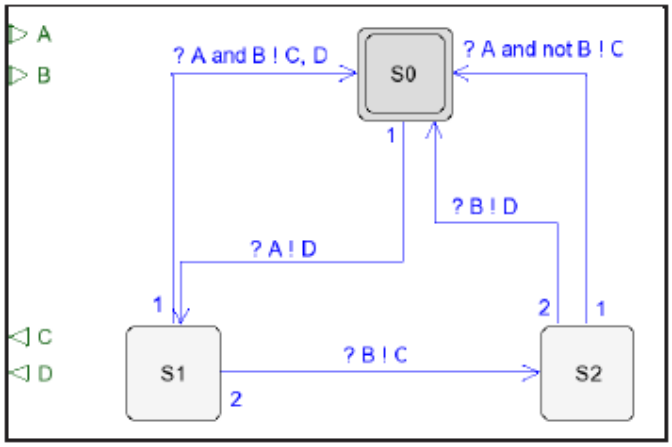

Fig. 1. Exemple d'automate SAM

La Figure 1 illustre l'exemple d'un composant SAM. Un point d'interrogation précède la condition F de chaque transition tandis qu'un point d'exclamation précède la liste G de variables de sortie auxquelles la valeur *vrai* doit être affectée. Les indices de priorité sont situés à la source des transitions.

La composition de composants SAM suit la sémantique classique de la composition des programmes synchrones. Les communications entre les différents programmes sont exprimées par la connexion graphique de ports de sortie et de ports d'entrée, en respectant les règles suivantes:

- Les ports d'entrée d'une composition peuvent être connectés aux ports de sortie de la composition ou bien aux ports d'entrée des sous-programmes (i.e., les programmes qui participent à la composition).
- Les ports de sortie d'un sous-programme peuvent être connectés aux ports d'entrée d'autres sous-programmes ou bien aux ports de sortie de la composition.
- Les dépendances cycliques sont interdites : il est interdit de connecter le port de sortie d'un sous-programme au port d'entrée du même sous-programme, que ce soit directement ou par transitivité, au moyen d'un ou plusieurs sous-programmes intermédiaires.

2.3. Traduction de SAM en LOTOS NT. Dans cette section, nous illustrons la façon dont un automate SAM peut être représenté par sa fonction de Mealy encodée en LOTOS NT. Par exemple, l'automate de la figure 1 peut être encodé comme suit en LOTOS NT :

```
type State is
      S0, S1, S2 -- c'est un type enumere
end type
```

```
function Transition (in CurrentState:State, in A:Bool, in B:Bool
                     out NextState:State, out C:Bool, out D:Bool)
is
      NextState := CurrentState; C := false ; D := false ;
      case CurrentState in
      S0 ->
            if A then
                  NextState := S1; D := true
            end if
      |S1 ->
       if A and B then
            NextState := S0; C := true; D := true
       elsif B then
            NextState := S2; C := true
        endif
      |S2 ->
        if A and not (B) then
            NextState := S0; C := true
        elsif B then
            NextState := S0; D := true
        end if
      end case
end function
```

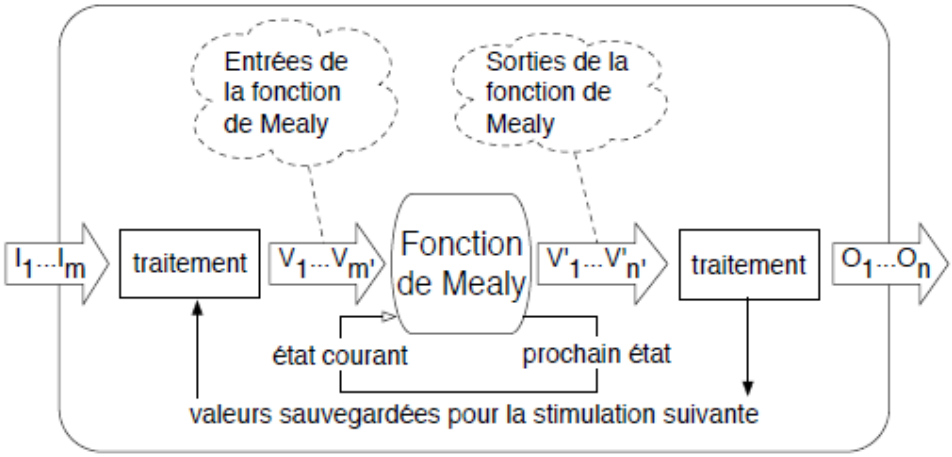

Fig. 2. Processus asynchrone encapsulateur (wrapper) général

Un système SAM comprenant plusieurs automates SAM se traduit aisément en LOTOS NT. Comme les dépendances cycliques sont interdites, il est possible d'effectuer un tri topologique des composants en fonction de leurs dépendances les uns aux autres. A partir de l'ordre obtenu par ce tri, le système SAM peut être encodé en LOTOS NT comme la composition séquentielle des fonctions de Mealy de ses composants, c'est-à-dire en appelant les fonctions de Mealy des composants dans l'ordre induit par le tri, de telle sorte que lors de

l'appel de la fonction de Mealy d'un composant donné, les valeurs de toutes ses variables d'entrée sont connues.

Une approche alternative à la traduction d'un langage synchrone L vers LOTOS NT serait, si il existe un générateur de code de L vers le langage C, d'invoquer directement la fonction de Mealy (générée en C) depuis un programme LOTOS NT, comme une fonction externe (une fonctionnalité supportée par CADP). De cette façon, notre approche pourrait même permettre le mélange de composants écrits dans différents langages synchrones.

2.4. Encapsulation de fonctions de Mealy dans des processus LOTOS NT. A la différence des programmes synchrones, les composants de programmes asynchrones évoluent en parallèle, à leur propre rythme et se synchronisent ponctuellement à l'aide de canaux de communication. Notre approche pour la modélisation des systèmes GALS dans des langages asynchrones consiste à encoder un programme synchrone comme un ensemble de types et fonctions natifs dans l'algèbre de processus considérée. Mais, la fonction de Mealy d'un programme synchrone, seule, ne peut interagir avec un environnement asynchrone. Elle doit être encapsulée dans un *wrapper*, c'est-à-dire un processus asynchrone qui fait l'interface entre l'environnement asynchrone et la fonction de Mealy. Ce *wrapper* transforme la fonction deMealy en STE (*Système de Transitions Etiquetées*). Dans notre cas, la fonction deMealy est une fonction LOTOS NT tandis que le *wrapper* est un processus LOTOS NT.

La quantité de traitement qu'un *wrapper* peut faire dépend du système GALS à modéliser. La Figure 2 montre le fonctionnement de général d'un *wrapper* : réception des entrées, envoi des sorties et sauvegarde de certaines valeurs pour les réutiliser à l'itération suivante. Dans certains cas, le *wrapper* peut aussi implémenter des comportements additionnels, non spécifiés par la fonction de Mealy.

Une fois que la fonction de Mealy est encapsulée dans un *wrapper*, elle peut se synchroniser et communiquer avec les autres processus asynchrones grâce à l'opérateur de composition parallèle de LOTOS NT.

3. Description de l'étude de cas

Cette étude de cas a été distribuée par Airbus aux participants du projet TOPCASED pour illustrer un système embarqué avionique typique. Dans cette section, nous commençons par présenter les principes du protocole TFTP avant de décrire les changements effectués sur ce protocole par Airbus pour permettre la communication entre un avion et le sol.

3.1. Protocole TFTP. TFTP [27] est l'acronyme de *Trivial File Transfer Protocol*. Il s'agit d'un protocole client/serveur grâce auquel plusieurs clients peuvent écrire (*resp.* lire) un fichier sur (*resp.* depuis) un serveur. Pour des raisons de vitesse de transmission, TFTP utilise UDP (*User Datagram Protocol*) de transport et doit donc implémenter un mécanisme de contrôle du flux des messages afin de pallier les éventuelles erreurs se produisant dans la couche de transport UDP. Pour permettre au serveur de différencier les clients qu'il sert, chaque transfert de fichier s'effectue sur un port UDP différent.

Le protocole TFTP définit 5 types de message :

- `RRQ` (*Read ReQuest*) pour demander à lire un fichier depuis le serveur,
- `WRQ` (*Write ReQuest*) pour demander à écrire un fichier sur le serveur,
- `DATA` qui contient un fragment de fichier numéroté de 512 octets ; Le dernier fragment est celui dont la taille est différente de 512,
- `ACK` qui contient le numéro du fragment acquitté,
- `ERROR` pour indiquer qu'une erreur s'est produite.

Le protocole est robuste : un message perdu (`RRQ`, `WRQ`, `DATA` ou `ACK`) peut être retransmis après un *timeout*. Les acquittements dupliqués (renvoyés à cause d'un *timeout* par exemple) doivent être ignorés afin d'éviter le *bogue de l'apprenti sorcier* [28].

En cas d'erreur (épuisement de la mémoire disponible, erreur du système...), un message `ERROR` est envoyé pour annuler le transfert.

3.2. Variante AIRBUS du protocole TFTP. Lorsqu'un avion atteint sa position finale dans l'aéroport, il est connecté au réseau informatique de cet aéroport. A l'heure actuelle, les communications qui se déroulent entre l'avion et les serveurs de l'aéroport sont régies par un protocole de communication très simple et certifié correct. Airbus nous a demandé d'étudier un protocole plus complexe, une variante du protocole TFTP qui pourrait être d'intérêt pour de nouvelles générations d'avions. Les principales différences entre ce protocole et le protocole TFTP classique sont :

- Dans la pile de protocoles considérée par Airbus, la variante du protocole TFTP repose toujours sur le protocole UDP pour la transmission des messages. Cependant, ce ne sont plus des fichiers qui sont transportés mais les trames d'un protocole de communication de plus haut niveau dédié à l'avionique (comme ARINC 615a).

- Chaque entité communicante de cette variante du protocole TFTP a la faculté d'être à la fois client ou serveur, selon ce que requiert le protocole de communication de plus haut niveau.
- Chaque entité ne communique qu'avec une seule autre entité. En effet, pour chaque avion qui se connecte, il y a dans les serveurs de l'aéroport une entité TFTP qui lui est réservée. Cela nous permet de ne pas modéliser le fait qu'une entité peut transférer plusieurs fichiers simultanément sur des ports UDP différents. Dans le reste de ce document, l'abréviation TFTP désigne (sauf mention contraire) la variante du protocole TFTP définie par Airbus.

Les entités TFTP ont été spécifiées par Airbus au moyen d'un automate SAM de 7 états, 39 transitions, 15 ports d'entrée et 11 ports de sortie.

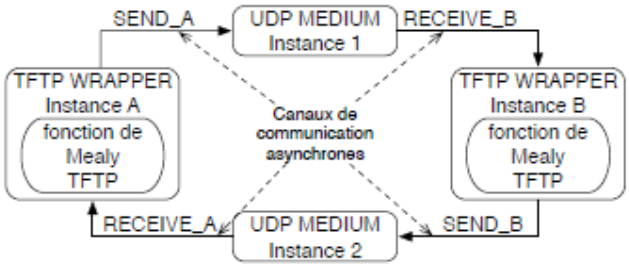

Fig. 3. Connexion asynchrone de deux processus TFTP via deux média UDP

Dans la suite de ce document, nous désignons cet automate par l'appellation "automate TFTP SAM". Airbus était intéressé par l'étude du comportement de deux entités TFTP (dont le comportement est régi par l'automate TFTP SAM) communiquant par un médium non sûr comme UDP (c'est-à-dire avec des pertes, des duplications et des permutations de messages).

4. Modélisation en LOTOS NT

Nous avons modélisé une spécification qui comporte deux entités TFTP connectées par deux média UDP. Comme illustré en figure 3, les entités TFTP sont deux instances du même processus LOTOS NT qui lui-même encapsule la fonction de Mealy de l'automate TFTP SAM. Cet automate a été traduit manuellement en 215 lignes de code LOTOS NT (ce nombre inclut la fonction de Mealy et le type énuméré qui encode les états). Les média sont deux instances du même processus LOTOS NT qui reproduit les propriétés (perte, duplication et permutation de messages) du protocole de transport UDP.

4.2. Modélisation d'entités TFTP. Le processus TFTP reçoit et envoie des messages TFTP tels que définis dans la norme. Sa définition en LOTOS NT est longue de 670 lignes de code.

Afin de modéliser fidèlement les messages du protocole TFTP, nous devons modéliser les fichiers et leurs fragments. Pour ce faire, nous considérons qu'à chaque entité TFTP est associé un répertoire de fichiers et que chaque entité TFTP est instanciée avec les paramètres suivants :

- une liste de fichiers, parmi ceux du répertoire, à écrire sur l'autre entité ; nous désignons cette liste de fichiers par "liste de fichiers à écrire",
- une liste de fichiers, parmi ceux du répertoire de l'autre entité, à lire depuis l'autre entité ; nous désignons cette liste de fichiers par "liste de fichiers à lire".

Lorsqu'il n'y a pas de transfert en cours, l'une des entités peut choisir, de façon non déterministe, un fichier parmi sa liste de fichiers à lire ou à écrire et commencer le transfert de ce fichier.

Le type de données que nous utilisons pour représenter les fichiers est une liste de fragments (dans notre modèle, le fichier est donc déjà fragmenté). Les noms des fichiers sont représentés par un entier naturel unique associé au fichier dans le répertoire qui le contient. Chaque fragment de fichier est représenté par un caractère différent.

En plus de l'état courant de l'automate TFTP SAM, d'autres valeurs doivent être sauvegardées entre deux stimulations comme par exemple le nom du fichier en cours de transfert, l'indice du dernier fragment (ou acquittement) reçu ou envoyé, le nombre de renvois du dernier message...

Les listes de fichiers et le contenu de chaque fichier sont des paramètres modifiables auxquels s'ajoute la possibilité de spécifier le nombre maximal de renvois des messages. Ces paramètres nous permettent d'explorer différents scénarios dans la section 5. En jouant sur les valeurs de ces paramètres, nous pouvons aussi contrôler, dans une certaine mesure, la taille de l'espace d'états de notre spécification.

4.3. Modélisation des liens de communication. Les deux processus LOTOS NT décrivant les média UDP n'ont pas été dérivés d'une spécification SAM mais écrits directement, par nos soins, en LOTOS NT.

Ces processus reproduisent de façon précise la couche de transport UDP mise en œuvre dans le réseau informatique reliant le sol et l'avion. UDP est un protocole dit *non connecté*, c'est-à-dire que chaque message est envoyé sans que les mécanismes du protocole ne permettent de déterminer qu'il a bien été reçu. Ce protocole ne détecte pas, ni ne répare les erreurs survenant dans les

communications. Ces erreurs, lorsqu'elles se produisent, doivent donc être gérées par les applications qui utilisent le protocole UDP pour communiquer (les entités TFTP dans notre cas). Ces erreurs peuvent être des pertes, des permutations ou des duplications de messages.

Nous avons choisi de modéliser le médium UDP de deux façons différentes, au moyen de deux processus LOTOS NT différents afin de nous assurer que les entités TFTP se comportent correctement, indépendamment du médium choisi. Ces deux processus LOTOS NT peuvent perdre les messages et ont une mémoire tampon dans laquelle les messages reçus non perdus sont enregistrés dans l'attente de leur acheminement. Nous ne modélisons pas explicitement les duplications de messages causées par le médium UDP car chaque entité TFTP peut déjà renvoyer un même message un nombre borné de fois (borne qui peut d'ailleurs être différente pour chaque entité TFTP).

Le premier processus modélise le cas où les permutations de messages ne se produisent pas. Il utilise une `FIFO` comme mémoire tampon : les messages sont acheminés dans le même ordre que celui dans lequel ils arrivent. Le second processus modélise le cas ou les permutations de messages se produisent. Il utilise un *bag* comme mémoire tampon. Dans la suite du document, nous notons *`FIFO(`*n) (*resp.* *`BAG(`*n*)*) un médium "`FIFO`" (*resp.* "bag") dont la mémoire tampon a une taille de *n*. *`FIFO(1)`* et *`BAG(1)`* sont identiques.

4.4. Composition parallèle des liens de communication et des entités TFTP. Afin de composer, de façon asynchrone, les entités TFTP et les média UDP comme illustré par la figure 3, nous utilisons l'opérateur parallèle de LOTOS NT :

```
par RECEIVE_A, SEND_A -> TFTP_WRAPPER [RECEIVE_A, SEND_A]
 || RECEIVE_B, SEND_B -> TFTP_WRAPPER [RECEIVE_B, SEND_B]
 || SEND_A, RECEIVE_B -> UDP_MEDIUM [SEND_A, RECEIVE_B
 || SEND_B, RECEIVE_A -> UDP_MEDIUM [SEND_B, RECEIVE_A]
end par
```

Comme nous deux média différents, nous obtenons deux spécifications différentes à vérifier, selon que le médium utilisé est "FIFO" ou "BAG".

5. Vérification fonctionnelle des modèles

Dans cette section, nous discutons des difficultés liées à la génération des espaces d'états des spécifications pour l'étude de cas TFTP et nous présentons les résultats de vérification obtenus à l'aide de CADP.

Les spécifications LOTOS NT sont automatiquement traduites en spécifications LOTOS (par le traducteur "LOTOS NT to LOTOS" [23]) qui sont,

à leur tour, compilées en STE en utilisant les compilateurs CÆSAR.ADT et CÆSAR de CADP.

Un problème récurrent en *model checking* est le phénomène de l'explosion de l'espace d'états. Dans notre cas, ce phénomène peut survenir soit durant la génération de l'espace d'états (quand le STE devient trop large pour être généré dans sa totalité) soit durant la vérification des formules de logique temporelle (quand le *model checker* épuise la mémoire disponible durant l'évaluation d'une formule sur un STE).

Pour lutter contre ce phénomène, nous restreignons la taille de la mémoire tampon des média UDP à de petites valeurs (i.e., n = 1, 2, 3...). Nous limitons aussi la taille de chaque fichier à deux fragments puisque nous avons observé que c'était suffisant pour exercer toutes les transitions de l'automate TFTP SAM. De plus, nous avons remarqué que l'utilisation de plus de deux fragments par fichier n'entraîne pas l'invocation de la fonction de Mealy de l'automate TFTP SAM avec un ensemble de valeurs d'entrée qui n'existait pas déjà lors de l'utilisation de seulement deux fragments. Nous contraignons aussi le nombre de fichiers échangés par les deux entités TFTP en bornant la taille des listes de fichiers à lire et à écrire. Pour couvrir toutes les possibilités d'échanges, nous considérons les cinq scénarios suivants :

- Scénario A: l'entité TFTP A écrit un fichier ;
- Scénario B: l'entité TFTP A lit un fichier ;
- Scénario C: les deux entités TFTP A et B écrivent un fichier ;
- Scénario D: l'entité TFTP A écrit un fichier et l'entité TFTP B écrit un fichier ;
- Scénario E: les deux entités TFTP A et B lisent un fichier.

Nous avons écrit 29 formules de logiques temporelles que nous avons vérifiées pour chacun des cinq scénarios en faisant varier la taille des média. Pour donner un ordre d'idée, sur une machine équipée de deux processeurs Intel Xeon 2 Ghz et de 7 Go de RAM, il a fallu 182 secondes pour générer la spécification correspondant au scénario D avec un médium *BAG(2)* (16 687 096 états et 83 289 158 transitions) et 17 707 secondes pour vérifier sur cette spécification les 29 propriétés.

La vérification des propriétés nous a permis de découvrir 19 erreurs dans la variante TFTP d'Airbus. Ces erreurs ont été confirmées par Airbus comme étant de réelles erreurs. Nous avons suggéré, pour chacune, un correctif à appliquer sur l'automate SAM TFTP.

6. Conclusion

Dans cet article, nous avons proposé une approche simple et élégante pour la modélisation et l'analyse des systèmes comprenant des composants synchrones

interagissant de façon asynchrone et que l'on appelle couramment GALS (Globalement Asynchrone Localement Synchrone).

Contrairement aux autres approches qui étendent le paradigme synchrone pour modéliser l'asynchronisme, notre approche préserve la sémantique originale des langages synchrones ainsi que la sémantique asynchrone des algèbres de processus. Notre approche nous permet de réutiliser, sans la moindre modification les compilateurs des langages synchrones avec les outils de vérification et les compilateurs des algèbres de processus.

Nous avons démontré la faisabilité de notre approche sur une étude de cas industrielle, une variante du protocole TFTP/UDP dont nous avons vérifié le bon comportement et évalué les performances au moyen de la plateforme TOPCASED et des outils de CADP. Bien que nous ayons illustré notre approche par le langage synchrone SAM et les algèbres de processus LOTOS et LOTOS NT, nous pensons qu'elle est généralisable à d'autres langages synchrones dont le compilateur est capable de traduire des programmes synchrones en machines de Mealy (ce qui est normalement toujours le cas) et à toute algèbre de processus qui permet le parallélisme asynchrone et la définition de types et fonctions.

En ce qui concerne les perspectives de recherche sur ce travail, nous avons reçu un fort soutien d'Airbus. Nous travaillons à l'heure actuelle sur la vérification d'autres systèmes embarqués avioniques. Nous aimerions aussi appliquer notre approche à d'autres langages synchrones que SAM.

RÉFÉRENCES

[1]. *G. Gonthier, G. Berry.* The Esterel Synchronous Programming Language. 19, 1992, Science of Computer Programming, **Vol. 2**, pp. 87–152

[2]. *N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud*, The Synchronous Dataflow Programming Language LUSTRE, September 1991, Proceedings of the IEEE, **Vol. 79**, pp. 1305–1320

[3]. *A. Benveniste, P. Le Guernic, C. Jacquemot,* Synchronous Programming with Events and Relations: The SIGNAL Language and Its Semantics. 2, 1991, Science of Computer Programming, **Vol. 16**, pp. 103–149

[4]. *F. Maraninchi, Y. Rémond,* Argos: an Automaton-Based Synchronous Language**.** 1-3, October 2001, Computer Languages, **Vol. 27**, pp. 61–92

[5]. *S.P. Miller, M.W. Whalen, D.D. Cofer,* Software model checking takes off. 2, 2010, Communications of the ACM, **Vol. 53**, pp. 58–64

[6]. *R. Milner,* Calculi for Synchrony and Asynchrony. 1983, Theoretical Computer Science, **Vol. 26**, pp. 267–310

[7]. *N.H. Baghdadi, S. Baghdadi,* Synchronous Modelling of Asynchronous Systems. London, UK : Springer-Verlag, 2002. EMSOFT '02. pp. 240–251

[8]. *P. Le Guernic, J.-P. Talpin, J.-C. Le Lann,* Polychrony for System Design. World Scientific, 2003, Journal of Circuits, Systems and Computers, **Vol. 12**

[9]. *M.R. Mousavi, P. Le Guernic, J.-P. Talpin, S.K. Shukla, T. Basten,* Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. Washington DC, USA : IEEE Computer Society, 2004. DATE '04

[10]. *N.H. Mandel, L. Mandel,* Simulation and Verification of Asynchronous Systems by Means of a Synchronous Model. Washington DC, USA : IEEE Computer Society, 2006. ACSD '06. pp. 3-14

[11]. *G. Berry, S. Ramesh, R.K. Shyamasundar,* Communicating Reactive Processes. New York, USA : ACM, 1993. POPL'93. pp. 85–98

[12]. *S. Ramesh,* Communicating Reactive State Machines: Design, Model and Implementation. 1998. IFAC Workshop on Distributed Computer Control Systems

[13]. *G.B. Sentovich, E. Sentovich,* Multiclock Esterel. London, UK : Springer-Verlag, 2001. CHARME'01. pp. 110–125

[14]. *A. Girault, C. Ménier,* Automatic Production of Globally Asynchronous Locally Synchronous Systems. London, UK : Springer-Verlag, 2002. EMSOFT '02. pp. 266–281

[15]. *D. Potop-Butucaru, B. Caillaud,* Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. 2007, Fundamenta Informaticae, **Vol. 78**

[16]. *S.D. Brookes, C.A.R. Hoare, A.W. Roscoe,* A Theory of Communicating Sequential Processes. 3, July 1984, Journal of the ACM, **Vol. 31**, pp. 560–599

[17]. *ISO/IEC***.** LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Organization for Standardization — Information Processing Systems —Open Systems Interconnection. Genève : 1989. International Standard 8807

[18]. *G.J. Holzmann,* The Spin Model Checker - Primer and Reference Manual. Addison-Wesley, 2004

[19]. *S. Ramesh, S. Sonalkar, V. D'Silva, N. Chandra, B. Vijayalakshmi,* A Toolset for Modelling and Verification of GALS Systems. Springer-Verlag, 2004. CAV '04 **vol. 3114** of LNCS. pp. 506–509

[20]. *F. Doucet, M. Menarini, I.H. Krüger, R.K. Gupta, J.-P. Talpin,* A Verification Approach for GALS Integration of Synchronous Components. 2, 2006, ENTCS, **Vol. 146**, pp. 105–131

[21]. *A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin,* A Protocol for Loosely Time-Triggered Architectures. London, UK : Springer-Verlag, 2002. EM-SOFT '02. pp. 252-262

[22]. *X. Clerc, H. Garavel, D. Thivolle,* Présentation du langage SAM d'Airbus. VASY, INRIA. 2008. Internal Report

[23]. *D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, W. Serwe, G. Smeding,* Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.0). INRIA/VASY. 2010

[24]. *ISO/IEC*. Enhancements to LOTOS (E-LOTOS). International Organization for Standardization - Information Technology. Genève : 2001. International Standard 15437:2001

[25]. *H. Garavel, F. Lang, R. Mateescu, W. Serwe,* CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes**.**. Berlin, Germany : Springer-Verlag, 2007. CAV'07, **vol. 4590** of LNCS. pp. 158–163

[26]. *H.G. Mealy,* A Method for Synthesizing Sequential Circuits**.**. 5, 1955, Bell System Technical Journal, **Vol. 34**, pp. 1045–1079

[27]. *K. Sollins,* The TFTP Protocol (Revision 2). IETF. 1992. RFC 1350

[28]. *R. Braden,* Requirements for Internet Hosts - Application and Support. IETF. 1989. RFC 1123.