Intégration de Modules Synchrones dans un Langage à Objets

Frédéric Boulanger[①] et [✿], Henri Delebecque[①], Guy Vidal-Naquet[①] et [✿]

[①] École Supérieure d'Électricité - Service Informatique
Plateau de Moulon
91192 Gif sur Yvette Cedex

[✿] Laboratoire de Recherche en Informatique
Université de Paris-Sud - Centre d'Orsay
91400 Orsay

Contact e-mail : Frederic.Boulanger@supelec.fr

1. Introduction

Depuis quelques années, le traitement informatique des systèmes de contrôle-commande a vu se développer une nouvelle approche dite synchrone [BEN&BER - 91].
Grâce à l'hypothèse de synchronisme, qui permet de considérer que la réaction d'un système à un événement est instantanée, cette approche permet de traiter le temps avec rigueur et d'allier parallélisme et déterminisme.
Un des principaux avantages de l'approche synchrone est que la preuve de propriétés peut se faire à partir du code source du programme après sa traduction automatique dans un formalisme mathématique [BEN&LGU - 90], [HAL - 92].
Cependant, le code produit à partir des langages synchrones nécessite une machine d'exécution pour pouvoir être utilisé en tant que programme [AND&PER - 93]. Ce code ne donne en effet que le comportement réactif du programme, et il faut lui fournir ses données, l'activer et traiter ses sorties pour obtenir un programme utilisable. Cette machine d'exécution a aussi pour rôle de déterminer la simultanéité des événements, c'est-à-dire de transformer les événements asynchrones du monde extérieur en événements synchrones pour le module et vice-versa.
De plus, le traitement d'un problème fait en général appel à différentes techniques de programmation, et la partie traitée par l'approche synchrone doit être intégrée aux autres parties. Cette intégration nécessite de définir une interface standard pour les modules synchrones, et de développer des outils d'intégration s'appuyant sur cette interface.

D'autre part, d'un point de vue méthodologique, la nécessité de structurer les gros problèmes pour pouvoir les traiter a mené à la définition de langages supportant la modularité et l'abstraction des types de données.
Dans ce contexte, la notion d'objet est apparue comme un bon paradigme pour la programmation de systèmes complexes, et les méthodologies de développement qui l'utilisent ont connu un certain succès.
Un des principaux intérêts de l'approche objet est de permettre de définir l'interface des entités d'un programme en faisant totalement abstraction de leur structure, donc des détails de leur implémentation. Il est donc intéressant de faire apparaître les entités synchrones d'une application sous la forme d'objets. Cela permet de faire abstraction de la mécanique interne qui leur donne un comportement synchrone, tout en accédant aux services rendus par cette mécanique au travers d'une interface standard. Cette intégration de modules synchrones dans le développement par objets d'une application permet de bénéficier des outils de l'approche synchrone pour vérifier les propriétés de chaque module, et des outils de l'approche objet pour décrire la structure globale de l'application.

Notre but est donc de fournir des outils permettant d'intégrer automatiquement des modules synchrones dans un langage à objets. Ces outils doivent aider le concepteur d'un système à éviter le maximum d'erreurs, et donc faire toutes les vérifications possibles à partir des informations dont ils disposent.

1.1. Problèmes traités

L'intégration de modules synchrones dans un langage à objets pose trois problèmes principaux:

- La *communication synchrone* entre objets synchrones. Des modules synchrones, une fois intégrés dans le langage à objets sous forme d'objets synchrones et interconnectés, doivent se comporter de manière synchrone. Ceci n'est pas évident a priori car leur interconnexion est faite dans le langage à objets. Il est donc nécessaire de mettre en place des mécanismes de commu-

nication entre objets synchrones qui respectent la sémantique synchrone sous certaines conditions, et des mécanismes qui permettent d'assurer que ces conditions sont vérifiées.

•La *communication asynchrone* entre les objets synchrones et les autres objets du programme. Il doit en effet y avoir échange d'information entre la partie synchrone de l'application et sa partie non synchrone. On peut concevoir des systèmes dans lesquels certaines tâches critiques sont assurées par des modules synchrones, leur comportement étant supervisé par des tâches asynchrones qui définissent une politique globale pour le système en les paramètrant. Les tâches asynchrones peuvent aussi exécuter, sur requête d'un objet synchrone, des opérations qui ne peuvent pas être considérées comme instantanées.

•La *dynamicité des objets synchrones*, qui permet d'en créer, d'en détruire et de modifier leur interconnexion au cours de l'exécution. Cette propriété permet de prendre en compte des situations dans lesquelles le nombre d'entités n'est pas connu a priori. En contrepartie, on ne dispose pas pour ces systèmes d'outils de vérification complets aussi puissants que pour les systèmes synchrones statiques.

On peut prendre un système de contrôle de circuit d'aérodrome comme exemple d'application faisant intervenir la dynamicité. Le nombre et le type des avions dans le circuit n'étant pas connu à l'avance, il faut réagir à leur arrivée au cours du fonctionnement du système. La détection d'un spot sur un écran radar, ou la réception d'un message radio provoquera la création d'un nouvel objet dans le système. A la fin de son transit dans la zone, l'avion n'intéressant plus le système, l'objet correspondant sera détruit. Au cours de son existence, il aura été en contact avec le centre d'approche, puis avec la tour de contrôle, et enfin, après l'atterrissage, avec le centre de gestion des appareils évoluant au sol, ce qui montre bien l'intérêt de la dynamicité des connexions entre objets synchrones.

La technique usuelle pour traiter de manière statique un nombre variable d'entités est de prévoir le nombre maximum que l'on aura à gérer, et de n'utiliser que celles qui sont effectivement présentes. Mais cette technique a l'inconvénient de manquer de souplesse: chaque "case" prévue pour représenter une entité doit être suffisamment générique pour recevoir dans notre exemple un planeur, un hélicoptère ou un Airbus. Dans le cas du traitement dynamique, l'objet créé dynamiquement correspond exactement à l'aéronef qu'il représente, et du point de vue de la maintenance, l'ajout d'une nouvelle classe d'appareils ne demande que la description de ce nouvel appareil. Dans le cas de l'approche statique, il faut modifier la structure des "cases" prévues pour correspondre à n'importe quel type d'appareil, ce qui est plus coûteux et peut conduire à des erreurs puisque l'on va modifier le code existant.

2. Les Langages Synchrones

Les langages synchrones permettent de programmer des systèmes réactifs de manière déterministe dans un langage de haut niveau tout en conservant de bonnes performances à l'exécution.
La notion de système réactif a été introduite [HAR&PNU - 85] afin de distinguer les systèmes qui réagissent au rythme de leur environnement des systèmes dits "transformationnels", qui disposent de toutes leurs entrées au démarrage et s'arrêtent lorsqu'ils ont produit leurs sorties, et des systèmes dits "interactifs", qui, bien qu'interagissant avec leur environnement, le font à leur rythme propre.

2.1. Les approches classiques

La programmation des systèmes réactifs peut se faire sous forme d'un automate déterministe dont les transitions sont activées par l'arrivée d'un événement. Cette approche a l'avantage d'être déterministe et de donner de bonnes performances à l'exécution. Par contre, la taille de l'automate peut devenir très grande dès que le système est un peu complexe. De plus, une petite modification dans

les spécifications du système peut conduire à un changement total de la structure de l'automate. Il est donc extrêmement difficile de maintenir de tels systèmes.
Les automates ne permettent pas d'exprimer le parallélisme qui est naturellement présent dans la description des systèmes réactifs. Il faut envisager toutes les exécutions parallèles possibles pour déterminer le nombre d'états et les transitions de l'automate.
A l'opposé, des langages parallèles de haut niveau comme ADA ou OCCAM offrent des mécanismes de communication et de synchronisation entre processus parallèles. On peut donc y exprimer plus naturellement le comportement de systèmes réactifs. Ils ont toutefois l'inconvénient de ne pas être déterministes. En effet, les instants auxquels les points de synchronisation entre processus sont rencontrés ne sont déterminés qu'à l'exécution.

2.2. L'approche synchrone

L'approche synchrone permet de programmer les systèmes réactifs dans un langage de haut niveau intégrant le parallélisme, tout en bénéficiant du déterminisme et de bonnes performances à l'exécution. Pour cela, les langages synchrones s'appuient sur l'hypothèse que la réaction du système à un événement est instantanée. Ainsi, le déroulement du temps n'est plus lié à l'exécution du programme, mais uniquement à l'occurrence des événements qu'il traite. Le temps physique est éliminé, il est remplacé par une succession d'événements. C'est ce que l'on appelle le temps multiforme pour rappeler que l'occurrence d'un événement peut aussi bien correspondre à des secondes qu'à des mètres, et pourtant faire progresser le temps du point de vue synchrone.
Une conséquence importante de l'hypothèse de synchronisme est la diffusion instantanée des valeurs des signaux. Ainsi, lorsqu'un processus émet le signal `minute` toutes les soixante occurrences du signal `seconde`, tous les processus perçoivent le signal `minute` au même instant, et l'émission de ce signal a lieu au même instant que la $60^{ème}$ occurrence du signal `seconde`. Ce n'est pas le cas dans un langage classique où un certain laps de temps s'écoule entre l'exécution de l'instruction qui détecte la $60^{ème}$ occurrence de la seconde et celle qui provoque l'occurrence de la minute.

Il existe plusieurs langages synchrones: Esterel [BER - 87] est un langage impératif dans lequel on décrit les actions à effectuer lorsqu'un événement arrive, alors que Lustre [HAL - 91] et Signal [BEN&LGU - 90] sont des langages déclaratifs dans lesquels on exprime directement le comportement du système sous forme d'équations entre les signaux. Il existe aussi des formalismes graphiques comme les Statecharts [HAR - 87] dont la sémantique n'est pas entièrement synchrone, et Argos [MAR - 90] qui corrige certains problèmes des Statecharts et a une sémantique synchrone.
Lustre et Esterel se compilent en automates finis qui sont décrits dans un format commun: OC. Signal se compile en ce que l'on appelle du code mono-boucle. Il s'agit d'une fonction qui, après les initialisations, exécute une boucle sans fin dont chaque tour correspond à un cycle du système. Signalons toutefois qu'un format commun aux langages synchrones déclaratifs, GC, et un compilateur de GC en OC, devrait permettre de produire du code OC à partir de Signal.
Le code OC de l'automate généré est utilisé pour faire des preuves de propriétés sur le système, et aussi pour produire le code du module, qui n'est autre que l'implémentation de l'automate dans un langage hôte (C, ADA, C++, etc.).

3. Les Langages à Objets

Les langages à objets permettent de regrouper une structure de données et les opérations qui la manipulent dans une même entité: l'objet.
La structure de données est cachée dans l'objet et n'est accessible qu'au travers d'un protocole, qui est l'ensemble des opérations que l'objet sait réaliser.

La structure d'un objet et son protocole sont définis dans sa classe. On peut considérer une classe comme un moule d'objets. Les classes peuvent être liées par une relation d'héritage: si une classe B hérite d'une classe A (on dit aussi que B dérive de A), elle hérite de son schéma d'instance et de son protocole. B peut ajouter des attributs au schéma d'instance hérité et définir de nouvelles opérations sur ces instances. Elle peut aussi redéfinir le comportement des opérations dont elle hérite. Par contre, B ne peut rien enlever au protocole hérité, ce qui garantit qu'une instance de B (ou objet de classe B) peut être utilisée à la place d'une instance de A puisqu'elle sait répondre aux mêmes messages.
Smalltalk-80, C++ et Eiffel sont des langages à objets. Nous avons choisi C++ car il s'intègre facilement à un environnement de développement et est assez répandu.

3.1. Intérêt de C++ pour l'intégration de modules synchrones

L'intégration de modules synchrones en C++ s'appuie à la fois sur ses propriétés de langage à objets et sur ses propriétés de langage fortement typé. Nous faisons aussi appel à la généricité à travers les gabarits de classes, ou types paramétrés. Il s'agit de classes qui sont définies en fonction d'un ou de plusieurs types muets. Si on définit le comportement d'une pile d'objets de type X dans le gabarit `Pile<X>`, on peut faire correspondre un véritable type au type muet pour obtenir une `Pile<entier>` ou une `Pile<complexe>`.
L'encapsulation dans un objet des données et des opérations sur ces données nous permet de donner la même interface à des modules synchrones, quelle que soit leur implémentation. Leur intégration dans une application peut donc se faire en ne s'appuyant que sur cette interface, qui se réduit aux noms et aux types de leurs signaux d'entrée et de sortie.
L'héritage nous autorise à définir le comportement général d'un objet synchrone, et à spécialiser ensuite ce comportement pour chaque classe implémentant un module particulier. Ainsi, les mécanismes qui permettent aux objets synchrones de réagir correctement sont construits en ne s'appuyant que sur ces comportements généraux. Ils conviennent donc à tous les objets synchrones.
Enfin, les types paramétrés nous permettent de spécifier la manière dont les objets synchrones échangent des valeurs de type quelconque, la vérification de la compatibilité de ces types étant faite automatiquement par le compilateur.
La possibilité de créer et de détruire des objets synchrones dynamiquement au cours de l'exécution d'un programme permet de traiter des systèmes dynamiques dans lesquels le nombre maximal d'entités d'un certain type n'est pas prévisible [BIH&GOP - 92]. La dynamicité des objets permet en effet de recycler les ressources de la machine hôte qui sont en nombre fini, et de les utiliser au mieux pour représenter les entités du système.

4. Présentation du système

Nous donnons ici une présentation informelle de notre système. Un modèle formel a été élaboré afin de servir de référence pour l'implémentation, mais sa description sort du cadre de cet article.
Cette présentation informelle montre comment, à partir de la notion de module synchrone et de celle d'objet, nous avons construit un système qui permet de construire des groupes d'objets dont le comportement est synchrone sous certaines contraintes.

4.1. Modules synchrones

Un module synchrone est une unité de compilation pour un langage synchrone. C'est la plus petite entité qui puisse être traitée par le compilateur.
Notre matière première est donc la forme compilée d'un module synchrone. Cette forme compilée est le code OC de l'automate produit par le compilateur Esterel. Mais rien n'interdit d'utiliser une autre forme compilée, par exemple du code monoboucle, ceci n'étant qu'un problème d'implémentation. L'utilisation de code OC produit par d'autres compilateurs de langages synchrones, Lustre

par exemple, ne nécessiterait qu'une mise à jour de notre traducteur d'OC vers C++ afin qu'il puisse prendre en compte les éventuelles particularités de ce code.
Les modules que nous intégrons dans le langage à objets étant produits par un compilateur de langage synchrone, ils bénéficient individuellement des techniques de vérification de l'approche synchrone.

4.2. Objets synchrones

Un objet synchrone est la traduction dans un langage à objets d'un module synchrone. Ce module, vu comme une boîte noire, se prête bien à l'encapsulation. Sa mécanique interne, produite par le compilateur de langage synchrone, est encapsulée dans un objet dont le protocole permet d'accéder aux signaux d'entrée et de sortie du module.
Ainsi, la forme compilée du module produite par le compilateur du langage synchrone est protégée des interventions extérieures et ne peut être manipulée qu'à travers le protocole de l'objet, qui assure sa cohérence.
Ceci apporte une sécurité supplémentaire pour le fonctionnement correct du module ainsi implémenté: son comportement ne risque pas d'être altéré accidentellement, il sera conforme à ce qui a été décrit dans le langage synchrone.

4.3. Classes synchrones

Nous transformons la forme compilée d'un module synchrone en une classe qui définit le comportement et la structure des objets synchrones qui implémentent ce module. Les classes ainsi produites sont appelées classes synchrones pour les distinguer des autres.
Il est possible d'instancier plusieurs fois une classe. On obtient alors des objets qui ont tous le comportement et la structure décrits par la classe, mais ont chacun leur propre état.
La création dynamique d'objets synchrones permet de construire des systèmes qui ne peuvent pas être décrits dans un langage synchrone. En contrepartie, les objets synchrones n'étant pas tous connus à la compilation, il n'est pas possible de faire globalement sur le système toutes les vérifications que fait un compilateur de langage synchrone. On ne peut que vérifier des propriétés sur le comportement individuel de chaque objet synchrone.
Nous nous attachons toutefois dans notre système à faire le maximum de vérifications. Certaines sont faites statiquement à la compilation (compatibilité des types de signaux) d'autres, comme la détection des boucles de causalité, sont faites à l'exécution.

4.4. Héritage entre classes synchrones

Comme les autres classes, les classes synchrones peuvent être liées par des relations d'héritage.
Pour nous, l'héritage est un moyen d'exprimer que le comportement d'une classe (la sous-classe) est un raffinement de celui d'une autre (la super-classe, ou classe de base). Donnons un exemple simple pour illustrer notre conception de l'héritage. On souhaite manipuler des figures géométriques, qui ont un centre et savent s'afficher.
Nous définissons donc une classe `Figure`, dont le protocole contient le message `affiche`, et le schéma d'instance comporte une variable `centre`. On fait ensuite hériter les classes `Carré` et `Cercle` de la classe `Figure`. `Carré` ajoute au schéma d'instance de `Figure` une variable `côté`, et `Cercle` y ajoute une variable `rayon`. Dans `Carré`, le message `affiche` active une méthode qui trace un carré de centre `centre` et de côté `côté`. Dans `Cercle`, le même message active une méthode qui trace un cercle de centre `centre` et de rayon `rayon`.
Dans un langage traditionnel, pour afficher une figure, il faudrait déterminer sa nature et appeler la procédure `afficheCarré` s'il s'agit d'un carré ou `afficheCercle` s'il s'agit d'un cercle.

Il nous faut maintenant définir comment se traduit l'héritage entre classes synchrones du point de

vue des modules synchrones.
Une classe synchrone B héritant d'une autre classe synchrone A devra correspondre à un module synchrone M_B dont le comportement est un raffinement de celui du module M_A. Les modules synchrones étant pour nous des boîtes noires, nous ne pouvons exprimer le raffinement de leur comportement qu'en modifiant les signaux qu'ils reçoivent ou qu'ils émettent.
A partir d'un module synchrone M_A, nous en construisons un autre M_B ayant au moins les mêmes signaux d'entrée et de sortie, et dont le comportement est obtenu en filtrant les signaux de M_A grâce à des modules synchrones auxiliaires.
Ainsi, à partir d'un module `Frein` ayant une entrée (la commande) et une sortie (la pression a appliquer sur les disques), on pourra définir un module `FreinABS` en filtrant la sortie de `Frein` grâce à un module `ContrôleABS`. Ce module utilise un capteur de glissement de la roue pour déterminer quand il faut limiter la pression de sortie, le module `FreinABS` aura donc un signal d'entrée supplémentaire pour le glissement. A ces modules correspondront les classes `Frein` et `FreinABS`, `FreinABS` héritant de `Frein`.
Ce mécanisme sera présenté plus en détail dans le chapitre "Outils de Développement" à propos de l'outil Mdlc.

4.5. Signaux et Connexions

Une fois les objets synchrones créés, il faut qu'ils puissent communiquer, c'est-à-dire que leurs signaux soit connectés.
Un signal peut être vu sous deux aspects: il établit une connexion entre deux objets synchrones, ce qui entraîne une relation de dépendance de l'objet consommateur envers l'objet producteur. D'autre part, un signal véhicule une valeur d'un type donné et peut être émis ou pas (il existe des signaux, dits "purs" qui ne véhiculent pas de valeur et sont simplement émis ou pas).
Un signal devra donc rendre un certain nombre de services: donner sa valeur, prendre une nouvelle valeur, se connecter à un autre signal etc. C'est pourquoi nous représentons les signaux sous forme d'objets.
Le protocole des signaux d'entrée comprend un message de connexion: on peut demander au signal d'entrée i de l'objet synchrone M de se connecter au signal de sortie o de l'objet synchrone N. Cette connexion est orientée, l'information circule du signal de sortie vers le signal d'entrée.
Nous utilisons les mécanismes de vérification de types du compilateur C++ pour détecter les connexions illégales: un signal d'entrée ne peut être connecté qu'à un signal de sortie véhiculant des valeurs de même type. L'existence de gabarits de classes permet de faire cette vérification alors que nous ne pouvons bien entendu pas connaître tous les types de valeur qui peuvent être véhiculés par des signaux.
Quant à la relation de dépendance entre objets synchrones induite par la connexion, sa validité, c'est-à-dire l'absence de boucle de causalité, ne peut être vérifiée qu'à l'exécution puisque les connexions sont dynamiques. Les méthodes de connexion des signaux font cette vérification avant de l'établir. Comme nous considérons le comportement synchrone des objets comme une boîte noire, nous n'avons aucune information sur les dépendances réelles entre les signaux de sortie et les signaux d'entrée. Nous considérons donc a priori que chaque signal de sortie d'un objet synchrone dépend de tous ses signaux d'entrée.
Se pose alors le problème du choix du comportement à adopter lorsqu'une connexion est refusée parce qu'elle crée une boucle. Ce problème est traité dans l'implémentation comme une erreur. Les éventuels traitements prévus pour pallier ce genre d'erreur ne sont que des mécanismes visant à assurer la sûreté du système, notamment lorsqu'on pilote un dispositif physique critique. Lorsqu'on ne peut pas prendre le risque qu'une telle erreur survienne à l'exécution, il faut soit faire la preuve que le comportement dynamique du système, d'après les contraintes qui lui sont imposées, ne peut pas la provoquer, soit se limiter à des systèmes statiques pour lesquels nous proposons un outil de

conception: Mdlc, qui sera présenté au paragraphe “Outils de Développement”.

4.6. Le Retard

Un retard est un objet qui a un signal d’entrée et un signal de sortie qui prend la valeur précédente du signal d’entrée. Autrement dit, à chaque fois que le signal d’entrée change, le signal de sortie prend l’ancienne valeur du signal d’entrée. On peut de plus spécifier une valeur initiale du signal de sortie.

Une propriété intéressante du retard est qu’il permet de “casser” les boucles de causalité. En effet, bien que le signal de sortie dépende du signal d’entrée, cette dépendance se fait avec un temps de retard: il n’est pas nécessaire de connaître la valeur actuelle du signal d’entrée pour connaître celle du signal de sortie, il suffit de connaître la valeur précédente.

On peut donc faire circuler de l’information le long d’une boucle sans provoquer d’erreur de causalité si cette boucle contient un retard [NAS - 92].

4.7. Les horloges

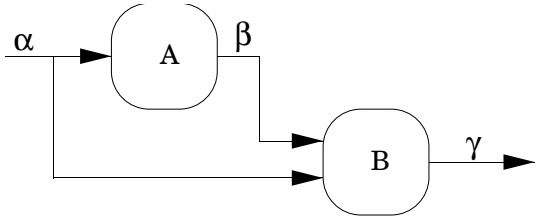
Nous disposons donc d’objets synchrones qui implémentent le comportement de modules synchrones, et nous savons connecter leurs signaux de façon à ce qu’ils puissent communiquer. Il nous reste à donner à ces objets interconnectés le comportement qu’ils auraient dans un langage synchrone. Il faut notamment qu’un signal ait la même valeur pour tous les objets synchrones qui y sont connectés.

Nous définissons pour cela les instants. Un instant est une topologie bien définie du système d’objets interconnectés associée à une valeur unique de leurs signaux. Pendant un instant, la valeur des signaux ne change pas, les connexions entre signaux ne changent pas, l’état des objets ne change pas et aucun objet n’apparaît ni ne disparaît. On peut considérer un instant comme une photographie du système d’objets. La réaction des objets à la valeur des signaux à un instant détermine l’instant suivant. Cette succession d’instants constitue une horloge.

Notre problème est de calculer la valeur des signaux à un instant d’une horloge afin de pouvoir en déduire la réaction des objets, et donc l’instant suivant.

Comme nous considérons que les signaux de sortie d’un objet dépendent a priori de tous ses signaux d’entrée, tous les signaux de sortie d’un objet doivent avoir été calculés (l’objet synchrone doit avoir réagit) avant que l’on puisse utiliser la valeur d’un de ces signaux. Ceci est rendu possible par l’absence de boucles dans le graphe de dépendance entre objets synchrones.

Figure 1:
Communication synchrone entre objets.



La figure ci-dessus montre deux objets synchrones A et B qui partagent un signal α. Comme la réaction de B dépend de la valeur du signal β, elle ne peut avoir lieu qu’une fois que A a réagit et a calculé la valeur de β. Quand B réagit, α doit avoir la même valeur que quand A a réagit.

Bien que la réaction de B doive avoir lieu après celle de A pour des raisons de dépendance, nous voulons qu’elle se déroule dans le même contexte que celle de A. Ce contexte correspond à la notion d’instant.

4.8. Dynamicité et Ordonnanceur

Les objets, signaux et connexions présents à un instant d'une horloge ne sont pas nécessairement les mêmes qu'à l'instant précédent puisque les horloges sont des systèmes dynamiques dont la topologie peut varier d'un instant à l'autre.

Les dépendances entre objets peuvent donc évoluer, et l'ordre de réaction des objets dans l'instant doit être déterminé dynamiquement à chaque instant.

Les modifications de la topologie de l'horloge (création d'objets synchrones, changements dans le réseau de connexion) ont lieu entre la fin de l'instant auquel elles ont été demandées et le début de l'instant suivant. En effet, par définition, la topologie de l'horloge ne peut pas changer pendant un instant.

La détermination de l'ordre dans lequel doivent réagir les objets synchrones d'un instant est confiée à un ordonnanceur. Cet ordonnanceur est chargé de déterminer un ordre de réaction des objets compatible avec leurs relations de dépendance, d'échantillonner les signaux du monde extérieur, de faire réagir les objets dans l'ordre qu'il a déterminé, et enfin de produire les signaux partant vers le monde extérieur. Chacun de ces cycles correspond à un instant de l'horloge.

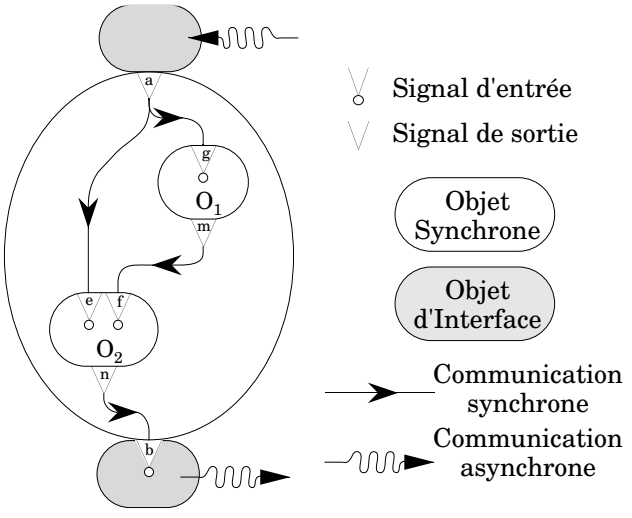
Les requêtes de modification de la topologie de l'horloge sont enregistrées par l'ordonnanceur pendant l'instant et lui permettent de construire la topologie de l'horloge à l'instant suivant. A partir de cette nouvelle topologie, l'ordonnanceur détermine un nouvel ordre de réaction pour les objets et le cycle continue.

4.9. Communications avec le monde asynchrone

Le monde extérieur à une horloge est considéré comme asynchrone: un événement peut y déclencher un processus dont la durée n'est pas négligeable devant la période de l'horloge [BER - 92], par exemple l'établissement d'une connexion sur un réseau. De plus, les événements du monde asynchrone ne se présentent pas sous la forme d'émissions de signaux et n'ont aucune raison de se produire uniquement aux instants de l'horloge synchrone.

La communication entre une horloge et le monde extérieur va donc faire appel à une interface [MAF - 91] qui permettra de traduire des événements de l'horloge en événements du monde asynchrone et réciproquement.

Figure 2:
Un instant d'une horloge.



Les objets constituant cette interface auront une double nature. D'un côté, ils seront sensibles à des modifications du monde asynchrone ou agiront sur ce monde en exécutant du code non-synchrone, de l'autre, ils présenteront une interface synchrone aux objets de l'horloge grâce à leurs signaux.
Bien que n'appartenant pas aux instants de l'horloge (puisque ce ne sont pas des objets synchrones), ils sont couplés à l'horloge de façon à ce que leurs signaux prennent une valeur aux instants de cette horloge.
Ces objets d'interface peuvent être utilisés comme échantillonneurs lorsque l'ordonnanceur génère les instants de l'horloge de façon périodique vis-à-vis du temps physique. Mais ils peuvent aussi déclencher le passage à l'instant suivant lorsqu'ils ont détecté un événement significatif. Ces deux modes de fonctionnement correspondent aux approches classiques par échantillonnage périodique ou par interruption.
L'approche objet permet de concevoir les classes d'interface indépendamment des modules synchrones, et de choisir le mode de fonctionnement le plus adapté à l'application sans avoir à modifier le code synchrone.

5. Outils de Développement

Notre modèle d'intégration de modules synchrones dans un langage à objets est supporté par des outils de développement. Ces outils permettent de produire des classes C++ à partir de modules synchrones, le moteur du modèle d'exécution étant fourni dans une bibliothèque de classes.

5.1. Occ++

Occ++ est un traducteur d'OC en C++ qui produit une classe pour chaque module présent dans le fichier OC. Il peut aussi produire un fichier qui décrit l'interface du module en mdl (Module Description Language) et pourra être utilisée par le constructeur de modules composites que nous présentons plus loin.
Les classes produites par occ++ dérivent de la classe `Esterel` qui est la classe des objets synchrones qui réagissent grâce à un automate. Ceci permet de ne mettre dans les classes synchrones que ce qui leur est spécifique (nom des signaux, tables de transition etc.) le moteur de l'automate étant fourni par la classe `Esterel`. Le traducteur occ++ est ainsi peu sensible à des modifications de la bibliothèque de classes qui implémente le moteur d'exécution. Il est donc possible de changer de modèle d'exécution à faible coût.

5.2. Mdlc

Mdlc est un outil de construction de modules synchrones. Le module à construire est décrit dans le langage mdl.
La construction de modules en mdl se fait par composition de modules, avec la contrainte qu'il ne doit pas y avoir de boucle dans le graphe de dépendance des modules. Ces boucles sont détectées et considérées comme des erreurs par mdlc.
La compilation du fichier de description par mdlc produit une classe synchrone qui peut être utilisée de la même façon que celles produites par occ++.
Pour pouvoir composer des modules, mdlc doit connaître leur interface. Cette information lui est fournie sous forme de fichiers de description de module qui peuvent être produit automatiquement par occ++ et mdlc.
Ces fichiers donnent le nom et le type des signaux du module qu'ils décrivent, et contiennent les informations permettant de retrouver son fichier source, l'éventuel fichier OC correspondant, ainsi que le fichier d'interface de la classe C++ qui l'implémente. Nous envisageons d'indiquer aussi les contraintes d'exclusion et de simultanéité sur les signaux, ce qui pourrait servir pour transformer les événements asynchrones en événements synchrones pour l'objet.

5.2.1. Modules composites

Un module composite est un module construit à partir d'autres modules interconnectés de manière statique. Ces modules composants peuvent avoir été décrits dans différents langages synchrones, voire être eux-même des modules composites ou des modules d'interface écrits en C++.

La description d'un module composite en mdl est constituée de déclarations de signaux, de modules, de connexions ou d'identités. Les déclarations de signaux indiquent le nom et le type des signaux d'entrée et de sortie du module composite. Les déclarations de modules indiquent le nom et le type des modules utilisés pour construire le comportement du module composite.

Il faut ensuite décrire comment les sous-modules sont interconnectés et à quoi correspondent les signaux d'entrée et de sortie du module composite. C'est le but des déclarations de connexions et d'identités. Les déclarations de connexion permettent de connecter un signal d'entrée d'un sous-module à un signal de sortie de même type d'un autre sous-module.

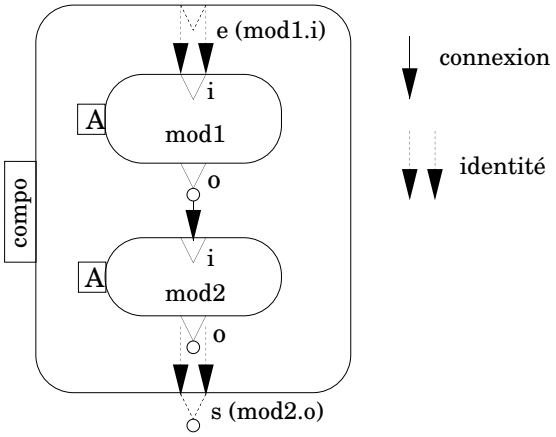
Les déclarations d'identité permettent de spécifier qu'un signal du module composite est en fait un signal d'un sous-module.

Supposons par exemple qu'il existe des modules de type `A`, ayant un signal d'entrée entier `i` et un signal de sortie pur `o`. On veut définir un module composite `compo` ayant un signal d'entrée `e` de type entier et un signal de sortie pur `s` à partir de deux modules de type `A`. On déclare pour cela:

```
compo : {
input: integer e;      // Signaux des modules compo
output: s;

A mod1, mod2;          // Sous-modules utilisés
mod1.i = e;            // e est l'entrée i de mod1
s = mod2.o;            // s est la sortie o de mod2
mod2.i << mod1.o;      // L'entrée i de mod2 est connectée
                       // à la sortie o de mod1.
}
```

Figure 3:
Exemple de module composite.



Il faut noter que l'opérateur d'identification '=' n'est pas commutatif. Il peut être vu comme une affectation: mod1.i reçoit la valeur de `e`. L'expression `e = mod1.i` provoquerait une erreur de compilation car le signal `e` étant une entrée du module `compo`, sa valeur est fixée par une source extérieure au module et ne peut pas être remplacée par celle de `mod1.i`. Nous préférons toutefois parler d'identification plutôt que d'affectation car le résultat de cette opération est que `mod1.i`

devient effectivement le signal `e` de `compo`. Ce signal est un alias de `mod1.i`, ce qui est figuré par sa représentation en pointillés sur la figure 3.
La classe synchrone `compo` produite par mdlc dérive de la classe `Composite`. En effet, la réaction des instances de `compo` ne se fait pas grâce à un automate. Elle se décompose en la réaction des sous-modules dans un ordre déterminé par mdlc. Ce type de réaction est supporté par la classe `Composite`. Les classes `Composite` et `Esterel` dérivent elles-mêmes de la classe `Synchronous` qui décrit le comportement commun à toutes les classes synchrones.

5.2.2. Modules dérivés

Un cas particulier de composition de modules est la dérivation. La dérivation d'un module à partir de son super-module est l'opération que nous faisons correspondre à la dérivation d'une classe à partir de sa super-classe dans le langage à objets.
Comme nous l'avons vu, l'interface du module dérivé doit inclure l'interface du super-module. Le comportement du module dérivé est obtenu en filtrant les entrées et les sorties du super-module à l'aide d'autres modules.
Prenons un exemple simple pour illustrer ce mécanisme: un module `frein` comporte un signal d'entrée `commande` et un signal de sortie `pression`, tous deux de type entier. Le signal `pression` varie en fonction du signal `commande` selon une loi adaptée au freinage.
On souhaite maintenant définir le comportement d'un module `frein_ABS`. L'utilisation de l'héritage pour définir `frein_ABS` à partir de `frein` est ici justifiée, car un `frein_ABS` est une sorte de frein particulière dans laquelle la valeur du signal `pression` est limitée en fonction d'un signal pur `glissement`, émis lorsque la roue glisse sur le sol. Le comportement du `frein_ABS` sera obtenu par filtrage de la sortie `pression` d'un frein grâce à un module `filtre_ABS`:

```
frein_ABS : frein {          // frein_ABS dérive de frein
input: glissement;     // Signal d'entrée supplémentaire
filtre_ABS filtre;     // Sous-module utilisé

filtre.pression_brute << pression;
                       // L'entrée du filtre est connectée
                       // à la sortie du module hérité.
filtre.glissement = glissement;
                       // L'entrée supplémentaire est
                       // celle du filtre.
pression = filtre.pression_filtree;
                       // La sortie est celle du filtre.
}
```

Le module filtre_ABS étant décrit par:

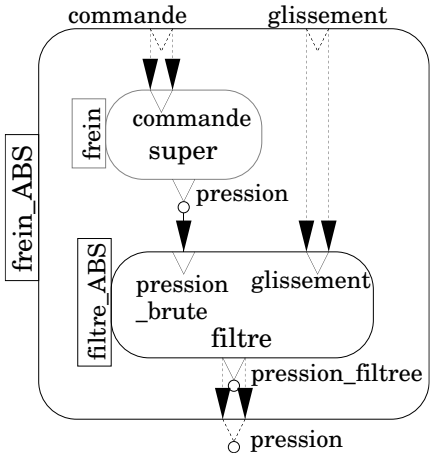
```
#source "filtre_ABS.strl"
#oc "filtre_ABS.oc"
#c++ "filtre_ABS.H"

filtre_ABS {
input: glissement;
input: integer pression_brute;
output: integer pression_filtree;
}
```

A partir de cette définition, mdlc produit la classe `frein_ABS` qui dérive de la classe `frein`, mais aussi de la classe `Composite` puisque sa réaction fait appel à celle d'autres objets synchrones.
La figure suivante montre la structure des instances de `frein_ABS`. L'objet de classe `frein` nommé `super` apparaît en pointillés car il correspond à la partie héritée du schéma d'instance de la

classe `frein_ABS` et n'est donc pas explicitement déclaré dans la définition.
On voit sur cette figure que le signal d'entrée `commande` de `frein_ABS` est celui de `super` alors que cette identité n'apparaît pas dans la description de `frein_ABS`. Ceci est dû au fait que les signaux de la sous-classe qui sont hérités de la super-classe sont identifiés par défaut à ceux de `super`.

Figure 4:
Structure des instances de la classe frein_ABS.

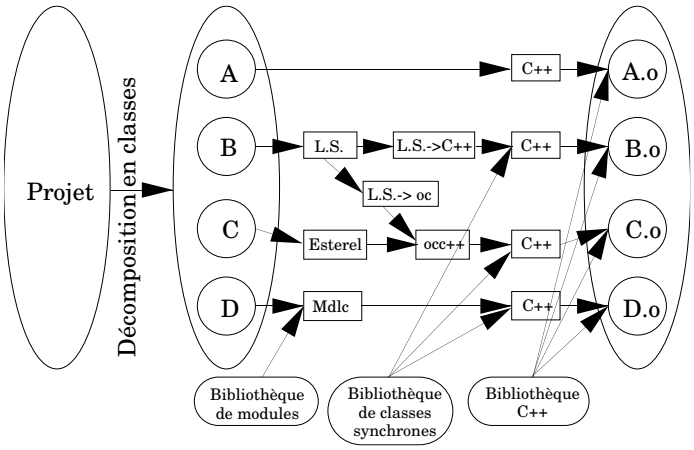


Il faut aussi remarquer que le même nom de signal peut désigner des signaux différents en fonction du contexte dans lequel il est utilisé. Ainsi, dans l'expression `filtre.pression_brute << pression`, pression désigne le signal de sortie de l'objet `super`. Mdlc accepte d'ailleurs la notation `super.pression` si l'homonymie pose problème au programmeur.
Dans l'expression `pression = filtre.pression_filtree`, `pression` désigne le signal de sortie du module `frein_ABS`. Mdlc lève l'ambiguïté entre le signal hérité et le signal du module grâce à l'utilisation qui est faite de ce signal.

5.3. Chaîne de développement

La figure suivante indique les différentes étapes du développement d'une application faisant intervenir du code synchrone:
Le projet est tout d'abord décomposé en classes. C'est dans cette étape que s'insère une éventuelle méthodologie de développement par objets. On choisit ensuite le langage dans lequel la classe va être codée. Ainsi, la classe A est directement codée en C++. La classe B est codée dans un langage synchrone qui peut être soit compilé directement en C++, soit compilé en code OC qui sera ensuite traduit en C++. Cette dernière solution correspond au cas de la classe C, codée en Esterel qui est pour l'instant le seul langage synchrone supporté par notre système (le code oc généré par Lustre n'a pas été testé avec occ++). La classe D est codée en mdl à partir d'une bibliothèque de modules synchrones.
La bibliothèque de classes synchrones fournit l'environnement d'exécution nécessaire aux classes synchrones produites par occ++ et mdlc.

Figure 5:
Chaîne de développement.



6. Conclusion

Le système présenté dans cet article permet de développer des applications faisant appel à des modules synchrones selon une méthodologie orientée objet. Ceci offre de nombreux avantages, tant sur le plan du prototypage que du développement séparé des diverses parties de l'application et de la maintenance.
Pour concevoir son application, l'utilisateur commence par déterminer, dans le cadre de sa méthodologie de développement, les classes qu'il utilisera et les services qu'elles devront rendre. Il choisit pour chaque classe l'approche qui convient le mieux: synchrone, objet, ou tout autre approche, puisque notre système se compose d'outils de développement et de bibliothèques et ne constitue pas un environnement de développement fermé.
Pour les classes qui doivent être traitées par l'approche synchrone, on écrit les modules dans un langage synchrone. On peut alors vérifier certaines propriétés de ces modules, les compiler en OC et obtenir les classes synchrones grâce à occ++.
Une classe synchrone peut aussi être produite par composition statique de modules synchrones grâce à l'outil mdlc. Cet outil permet aussi de construire des sous-classes de classes synchrones. La composition statique de modules autorise une forme de compilation séparée lorsque les modules ne dépendent pas cycliquement les uns des autres. Ceci permet de regrouper, dans un même objet synchrone, des modules écrits dans différents langages synchrones, ainsi que des modules d'interface écrits en C++.

Les classes synchrones ainsi produites peuvent être directement utilisées par l'application, ou être utilisées comme composants de classes plus complexes. On pourra par exemple définir une classe `voiture` en y intégrant quatre instances de `frein` et une instance d'`allumage_electronique`.
Les propriétés de chacune des classes synchrones peuvent être vérifiées grâce aux outils fournis par l'approche synchrone, l'intégration des modules synchrones en C++ ne modifiant pas leur comportement. De plus, la bibliothèque de classes synchrones a été conçue pour qu'un maximum de vérifications sur la bonne utilisation des objets synchrones soient faites automatiquement par le compilateur C++. Par contre, les propriétés globales du système synchrone pourront être difficiles à vérifier si ce dernier est dynamique et a donc un espace d'états variable au cours de son histoire. On peut toutefois espérer qu'au prix de certaines contraintes il soit possible de déterminer certaines propriétés des systèmes dynamiques.

L'interface des modules synchrones avec le monde asynchrone se faisant par des objets d'interface, il est facile de tester le comportement d'un objet synchrone en le connectant à une interface de simulation qui lui fournit des événements correspondant à la situation à tester comme s'ils provenaient du monde réel.

Les mécanismes du système sont implémentés sous forme de classes C++ et peuvent donc être complétés ou adaptés à un problème particulier.
Il est ainsi possible de définir de nouvelles classes d'interface et d'expérimenter de nouvelles méthodes de communication entre le monde synchrone et le monde asynchrone.

6.1. État actuel de l'implémentation

A l'heure actuelle, les outils occ++ et mdlc fonctionnent sous Ultrix 4.3 et sur Apple Macintosh sous MPW. Ils sont a priori portables sur n'importe quelle machine disposant d'un compilateur C et des bibliothèques standard.
La bibliothèque synchrone ne couvre pas actuellement tous les aspects du modèle. Elle a été compilée pour Ultrix 4.3 et pour DECelx 1.0 (version DIGITAL du noyau temps-réel VxWorks). Son portage sur d'autres plates-formes est conditionné par celui d'une bibliothèque de threads que nous n'avons actuellement créée que pour Ultrix (en utilisant les threads de la norme POSIX 1004.a) et pour DECelx (les threads étant natifs sur ce système).
Le code C++ de ces deux bibliothèques, ainsi que celui généré par occ++ et mdlc fait appel aux gabarits de classes. Il faut donc disposer d'un compilateur C++ qui les supporte. Cette contrainte nous a d'ailleurs obligés à porter le compilateur C++ d'ATT afin qu'il génère du code pour notre système temps-réel sous DECelx.
Ceci nous a permis de vérifier que notre système peut fonctionner dans un environnement temps-réel puisque nous avons développé une application gérant une maquette de feux de circulation (à base de LEDs et de boutons poussoirs), le comportement des feux étant écrit en Esterel, et l'interface avec le matériel étant écrite en C++. Le comportement des feux n'étant pas trivial (il y a trois modes de fonctionnement: automatique, manuel et clignotant, le mode pouvant changer en cours de fonctionnement), seule l'approche synchrone permettait de vérifier formellement que les feux ne sont jamais au vert en même temps. La preuve en a été faite par réduction du graphe de l'automate avec l'outil Auto.
Nous envisageons le développement d'un système comportant plusieurs maquettes, avec une tâche asynchrone pour analyser le trafic et paramètrer les feux de façon à optimiser le débit de voitures.

6.2. Perspectives

Il reste tout d'abord à terminer l'implémentation de la bibliothèque synchrone, avec passage à un nouveau modèle d'exécution plus simple et plus efficace. Nous envisageons ensuite de développer les mécanismes de communication entre tâches synchrones et asynchrones, ainsi que la communication entre tâches synchrones situées sur des processeurs distincts, ce qui ouvrirait notre système vers le développement d'applications distribuées.
Une autre voie de recherche est le développement de systèmes de communication adaptés au dynamisme des objets synchrones. En effet, lorsqu'on instancie des objets synchrones pour les connecter à des objets préexistants, se pose le problème de la disponibilité de signaux auxquels les connecter. Le nombre de signaux d'un objet est en effet fixe: on ne peut pas ajouter dynamiquement des signaux à un objet.
Reprenons l'exemple du contrôle de circuit d'aérodrome: à quoi sert de pouvoir créer dynamiquement des objets représentant les avions s'il est impossible de les connecter au centre de contrôle?
Une première solution est de rendre le centre de contrôle maître des communications. Il se connecte tour à tour avec chacun des objets, la communication se faisant à travers une file pour chaque aéro-

nef afin de ne pas perdre d'échantillons. Mais ceci ne permet pas de traiter les messages d'urgence.
Une autre solution est de multiplexer plusieurs signaux sur un même signal. On peut ainsi concevoir des connecteurs en Y qui multiplexent leurs deux entrées sur leur sortie. Mais cette technique introduit la notion d'échantillon multivalué, chaque échantillon d'un signal multiplexé étant constitué de l'ensemble des échantillons des signaux émis sur l'arbre binaire construit par les connecteurs en Y. Comment traiter de tels échantillons arborescents?
Enfin, mais cela sort du cadre de notre étude, il reste un problème théorique de fond: le développement de formalismes et d'outils permettant de traiter le comportement des systèmes dynamiques.
Le comportement de chaque objet synchrone étant totalement décrit, et l'évolution dynamique du système respectant le modèle que nous avons construit, nous pensons que sous certaines contraintes imposées à l'histoire du système, il doit être possible de vérifier formellement des propriétés de ce système.

7. Références bibliographiques

[AND&PER - 93] : Charles André et Marie-Agnès Peraldi
Effective Implementation of ESTEREL Programs
Workshop on Real-Time Systems, Euromicro'93, Oulu (Finland), June 1993

[BEN&BER-91] : Albert Benveniste, Gérard Berry
The Synchronous Approach to Reactive and Real-Time Systems
Proceedings of the IEEE, vol. 79, n° 9, September 1991

[BEN&LGU-90] : Albert Benveniste, Paul le Guernic
Hybrid Dynamical Systems Theory and the SIGNAL Language
IEEE Transactions on Automatic Control, vol. 35, n°5, May 1990

[BER-87] : Gérard Berry, Philippe Couronne, Georges Gonthier
Synchronous Programming of Reactive Systems : An Introduction to ESTEREL
Rapport de recherche n°647, INRIA - Sophia Antipolis, mars 1987

[BIH&GOP - 92] : Thomas E. Bihari et Prabha Gopinath
Object-Oriented Real-Time Systems: Concepts and Examples
Computer, IEEE Computer Society, December 1992

[HAL-91] : Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud
The Synchronous Data Flow Programming Language LUSTRE
Proceedings of the IEEE, vol. 79, n°9, September 1991

[HAL - 92] : Nicolas Halbwachs, Fabienne Lagnier et Christophe Ratel
Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE
IEEE Transactions on Software Engineering, vol. 18, n° 9, September 1992

[HAR - 87] : D. Harel
Statecharts: A visual approach to complex systems
Science of Computer Programming, vol. 8, n° 3, 1987

[HAR&PNU-85] : D. Harel, A. Pnuelli
On the Development of Reactive Systems
Weizmann Institute of Science, Rehovot, Israel, 1985

[MAR - 90] : F. Maraninchi
Argos, un langage graphique pour la conception, la description
et la validation des systèmes réactifs.
Thèse, Université Joseph Fourier, Grenoble, 1990

[NAS - 92] : Eric Nassor
Modélisation et validation d'applications temps réel distribuées
Thèse de doctorat, Orsay, Paris XI, 1988