ORSAY

n° d'ordre : 2977

UNIVERSITÉ DE PARIS-SUD
CENTRE D'ORSAY

ÉCOLE SUPÉRIEURE
D'ÉLECTRICITÉ

THÈSE

présentée
pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

par

Frédéric BOULANGER

Spécialité : Informatique

SUJET :

Intégration de Modules Synchrones dans la Programmation par Objets

soutenue le 15 décembre 1993 devant la commission d'examen

Mme	Marie-Claude GAUDEL	Président
MM.	Paul CASPI Olivier ROUX	Rapporteurs
MM.	Charles ANDRÉ Paul Le GUERNIC Guy VIDAL-NAQUET	Examinateurs

Remerciements

Je tiens à remercier :

Madame Marie-Claude Gaudel de m'avoir fait l'honneur de présider ce jury, de m'avoir accueilli dans son équipe au Laboratoire de Recherche en Informatique et pour le temps qu'elle a consacré à ce travail malgré ses nombreuses obligations.

Monsieur Paul Caspi et Monsieur Olivier Roux, qui ont accepté la lourde tâche de rapporteurs et dont j'ai particulièrement apprécié les remarques constructives sur mon travail.

Monsieur Charles André, pour l'intérêt qu'il a porté à ce travail, et pour sa participation à ce jury.

Monsieur Paul Le Guernic, pour les discussions fructueuses que nous avons eues sur l'approche synchrone alors que je commençais cette thèse.

Monsieur Guy Vidal-Naquet, qui a toujours suivi avec intérêt mon travail en tant que directeur de thèse, m'a donné de nombreuses idées, et ne m'a jamais compté son temps, même lors de son détachement à la société Alcatel Alsthom Recherche.

Monsieur Nicolas Mercouroff, Monsieur Henri Delebecque et Monsieur Dominique Marcadet qui m'ont apporté une aide précieuse.

Monsieur Jean-Philippe Szylowicz, qui m'a fourni le cadre pour effectuer cette thèse dans d'excellentes conditions au Service Informatique de l'Ecole Supérieure d'Electricité.

Tous mes collègues du Service Informatique de l'Ecole Supérieure d'Electricité, qui m'ont permis de travailler dans une ambiance sympathique tout au long de cette thèse.

Le Service de Reprographie de l'Ecole Supérieure d'Electricité, qui a réalisé le tirage et la reliure de cette thèse « en temps-réel ».

Table des matières

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 formaliser le comportement des systèmes réactifs. Des outils mathématiques peuvent alors être utilisés pour détecter des incohérences dans les programmes et prouver des propriétés des systèmes ainsi programmés. Il reste à vérifier a posteriori que l'hypothèse de synchronisme était justifiée, c'est-à-dire que les temps de calculs sont suffisamment courts pour que la réaction à un événement soit terminée avant l'arrivée du suivant. 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 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.

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. Il est donc intéressant 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 dé finir l'interface des entités d'un programme en faisant totalement abstraction de leur structure, donc des détails de leur implé mentation.

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.

But poursuivi

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.

Certaines de ces vérifications portent sur des propriétés statiques du programme (types de données, définition de comportements) et peuvent être faites a priori, c'est-à-dire lors de la compilation. D'autres propriétés permettent d'éviter les erreurs en respectant l'intuition du concepteur : le comportement des modules d'un programme doit être conforme à ce que le programmeur attend d'après le modèle de programmation qu'il emploie. Ainsi, le comportement d'un système d'objets synchrones interconnectés dans le langage à objets doit être celui qu'aurait le même système décrit dans un langage synchrone.

Enfin, l'encapsulation de modules synchrones dans des objets permet d'en créer dynamiquement au cours de l'exécution du programme. Ceci permet de construire des systèmes synchrones dynamiques, ce qui n'est pas possible dans un langage synchrone. En contrepartie, on ne dispose pas pour ces systèmes d'outils de vérification complets aussi puissants que pour les systèmes synchrones statiques. Nous devons donc donner un modèle du comportement de ces systèmes dynamiques que l'implémentation devra respecter. La vérification de la conformité du programme à ce modèle ne peut être faite qu'à l'exécution pour ce qui est des propriétés liées à la dynamicité.

Afin de ne pas obliger l'utilisateur à payer le prix de la dynamicité lorsqu'il n'en a pas besoin, nous fournissons aussi un outil de construction de systèmes statiques. Grâce à cet outil, il est possible d'intégrer dans le langage à objets un système statique de modules synchrones (à condition qu'il n'y ait pas de cycle dans les dépendances entre modules).

L'avantage de cet outil par rapport à la composition des mêmes modules dans un langage synchrone est qu'il autorise la composition de modules écrits dans des langages synchrones différents, et même de pseudo-modules synchrones, écrits dans un langage à objets, et chargés de faire l'interface entre les modules synchrones et leur environnement.

Problèmes traités

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

- La communication synchrone entre objets synchrones. Les modules synchrones, une fois intégrés dans le langage à objets sous forme d'objets synchrones, 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 communication 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. Une tâche asynchrone peut aussi surveiller l'état d'un système synchrone, détecter les situations potentiellement dangereuses et agir sur le système pour éviter la catastrophe.

  Une dernière application de la symbiose entre tâches synchrones et asynchrones est le traitement des opérations qui prennent du temps par des tâches asynchrones sur requête de tâches synchrones.

- La dynamicité des objets synchrones, qui permet d'en créer, d'en détruire et de modifier leurs interconnexions 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. On se rapproche alors de la notion d'acteur, présentée dans [HEW&BAK-77].

On peut prendre un système de contrôle de circuit d'aérodrome comme exemple d'application faisant intervenir la dynamicité. Le nombre d'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 ou de son roulage au parking, 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. On voit bien ici 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. Or la réaction du centre de

contrôle sera différente pour chacun de ces aéronefs puisqu'elle doit s'adapter à leurs possibilités d'évolution, aux contraintes d'espacement dues à la masse d'air qu'ils déplacent et aux contraintes de sécurité qui s'imposent pour chacun d'eux en cas d'incident. Dans le cas du traitement dynamique, le centre de contrôle traite avec des « aéronefs », chacun d'entre eux étant un objet spécifique avec ses propres caractéristiques et comportements, tout comme l'appareil qu'il représente. Le centre de contrôle n'a qu'à demander sa vitesse minimale d'évolution à un appareil pour savoir s'il va pouvoir rester derrière le précédent. En toute rigueur, il n'a même pas besoin de savoir de quel type d'appareil il s'agit. De plus, 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 dynamique. 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 du code qui fonctionnait jusqu'ici.

Plan

Nous présenterons tout d'abord les langages synchrones, leurs propriétés, les problèmes qu'ils permettent de traiter ainsi que les vérifications qu'ils permettent de faire sur les programmes. Nous insisterons plus particulièrement sur Esterel [BER-87] puisque c'est ce langage qui a servi de base à l'implémentation de notre système.

Nous donnerons ensuite un aperçu des langages à objets, en détaillant les propriétés sur lesquelles nous nous appuyons. Nous présenterons plus en détail certains aspects de C++ [STR-91] sur lesquels s'appuie notre implémentation.

Le chapitre III donnera une présentation informelle des problèmes traités et des solutions que nous y apportons. Ce chapitre permettra d'aborder les suivants en ayant une vue d'ensemble du système.

Nous présenterons ensuite le modèle que devra respecter l'implémentation. Ce modèle se décompose en deux parties :

- le modèle temporel, qui définit la notion d'objet synchrone et l'environnement temporel dans lequel il réagit : l'instant. Une horloge y sera définie comme une suite infinie d'instants. Ce modèle donne aussi les contraintes sur l'appartenance d'un objet aux instants d'une horloge.

- le modèle objet, qui définit la notion de classe synchrone, d'héritage entre classes synchrone et exprime les propriétés des objets synchrones en fonction de ceux de leur classe. Une classe synchrone définit le comportement de toute une famille d'objets synchrones, appelés ses

instances. L'héritage entre classes synchrones est défini comme un raffinement de ce comportement, que nous exprimons par le filtrage des signaux d'entrée et de sortie du comportement de base.

Le chapitre VI complète le modèle en indiquant comment les opérations sur les objets (instanciation, destruction et connexion de signaux) se traduisent par des modifications de la topologie des horloges. Ce modèle des interactions entre le modèle objet et le modèle temporel décrit l'aspect dynamique des objets synchrones. Il donne les règles que doit suivre l'évolution d'un système dynamique d'objets synchrones afin que chaque instant de son horloge puisse être défini.

Ce modèle sert de référence pour l'implémentation, et nous espérons pouvoir l'utiliser par la suite pour faire des preuves sur les systè mes dynamiques sous certaines contraintes. Le chapitre VII donnera ensuite le modèle d'exécution retenu pour que les objets synchrones aient le comportement décrit par le modèle.

Nous passerons alors à l'implémentation en présentant au chapitre VIII les outils de développement qui permettent d'intégrer les modules synchrones en C++ et de construire de nouveaux modules à partir de modules déjà traduits en C++, ce qui autorise une forme de compilation séparée sous certaines conditions.

La bibliothèque de classes qui fournit le support d'exécution des objets synchrones sera présentée au chapitre IX, « Implémentation ». Nous y décrirons les mécanismes qui permettent de respecter le modèle de notre système et nous donnerons l'interface des classes de la bibliothèque synchrone. Ceci permettra de mieux saisir les interactions entre les différentes entités qui permettent aux objets synchrones de réagir correctement et de communiquer avec leur environnement asynchrone.

Un exemple d'utilisation d'objets synchrones en C++ est donné au chapitre X. Cet exemple montre comment un objet synchrone est instancié dans une horloge et comment les signaux sont connectés. Il illustre aussi l'utilisation des objets d'interface puisque les entrées et les sorties du système se font sur une console.

Le chapitre XI donne un exemple de communications asynchrones entre objets situés sur des horloges différentes. Le comportement obtenu est comparé à celui des mêmes objets instanciés sur une même horloge. Cet exemple illustre les modes de communication asynchrone par valeur la plus récente et avec historique.

Le chapitre XII donne un exemple d'instanciation dynamique d'objets synchrones.

La conclusion rappelle les caractéristiques du système qui sont :

– l'adaptabilité à différentes méthodes de développement. Les outils que nous proposons respectent à la fois les formalismes de l'approche synchrone, ce qui permet d'utiliser les outils classiques pour la vérification

des modules, et ceux de l'approche objet, ce qui permet d'intégrer ces outils dans le cadre d'une méthodologie.

– l'ouverture du système, qui autorise le prototypage et l'extension de ses possibilités. En effet, tous les mécanismes mis en jeu sont accessibles au travers des classes de la bibliothèque synchrone, et peuvent donc être enrichis ou étendus. Il est ainsi possible d'expérimenter de nouvelles techniques de communication entre objets synchrones et tâches asynchrones sans avoir à modifier les outils fournis.

Nous indiquons l'état actuel de l'implémentation et donnons les directions vers lesquelles s'orientent nos travaux, le but général étant une meilleure gestion de la dynamicité des objets synchrones.

Les références bibliographiques permettent de retrouver les articles cités dans le corps de la thèse, et la bibliographie indique les documents qui ont servi pendant le développement du système, bien qu'ils ne soient pas cités, soit que leur apport ne se limite pas à un point précis, soit qu'ils viennent en complément d'un article cité.

Enfin, les annexes contiennent les manuels de référence des outils de développement.

I Les Langages Synchrones

Les langages synchrones ont été développés pour permettre de programmer des systèmes réactifs de manière déterministe dans un langage de haut niveau tout en conservant de bonnes performances à l'exécution.

Voyons tout d'abord ce que l'on entend par « système réactif », et quelles sont les approches classiques pour traiter ces systèmes.

I.1 Les systèmes réactifs

Un système réactif est un système dont la tâche principale est de maintenir une interaction avec son environnement. Autrement dit, il assure le respect de certaines contraintes entre ses entrées et ses sorties.

Un tel système doit donc réagir à son environnement au fur et à mesure que ce dernier évolue. C'est pourquoi la notion de système réactif a été introduite [HAR&PNU-85] afin de les distinguer 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.

Cette catégorie de systèmes englobe la plupart des systèmes temps-réel (automatismes, traitement du signal), mais aussi les interfaces homme-machine ou les protocoles de communication sur un réseau. Ces systèmes sont en général déterministes, doivent obéir à des contraintes de temps et de fiabilité strictes, et font intervenir le parallélisme.

Un système réactif est déterministe car ses sorties sont entièrement déterminées par ses entrées et par son histoire, c'est-à-dire par la suite d'événements qu'il a reçu. La spécification d'un tel système étant déterministe, il est intéressant de conserver ce déterminisme lors de son implémentation car le comportement d'un système déterministe est beaucoup plus facile à appréhender, donc à analyser et à diagnostiquer, que celui d'un système non-déterministe.

Les contraintes temporelles que doit respecter un système réactif proviennent du fait qu'il doit réagir au rythme de son environnement. Il doit donc non seulement être capable d'enregistrer les événements au fur et à mesure qu'ils se produisent, mais aussi d'y réagir suffisamment rapidement pour que son influence sur son environnement ne soit pas trop décalée par rapport à ses entrées.

Les contraintes de fiabilité proviennent de l'utilisation qui est faite des systèmes réactifs. Ils interviennent en effet dans le contrôle de dispositifs physiques qui peuvent être critiques : une erreur dans le système de conduite

automatique d'un train ou de pilotage automatique d'un avion peuvent être dramatiques. De tels dispositifs sont en général conçus de manière à ce que la sécurité soit assurée en cas de défaut de fiabilité (un train s'arrête si le système de conduite tombe en panne, un avion garde sa ligne de vol si le pilote automatique est défaillant). Mais le coût d'une panne peut être très élevé, sans compter que la sécurité assurée par la conception du système peut être toute relative, voire ne pas exister (certains avions de chasse sont intrinsèquement instables, afin d'avoir une grande maniabilité, et ne gardent leur ligne de vol que grâce à des systèmes de contrôle). Il est donc important de pouvoir vérifier formellement les propriétés d'un système réactif afin d'éliminer une source d'erreur potentielle.

Enfin, les systèmes réactifs font intervenir le parallélisme parce que leur conception se fait naturellement par assemblage de composants fonctionnant en parallèle, chaque composant assurant le maintient d'une propriété entre ses entrées et ses sorties. Cette manière de programmer les systèmes réactifs est analogue à la conception de systèmes similaires à l'aide de boîtiers électroniques. De plus, pour ces systèmes, le monde extérieur doit être considéré comme un processus concurrent qui évolue selon ses propres lois. Ainsi, même le système réactif le plus simple met en jeu deux processus en parallèle.

I.2 Les approches classiques

La programmation des systèmes réactifs peut être abordée de deux façons extrêmes. Une première approche consiste à décrire le système sous forme d'un automate déterministe. L'automate étant dans un certain état, l'arrivée d'un événement provoque une transition qui active une procédure correspondant au comportement à adopter, et place l'automate dans un nouvel état.

Cette approche a l'avantage d'être déterministe et de donner de bonnes performances à l'exécution. En effet, la réaction du système se traduit par une transition de l'automate et par l'exécution de code purement linéaire dont la durée d'exécution peut être majorée. De plus des techniques et des outils de vérification existent sur les automates, ce qui permet de faire des vérifications formelles sur le comportement des systèmes que l'on programme ainsi.

Par contre, la taille de l'automate peut devenir très grande dès que le système est un peu complexe. Il devient alors très difficile d'en avoir une vue globale et donc de le construire. De plus, une petite modification dans les spécifications du système ne se traduit en général pas par une modification localisée de l'automate, mais peut conduire à un changement total de sa structure. 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. Toutes les exécu-

tions parallèles possibles doivent être envisagées 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, et ont l'avantage d'être portables. On peut donc y exprimer plus naturellement le comportement de systèmes réactifs. Ils ont toutefois l'inconvénient de ne pas être déterministes : le comportement d'un programme ne peut pas être déduit uniquement de son code source. En effet, les instants auxquels les points de synchronisation entre processus sont rencontrés ne sont déterminés qu'à l'exécution.

De plus, les instructions qu'ils fournissent pour traiter le temps ont souvent une sémantique imprécise. Ainsi, l'attente d'un délai de 5 secondes durera en général au moins 5 secondes, et sa durée exacte ne sera pas la même d'une exécution à l'autre.

I.3 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.

Cette absence de durée d'exécution permet de considérer que deux instructions peuvent s'exécuter effectivement au même instant. C'est ce qui permet de simplifier le traitement du parallélisme et de le rendre déterministe puisque l'on n'a pas à envisager tous les entrelacements possibles pour l'exécution d'instructions en parallèle : ces instructions s'exécutent au même instant.

Une autre conséquence importante de l'hypothèse de synchronisme est la diffusion instantanée des valeurs des signaux. Des instructions placées en parallèle s'exécutant au même instant, si une instruction donne une valeur à un signal à cet instant, toutes les autres instructions ont accès à la même valeur. 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^{e}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^{e} occurrence de la seconde et celle qui provoque l'occurrence de la minute.

Plusieurs langages synchrones ont été développés, chacun ayant ses particularités. Ainsi, 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 construit un réseau d'opérateurs à travers lequel vont circuler les événements.

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 précise et synchrone. Ces deux formalismes s'appuient sur la mise en parallèle et la hiérarchisation d'automates. La hiérarchisation d'automates consistant à représenter le comportement d'un automate dans un certain état par un autre automate.

I.3.1 Lustre et Signal

Ces deux langages permettent de programmer des systèmes synchrones selon une approche par flots de données et restent ainsi assez proches des méthodes d'analyse habituelles en automatique comme les systèmes d'équations aux différences finies.

Dans ce modèle, un système est représenté par un réseau d'opérateurs fonctionnant en parallèle et activés par leurs signaux d'entrée. Selon l'hypothèse de synchronisme, la réaction de ces opérateurs est instantanée. Les signaux entrant dans le réseau le traversent donc instantanément pour produire les signaux de sortie.

Cette approche se prête bien à une représentation graphique structurée du système. On peut en effet construire des opérateurs complexes à partir des opérateurs de base, et obtenir ainsi différentes représentations du système selon le niveau de détail que l'on souhaite. Les signaux de Lustre et de Signal sont des suites de valeurs associées à une horloge qui définit les instants auxquels les valeurs sont présentes. Le réseau d'opérateurs est construit en écrivant des équations entre les signaux. Ces équations indiquent l'égalité des horloges de leurs deux membres ainsi que des valeurs qu'ils prennent aux instants de cette horloge commune.

Une des grandes différences entre Lustre et Signal réside dans la nature de ces équations. Lustre est fonctionnel : les équations indiquent comment obtenir les signaux de sortie en fonction des signaux d'entrée, alors que Signal est relationnel : les équations définissent une relation entre les signaux d'entrée et les signaux de sortie. Ceci se traduit par le fait qu'en Lustre, les équations n'expriment pas de contraintes sur les signaux d'entrée à partir des signaux de sortie, alors qu'en Signal, de telles contraintes peuvent être exprimées. Il est

ainsi possible en Signal de sur-échantillonner un signal, c'est-à-dire de produire des sorties à un rythme plus élevé que celui des entrées, et de forcer une entrée à être présente toutes les x occurrences d'un signal de sortie.

I.3.2 Esterel

Esterel est le plus ancien des trois langages synchrones que nous présentons. Il s'agit d'un langage parallèle impératif dans lequel les processus communiquent instantanément par signaux. A un signal sont associées deux informations : la valeur, qui est rémanente, et les tops, qui sont transitoires et indiquent les instants auxquels le signal est émis. Il existe des signaux dits « purs » qui n'ont pas de valeur, et peuvent donc uniquement être émis ou non. Lorsqu'un signal est valué, sa valeur ne change que lorsqu'un top est émis. Ce top est utilisé pour propager et détecter la nouvelle valeur.

L'unité de compilation en Esterel est le module. Un module peut lui-même être composé d'autres modules qui communiquent entre eux grâce à des signaux locaux. Il est ainsi possible de structurer les modules complexes.

En plus des signaux d'entrée et de sortie, un module peut communiquer avec son environnement par des senseurs. Un senseur a une valeur, mais pas de top. Il est donc impossible de détecter ses changements de valeur. Les senseurs sont utiles pour obtenir des informations dont on a besoin pour traiter un événement, mais dont les variations ne constituent pas un événement pour le système.

En Esterel, un signal n'est pas défini par une équation, mais par son émission par un processus, il est donc possible que plusieurs processus émettent le même signal au même instant. Pour un signal pur, ceci ne pose pas de problème : si le signal est émis plusieurs fois dans un instant, il est simplement considéré comme émis à cet instant.

Par contre, pour les signaux valués, les valeurs avec lesquelles le signal est émis ne sont pas forcément les mêmes pour tous les processus. On utilise dans ce cas la fonction de combinaison associée au signal pour déterminer la valeur du signal en fonction des différentes valeurs émises. Si un signal valué n'a pas de fonction de combinaison, il est interdit de l'émettre plusieurs fois dans un instant. Les émissions multiples sont alors considérées comme des erreurs par le compilateur.

Esterel étant un langage impératif, il dispose de structures de contrôle. Comme il s'agit aussi d'un langage synchrone, le contrôle ne prend pas de temps. Ainsi, le choix de la branche à exécuter dans une structure conditionnelle étant instantané, l'exécution de cette branche commence à l'instant même auquel la condition est évaluée. Il est donc possible d'exprimer des paradoxes comme : si un signal `s` n'est pas émis, émettre `s`, ce qui se traduit par : si `s` n'est pas émis, il est émis. Le compilateur Esterel détecte ces paradoxes qui peuvent être provoquées par une chaîne causale beaucoup plus longue

que dans notre exemple, et faisant intervenir plusieurs processus en parallèle. Il les traite comme des erreurs puisqu'aucune exécution ne correspond à ces expressions. L'origine de telles erreurs peut être difficile à trouver dans le code du programme.

En plus des structures de contrôles classiques (mise en séquence, mise en parallèle, boucle, conditionnelle) Esterel dispose d'un mécanisme d'exceptions. Ce mécanisme permet d'interrompre un processus pour traiter une exception. L'exception peut être levée par un processus s'exécutant en parallèle avec le processus interrompu, ce qui en fait un mécanisme de contrôle très puissant.

Les instructions d'Esterel peuvent s'exprimer à partir de trois primitives : l'émission d'un signal `s` (`emit s`), le test de la présence d'un signal `s` (`present s then ... else ... end`) et l'exécution d'une instruction jusqu'à l'occurrence d'un signal `s` (`do ... watching s`).

Cette dernière primitive limite l'exécution d'une instruction à la date à laquelle le signal `s` est émis : si l'instruction se termine avant l'occurrence de `s`, le watching se termine au même instant. Par contre, si `s` est émis avant que l'instruction se termine, elle s'exécute jusqu'à l'instant, exclus, auquel `s` est émis.

Récemment [BER-92, AND-91] une primitive a été ajoutée à Esterel pour traiter les interactions entre un processus synchrone et une tâche asynchrone. Cette primitive, exec, permet d'exécuter une tâche asynchrone tout en respectant la sémantique des autres constructions du langage. Ainsi, cette exécution sera interrompue lorsque `s` est émis si elle est placée dans le corps d'un `do ... watching s` ou si une exception vient interrompre l'exécution de l'exec.

Comme Lustre et Signal, Esterel ne dispose que d'un ensemble de types et d'opérateurs restreint (entiers, booléens, opérateurs logiques et arithmétiques). Les autres types, constantes, procédures et fonctions doivent être définis dans le langage hôte et sont déclarés dans le module de façon à ce qu'Esterel puisse faire les vérifications de types et de conformité des appels au prototype des procédures et fonctions.

I.4 La vérification de programmes

Lustre et Esterel se compilent en automates finis qui sont décrits dans un format portable : 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.

La vérification des programmes Lustre et Esterel s'appuie sur l'automate généré lors de la compilation. En effet, les propriétés que l'on souhaite vérifier pour un système réactif sont en général des propriétés de sécurité. On souhaite vérifier que telle situation ne peut jamais se produire, ou que telle expression booléenne est toujours vraie.

Ceci peut se faire en démontrant qu'un état de l'automate n'est pas atteignable, ou que l'expression booléenne est vraie dans tous les états atteignables.

Le problème qui se pose rapidement est le grand nombre d'états de l'automate. Il est donc nécessaire de réduire le nombre d'états de l'automate utilisé pour la preuve.

En Lustre, il est possible d'exprimer des assertions dans le programme. Ces assertions sont supposées être toujours vraies et utilisées par le compilateur pour optimiser le code généré. Un mécanisme similaire, bien que moins puissant, existe en Esterel et permet d'exprimer que certains signaux ne sont jamais émis en même temps, ou au contraire, que l'émission d'un signal implique celle d'un autre. Grâce à ces informations, le nombre d'états de l'automate est réduit, mais ce n'est pas toujours suffisant pour pouvoir faire une preuve.

Pour les programmes Esterel, on utilise l'outil Auto pour réduire l'automate en ne considérant que les signaux qui interviennent dans la propriété à prouver. On peut alors considérer certains états comme équivalents et l'on obtient un automate réduit sur lequel on peut faire la preuve.

En Lustre, on ajoute au programme un processus qui calcule la propriété à prouver, et l'on ne considère que la sortie booléenne de ce processus. L'automate généré est en général petit puisque tout ce qui n'influe pas sur la valeur de la propriété est éliminé. Il suffit alors de vérifier que cette valeur est bien celle que l'on veut dans tous les états de cet automate.

En Signal, on emploie une technique similaire en conjonction avec l'outil de preuve Sigali. On définit pour cela un signal qui est émis quand la propriété du programme n'est pas vérifiée, et il ne reste qu'à vérifier que son horloge est vide. Cette technique fait appel au calcul d'horloge de Signal qui consiste à résoudre un système d'équations dans l'ensemble $Z/3Z$ des entiers modulo 3, le 0 représentant l'absence, 1 et -1 représentant respectivement les valeurs vrai et faux. Les équations de ce système sont les équations sur les horloges induites par les équations sur les signaux du programme.

Les formalismes utilisés pour les preuves de programme ne représentent que la partie « contrôle » de ces programmes. Ainsi, le passage dans $Z/3Z$ pour Signal ou à l'automate pour Lustre et Esterel, fait abstraction de la valeur des signaux, et de celle des variables dans le cas d'Esterel. Les vérifications sont donc limitées à des propriétés qui ne font pas intervenir la valeur des signaux ou des variables.

II Les Langages à Objets

La programmation par objets est une technique de programmation que l'on peut utiliser dans de nombreux langages, mais au prix d'un effort qui peut être important. Un langage à objets est un langage de programmation qui facilite l'emploi de cette technique grâce à des mécanismes qui rendent son utilisation sûre et efficace [STR-87].

Nous examinons maintenant l'évolution des techniques de programmation pour faire ressortir les particularités de la programmation par objets.

La programmation procédurale met l'accent sur les algorithmes utilisés. Chaque procédure applique un algorithme bien défini à ses arguments pour produire un résultat.

Mais l'augmentation de la complexité des problèmes traités a mené à porter l'accent sur la structuration des données. On est ainsi arrivé à la notion de module, qui regroupe un jeu d'opérations avec les données qu'il manipule. Les structures de données sont cachées dans le module et ne peuvent être traitées qu'à travers les opérations qui constituent l'interface du module. L'interface de programmation est ainsi découplée de la représentation des données.

Pour reprendre un exemple classique, l'interface d'une pile se composera des fonctions `creer_pile`, `est_vide`, `empiler`, `depiler` et `detruire_pile`. Cette interface est la même que la pile soit implémentée par un tableau ou par une liste chaînée.

Un module centralise le traitement de toutes les données d'un type et peut être considéré comme un gestionnaire de type. Un problème apparaît toutefois rapidement : ces types ne se comportent pas comme les types prédéfinis du langage.

Des langages comme ADA et C++ autorisent le programmeur à définir des types qui se comportent pratiquement comme les types prédéfinis. C'est ce que l'on appelle des types abstraits, bien que ce terme ne semble pas très approprié puisqu'il s'agirait plutôt d'interprétations de types abstraits algébriques.

D'après la classification des langages faite par Wegner dans [WEG-87b], ces langages sont des langages « s'appuyant sur les objets » (object-based languages). Ils supportent la notion d'objet, c'est-à-dire d'entités regroupant un jeu d'opérations et un état qui mémorise l'effet de ces opérations.

Le fait qu'un langage supporte les objets n'est toutefois pas totalement satisfaisant dans certains cas, ce qui va nous amener à la définition des langages à objets.

En effet, une fois qu'un type d'objets est défini dans un langage qui supporte les objets sans être un langage à objets, il est impossible de lui ajouter

de nouvelles fonctionnalités sans modifier sa définition. Les opérations sur l'objet doivent prendre en compte toutes les variantes du type. Un exemple classique est la définition d'un type `figure` ayant deux variantes : `carre` et `cercle`. L'opération de tracé pour les objets de ce type doit traiter différemment le tracé d'un cercle et celui d'un carré. Si l'on veut ensuite ajouter une nouvelle variante au type, par exemple `triangle`, il faut modifier le code de la fonction de tracé. Ceci suppose que l'on ait accès à ce code, et la modification peut entraîner des erreurs dans le tracé des cercles et des carrés qui étaient pourtant corrects auparavant.

Une solution est de faire la distinction entre les propriétés générales d'un type et les propriétés spécifiques à chaque variante. Ceci mène aux notions de classe et d'héritage.

Une classe définit la structure de toute une famille d'objets appelés ses instances, ainsi que les opérations sur cette structure. Une classe peut hériter d'une autre classe, c'est-à-dire spécialiser la structure et les opérations de sa super-classe pour les adapter au comportement de ses instances.

Pour reprendre l'exemple des figures géométriques, la classe `figure` définit les propriétés générales des figures géométriques, comme avoir un centre et pouvoir être tracée. Les classes `carre` et `cercle` hé ritent de `figure` et en spécialisent la structure (avec un coté pour `carre` et un rayon pour `cercle`) et le comportement (la fonction de tracé sera différente dans `carre` et dans `cercle`).

Toujours selon Wegner, un langage à objets est un langage qui supporte les objets, dans lequel chaque objet a une classe, et où des hiérarchies de classes peuvent être définies de manière incrémentale par un mécanisme d'héritage.

Selon cette définition, Smalltalk-80 et C++ sont des langages à objets, mais ADA n'en est pas un puisqu'il ne supporte pas l'héritage.

L'héritage n'a toutefois pas que des avantages. Il est notamment très difficile d'en donner une sémantique précise, ce qui est un obstacle à la vérification formelle de programmes faisant appel à l'héritage.

II.1 Smalltalk-80

Smalltalk-80 [GOL&ROB-83] pousse l'uniformité du modèle objet jusqu'au bout puisque même les classes y sont des objets. Comme tout objet doit être instance d'une classe, chaque classe est instance de sa méta-classe qui est elle-même une classe... Cette hiérarchie se reboucle finalement sur elle-même pour que le nombre de classes reste fini et qu'il soit possible de créer des classes.

Bien que chaque objet ait une classe qui définit les opérations qui sont possibles sur lui (on parle aussi de son protocole, c'est-à-dire l'ensemble des

messages auxquels il sait réagir), il n'y a aucune vérification de type en Smalltalk. On peut donc envoyer n'importe quel message à n'importe quel objet, avec bien entendu le risque de ne pas être compris.

Le comportement associé à un message est en effet recherché lors de la réception du message par l'objet, donc à l'exécution. Smalltalk est un langage interprété, ce qui a l'avantage de le rendre très portable (il suffit de porter sa machine virtuelle), et permet de repousser facilement à l'exécution la liaison entre un message et la méthode correspondante. Partant de la classe de l'objet qui reçoit un message, l'interprète Smalltalk recherche une méthode associée à ce message dans la hiérarchie d'héritage. Il exécute la première qu'il trouve, mais s'il n'en trouve pas, on a une erreur d'exécution.

Bien qu'optimisée par l'utilisation de caches, cette technique est moins efficace qu'un appel de fonction, et qu'est donc censé faire l'utilisateur quand on lui annonce que l'objet `X` ne comprend pas le message `M?`

La souplesse de Smalltalk oblige le programmeur à être beaucoup plus attentif à la définition des classes de son application : Smalltalk ne lui indiquera pas toutes les erreurs que lui indiquerait le compilateur d'un langage fortement typé.

II.2 C++

L'approche de C++ [STR-91] est beaucoup plus pragmatique. Il s'agit d'un langage compilé et fortement typé dans lequel les classes ne sont pas des objets — ce qui nuit à l'uniformité du modèle — et sont toutes connues lors de la compilation. La recherche de la méthode à activer en réponse à la réception d'un message peut donc être faite lors de la compilation. L'activation d'une méthode se traduit par un appel de fonction, éventuellement après indexation dans une table lorsque le type exact de l'objet récepteur n'est connu qu'à l'exécution (lorsqu'une instance d'une sous-classe de `figure` reçoit le message `trace`, la méthode à activer sera déterminée par le fait que cet objet est un cercle, un carré ou un triangle lors de l'exécution).

Même lorsque la méthode à activer n'est pas connue à la compilation, on sait qu'elle existe puisqu'on connaît son index dans la table des méthodes virtuelles de l'objet récepteur.

II.2.1 Méthodes virtuelles

Les méthodes qui peuvent être redéfinies dans une sous-classe sont qualifiées de virtuelles (en Smalltalk, toutes les méthodes sont virtuelles). Celles qui doivent être définies dans une sous-classe sont qualifiées de virtuelles pures (en Smalltalk, leur corps fait appel à une méthode particulière : `SubclassResponsibility`). Une méthode virtuelle pure correspond à un comportement commun à toutes les sous-classes, mais dont le corps ne peut

être défini que pour chacune des sous-classes. C'est le cas de la méthode `trace` pour les figures : on sait que toute figure peut être tracée, mais on ignore comment tracer une figure en général.

Une méthode virtuelle correspond à un comportement commun défini pour toutes les sous-classes. Il s'agit en quelque sorte d'un comportement par défaut qui peut être redéfini par les sous-classes.

II.2.2 Protection des données

L'interface d'une classe en C++ comporte plusieurs parties correspondant à différents niveaux de protection. La partie privée contient tout ce qui n'est accessible qu'aux méthodes de la classe.

La partie protégée est accessible aux méthodes de la classe et aux méthodes des classes dérivées. Il arrive en effet que la définition d'une méthode dans une sous-classe fasse appel à des données de la super classe sans pour autant que ces données doivent être accessibles à tout le monde.

Enfin, la partie publique est accessible à toutes les fonctions. Elle contient les méthodes qui peuvent être utilisées librement sur les instances de la classe.

En plus de ces trois niveaux de protection, une classe peut autoriser une fonction à accéder à sa partie privée en la déclarant « amie ».

Ceci est particulièrement utile pour définir les opérateurs arithmétiques d'un type numérique. Ainsi, pour définir l'addition d'un entier à un complexe, on peut définir dans la classe des complexes l'opérateur + prenant un argument entier. Mais il faudrait alors rajouter dans la classe des entiers un opérateur + prenant un argument complexe, puisque l'addition est commutative. Or ceci va à l'encontre du but recherché dans les langages à objets : on ne doit pas avoir à modifier les classes existantes pour exprimer les fonctionnalités d'une nouvelle classe. On préférera donc définir un opérateur + global, prenant pour arguments deux complexes, et une méthode de conversion des entiers en complexes. L'opérateur + ainsi défini est en dehors de la classe des complexes. Pour qu'il puisse calculer la somme de deux complexes, il doit pouvoir accéder à la représentation privée d'un complexe, et donc être déclaré « ami » de la classe des complexes.

II.2.3 Initialisation et destruction d'objets

La structure d'un objet étant cachée, et les méthodes de l'objet supposant que son état vérifie certaines propriétés, il est nécessaire de fournir des méthodes d'initialisation. L'initialisation peut être explicite : le programmeur doit appeler la méthode d'initialisation avant toute autre opération sur l'objet. Mais ceci est une source d'erreurs de programmation et C++ permet au concepteur d'une classe de définir une méthode particulière, le constructeur, qui est appelée automatiquement lorsqu'un objet de la classe est créé.

Comme cette initialisation peut être complexe et monopoliser des ressources (mémoire, ressources du système), il est nécessaire de fournir un moyen de libérer ces ressources lorsque l'objet disparaît. C++ permet donc au concepteur d'une classe de définir une autre méthode particulière, le destructeur, qui est appelée automatiquement lorsqu'un objet de la classe est détruit.

Ces deux méthodes permettent d'intégrer la gestion des ressources dans la définition des classes, et d'éviter ainsi l'emploi d'un système de récupération automatique des ressources qui ne sont plus utilisées (garbage collector). On supprime ainsi un des inconvénients majeurs à l'utilisation des langages à objets dans des systèmes temps-réel.

II.2.4 Définition des opérateurs

Pour qu'une classe se comporte comme un type prédéfini du langage, il est possible de définir ses opérateurs d'affectation, de comparaison d'indexation, ainsi que ses opérateurs arithmétiques et logiques s'ils ont un sens. C++ fournit par défaut un opérateur d'affectation (affectation champ à champ de la structure) et un opérateur de test d'égalité (comparaison champ à champ de la structure). Si ces deux opérateurs n'ont pas de sens pour une classe, il est possible de les redéfinir dans l'interface privée de cette classe de façon à ce qu'ils ne puissent pas être utilisés.

II.2.5 Types paramétrés

Il est souvent utile de définir des classes de conteneurs, comme les piles, les tableaux ou les ensembles. Les opérations sur ces objets peuvent en général être exprimées sans faire référence au type des objets contenus.

Pour traiter ce problème, la version 3 de C++ permet de définir des types paramétrés, ou gabarits de classes : les templates.

La définition d'un gabarit de classes se fait en utilisant des types muets, qui sont les paramètres de la définition. Pour reprendre l'exemple de la pile, plutôt que de définir le type `pile_entiers`, on définit le gabarit `pile<X>`, `X` étant le paramètre du gabarit. On peut alors exprimer les opérations sur la pile en fonction du type `X`. Ainsi, la méthode `depiler` retournera un `X` et la méthode `empiler` prendra un `X` pour argument.

Lorsque l'on souhaite utiliser une pile d'entiers, il suffit d'identifier le type `X` au type entier en déclarant une `pile<entier>`. Le compilateur et un simulateur d'édition de liens vérifient la compatibilité des types ainsi que l'existence, dans le type entier, des méthodes invoquées pour le type `X` dans le code de la pile.

Avant l'apparition de ce mécanisme, il fallait recourir à une pile de pointeurs anonymes dans laquelle on rangeait des pointeurs sur des objets de n'importe quelle classe. Ceci pouvait avoir des conséquences catastrophiques

quand, après avoir empilé un pointeur sur une matrice de complexes, on cherchait par erreur à le dépiler dans un entier.

Les types paramétrés de C++ permettent donc de bénéficier de la généricité tout en conservant les avantages de la vérification des types.

II.3 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.

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. Ceci assure la détection à la compilation de bon nombre d'erreurs, ce qui est un facteur important puisque qu'un souci de bon fonctionnement est en général lié à l'utilisation de l'approche synchrone.

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.

III Présentation Informelle

Nous donnons ici une présentation informelle de notre système. Sa définition et ses propriétés seront données plus rigoureusement par la suite dans le modèle formel, où nous présenterons tout d'abord les propriétés statiques des horloges (celles qui sont vraies à chaque instant). Nous introduirons ensuite la notion de classe synchrone qui permet de définir l'instanciation d'un objet dans une horloge. Ceci nous amènera à définir les propriétés dynamiques du système, c'est-à-dire celle qui régissent le passage d'un instant d'une horloge au suivant.

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.

III.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, et c'est en cela qu'elle nous intéresse puisque notre travail commence lorsque celui du compilateur du langage synchrone se termine.

Notre matière première est donc la forme compilée d'un module synchrone. Pour des raisons matérielles, nous avons rapidement disposé d'un compilateur Esterel (il se trouve que le CMA utilise comme nous des machines DIGITAL, ce qui a supprimé tout problème de portage), cette forme compilée sera donc le code oc [COU-90] de l'automate correspondant au module. Mais rien n'interdit d'utiliser une autre forme compilée, par exemple du code monoboucle, qui est la forme compilée produite par le compilateur Signal ainsi que par la version 4 du compilateur Esterel, 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.

Il va de soi qu'une fois le module compilé, les détails de son fonctionnement ne sont plus visibles : le module se présente sous la forme d'une boîte noire avec un certain nombre de signaux d'entrée et de sortie, ainsi qu'un bouton qui provoque la réaction du module à ses signaux d'entrée.

Dans un langage synchrone, le module apparaît sous la forme de son code source. Les détails de sa conception sont donc visibles et peuvent être exploités par le compilateur. Ainsi, quand on construit un module complexe à par-

tir de modules plus simples, le compilateur du langage synchrone détermine son comportement en explorant toutes les possibilités d'interactions entre les comportements des sous-modules. Ceci permet de vérifier statiquement que la composition des modules a un sens (absence de boucles de causalité, compatibilité des types des signaux) mais n'est possible que parce que le système ainsi construit est statique : tous les modules et leurs interconnexions sont connus à la compilation.

Cette possibilité de vérifier des propriétés d'un système a priori et de manière formelle (les vérifications se font par transformation du code source) est un des principaux intérêts des langages synchrones [HAL-92]. Les modules que nous intégrons dans un langage à objets étant produits par un compilateur de langage synchrone, ils bénéficient individuellement de ces techniques de vérification.

III.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 (ensemble des messages auxquels il répond) 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. Dans le cas d'Esterel et du code oc, la seule façon de modifier l'état de l'automate, une fois encapsulé dans un objet, est de lui fournir des signaux d'entrée et de le faire réagir. Il n'y a pas de variable accessible codant cet état.

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, sauf en cas de faute dans la chaîne de développement ou dans le matériel. La conformité de ce comportement au code source du module dépend du compilateur du langage synchrone et des mécanismes qui ont permis de faire apparaître le module sous forme d'objet. Mais ces outils ont été utilisés suffisamment souvent pour qu'on puisse considérer qu'ils ne comportent pas d'erreur non détectée.

III.3 Classes synchrones

Il faut maintenant indiquer comment on passe de la forme compilée d'un module synchrone à un objet. Rappelons tout d'abord que dans un langage à objets, le comportement et la structure d'un objet sont définis par une

classe. Cette définition comporte trois aspects : le protocole, les méthodes et le schéma d'instance. Le schéma d'instance donne la structure de l'état des objets. Concrètement, il s'agit d'un jeu de variables dont la valeur définit l'état de l'objet.

Les méthodes sont les procédures et fonctions qui implémentent les services rendus par l'objet. Elles peuvent modifier l'état de l'objet et sont activées lorsqu'il reçoit un message. Le protocole est l'ensemble des messages reconnus par l'objet, c'est-à-dire des messages auxquels il sait associer une méthode.

La structure et le comportement des objets synchrones sont donc décrits par des classes que nous appelons classes synchrones pour les distinguer des autres classes.

Nous transformons la forme compilée d'un module synchrone en une classe synchrone. Le mécanisme qui permet d'obtenir un objet synchrone à partir de cette classe est l'instanciation.

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. De plus, l'instanciation est un mécanisme dynamique, c'est-à-dire que l'on peut instancier une classe au cours de l'exécution d'un programme. Il est donc possible de créer dynamiquement des objets synchrones.

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.

III.4 Héritage entre classes synchrones

Comme les autres classes, les classes synchrones peuvent être liées par des relations d'héritage. Rappelons que si une classe B hérite (ou dérive) d'une classe A, le protocole de B inclus celui de A, et le schéma d'instance de B inclus celui de A. On remarque que l'héritage n'impose pas de contrainte sur les méthodes. Le même message envoyé à une instance de A et à une instance de B peut activer des méthodes dont le comportement est fort différent. Cette absence de contrainte est un obstacle à la définition d'une sémantique de l'héritage.

Toutefois, en tant qu'utilisateurs de ce mécanisme, nous pouvons nous imposer des règles de bonne conduite. 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) [AME-90].

Donnons un exemple simple pour illustrer notre conception de l'héritage. On souhaite manipuler des figures géométriques, et notamment les afficher. Chaque figure a un comportement d'affichage différent : un cercle ne se dessine pas comme un carré.

Pourtant, toutes les figures 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` qui contient le centre de la figure. On fait ensuite hériter les classes `Carré` et `Cercle` de la classe `Figure`.

`Carré` ajoute au schéma d'instance de `Figure` une variable `côté` qui contient la longueur du côté du carré, et `Cercle` y ajoute une variable `rayon` qui contient la longueur du rayon du cercle. 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`.

Comme règle de bonne conduite, nous nous sommes imposé de conserver la sémantique intuitive du message `affiche`. Cette sémantique est difficilement formalisable puisqu'elle correspond à ce que l'utilisateur attend de la méthode d'après le nom du message et la classe de l'objet auquel il l'envoie. Pourtant, ce mécanisme est utile puisqu'il permet de manipuler des figures géométriques et de leur demander de s'afficher sans se soucier de leur nature.

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.

Notre conception de l'héritage étant exposée, 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 MB dont le comportement est un raffinement de celui du module MA. 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 MA, nous en construisons un autre MB ayant au moins les mêmes signaux d'entrée et de sortie, et dont le comportement est obtenu en filtrant les signaux de MA grâce à des modules synchrones auxiliaires.

La classe synchrone B correspondant à MB a donc au moins les mêmes signaux que sa super-classe A qui correspond à MA. Elle assure au minimum les mêmes services que A, et peut avoir des signaux supplémentaires servant à affiner le comportement de base.

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.

III.5 Signaux et Connexions

Nous savons transformer la forme compilée d'un module synchrone en une classe synchrone, et nous savons aussi instancier cette classe pour obtenir des objets synchrones dont le comportement est celui du module synchrone de départ.

Afin que tout ceci ne soit pas inutile, il faut que ces objets synchrones 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 certain type 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 dans le langage à objets. Ces objets font partie du schéma d'instance des objets synchrones. Il n'existe pas de signaux « libres » : un signal est toujours signal d'entrée ou de sortie d'un objet synchrone, éventuellement d'un objet d'interface lorsqu'il s'agit d'un signal qui permet de communiquer avec le monde asynchrone comme nous le verrons plus loin.

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.

Le langage à objets utilisé — il s'agit de C++ — est typé. Nous utilisons les mécanismes de vérification de types du compilateur 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.

Un mécanisme de la version 3 du C++ d'ATT [UNI-91] 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 : il s'agit des templates ou « gabarits de type ». Ce mécanisme sera exposé plus en détail

lorsque nous présenterons le protocole des classes synchrones et des signaux. Notons simplement pour l'instant qu'il permet de vérifier statiquement la validité d'une connexion pour ce qui est du transport de données typées.

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. Les méthodes de connexion des signaux font cette vérification avant d'établir la connexion.

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. Nous envisageons deux façons de traiter les erreurs de connexion : soit par l'émission d'un signal particulier, l'erreur étant alors traitée par un objet synchrone, soit par la levée d'une exception, l'erreur étant traitée par du code non synchrone.

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 chapitre « Outils de Développement ».

III.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, ce qui permet de retrouver le retard de Signal.

Une propriété intéressante du retard est qu'il permet de « casser » les boucles de causalité [GON-88, LGU&GAU-90]. 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].

III.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.

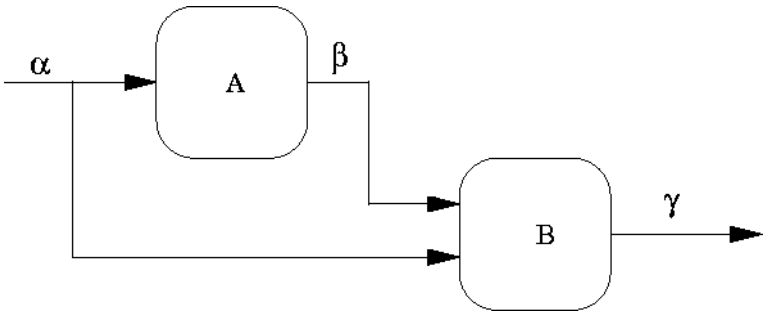

FIG. III.1 – Communication synchrone entre objets

La figure III.1 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.

III.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 évoluer d'un instant à l'autre d'une horloge, 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. Par définition, la topologie de l'horloge ne peut en effet pas changer pendant un instant. L'instant est une unité atomique dans laquelle on ne peut pas considérer d'événements comme l'apparition d'un objet : l'objet est présent ou pas à un instant, il ne peut pas apparaître « pendant » l'instant.

La détermination de l'ordre dans lequel doivent réagir les objets synchrones d'un instant est confiée à un ordonnanceur. L'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.

La dynamicité du réseau d'interconnexion entre les objets synchrones d'une horloge, et la possibilité qu'un nouvel objet soit créé en réaction à l'occurrence d'un événement correspondent à des comportements d'acteurs [HEW&BAK-77], les connexions correspondant à la relation d'« acquaintance » entre acteurs. On retrouve la même similitude pour l'ordre de réaction des objets, qui correspond à l'ordre partiel sur les événements dans le modèle des acteurs.

III.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 de construire des événements pour l'horloge à partir d'événements asynchrones.

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.

IV Modèle temporel

Nous présentons ici un modèle temporel de notre système. Ce modèle définit les objets synchrones, les horloges et les règles permettant de calculer la valeur des signaux.

Nous traitons tout d'abord le cas des objets synchrones réagissant sur la même horloge et qui communiquent donc de façon synchrone par des signaux. Nous traitons ensuite le cas des communications asynchrones, c'est-à-dire entre deux horloges distinctes ou entre une horloge et un processus asynchrone. Nous insistons sur le cas du retard associé à une horloge qui est présenté comme le bouclage d'un signal sur une horloge par un processus asynchrone suffisamment rapide.

Ce modèle donne les propriétés des instants d'une horloge. Les propriétés des objets synchrones en relation avec leur classe seront exposées au chapitre « Modèle Objet ». Nous traiterons enfin les aspects temporels liés à la création dynamique d'objets synchrones dans le chapitre « Relations entre le Modèle Temporel et le Modèle Objet ».

IV.1 Notations

Certaines entités du modèle sont des agrégats dont les champs portent un nom. L'accès aux champs d'un agrégat se fait par l'opérateur infixé '.' : `A.b` désigne le champ `b` de l'entité `A`. Le nom d'un champ est un nom local : il n'est unique que dans l'agrégat qui le contient.

Pour désigner de manière unique chaque entité, nous préfixons le nom d'une entité possédée par celui de l'entité possédante suivi du caractère •. Ce caractère ne doit donc pas faire partie de l'alphabet des noms simples. On aura donc : nom(A.b) = nom(A)•nom(b), nom(A)•nom(b) étant la concaténation du nom de l'entité A, du caractère '•' et du nom de l'entité b. On notera a•E l'ensemble des identificateurs construits en préfixant les éléments de E par le mot 'a' suivi du caractère '•'.

On note $\mathscr{B}$ l'ensemble {vrai, faux} des booléens, **N** l'ensemble des entiers naturels, F^E l'ensemble des fonctions de E dans F et $\mathscr{P}_f(E)$ l'ensemble des parties finies de E.

IV.2 Communication synchrone

Soit $\mathscr{A}$ un alphabet ne contenant pas le caractère '•', et $\mathscr{N} = \mathscr{A}^+$ l'ensemble des mots composés d'un nombre fini non-nul de caractères de $\mathscr{A}$. $\mathscr{N}$ est l'ensemble des *noms simples*, aussi appelés *noms locaux*.

Les valeurs échangées lors des communications sont typées. Nous avons donc un ensemble de noms de types auxquels sont associés des domaines de valeur. Une valeur ne peut appartenir qu'à un seul domaine (elle n'a qu'un type).

Soit $\mathscr{T}$ une partie finie de $\mathscr{N}$: $\mathscr{T} \in \mathscr{P}_f(\mathscr{N})$.

$\mathscr{T}$ est l'ensemble des types. A chaque élément t de $\mathscr{T}$, on associe un domaine de valeurs $\mathscr{D}_t$ de telle sorte que les $\mathscr{D}_t$ soient deux à deux disjoints. $\mathscr{D}_t$ est le domaine du type t.

Soit $\mathscr{V} = \bigcup_{t\in\mathscr{T}} \mathscr{D}_t$ l'ensemble des valeurs possibles, et *typ* la fonction qui associe à chaque valeur v l'élément t de $\mathscr{T}$ tel que $v \in \mathscr{D}_t$:

$$\begin{array}{rccl} typ: & \mathscr{V} & \rightarrow & T \\ & v & \mapsto & t / v \in \mathscr{D}_t \end{array}$$

$t = typ(v)$ est le *type de v*. On définit la relation d'équivalence τ qui relie les valeurs de même type par :

$$\forall v, w \in \mathscr{V}, v \tau w \Leftrightarrow typ(v) = typ(w)$$

Définition IV.1 *On appelle* objet synchrone *un quintuplé*

$$\mathbf{M} = \langle \nu, s_I, s_O, \Sigma, \mathscr{R} \rangle$$

tel que :

- $\nu \in \mathscr{N}$ *est le nom de l'objet*
- s_I *et* $s_O \in \nu \bullet \mathscr{P}_f(\mathscr{N})$
  s_I *est l'ensemble des signaux d'entrée de* **M**
  s_O *est l'ensemble des signaux de sortie de* **M**
- $\Sigma = [0, k], k \in \mathbf{N}$ *est l'ensemble des états de* **M**
- $\mathscr{R} : (\mathscr{V} \times \mathbf{B})^{s_I} \times \Sigma \rightarrow (\mathscr{V} \times \mathscr{B})^{s_O} \times \Sigma$
  $\mathscr{R}$ *est la fonction de réaction de* **M**

On note $\mathscr{M}$ *l'ensemble des objets synchrones.*

La fonction de réaction $\mathscr{R}$ d'un objet calcule le nouvel état de l'objet et la fonction d'évaluation de ses signaux de sortie en fonction de l'état courant de l'objet et de la fonction d'évaluation de ses signaux d'entrée.

Les fonctions d'évaluation ont $\mathscr{V} \times \mathscr{B}$ pour ensemble d'arrivée car un signal a une valeur dans $\mathscr{V}$ et peut être émis ou pas au moment où on l'évalue. La composante booléenne vaut vrai si le signal est émis, faux sinon. Quand un signal n'est pas émis, sa valeur est celle qu'il avait la dernière fois qu'il a été émis.

Nous allons maintenant définir la notion d'horloge comme une suite d'ensembles d'objets synchrones. Chaque élément de cette suite peut être vu

comme un instantané de l'horloge. A chacun de ces instants, un état est associé à chaque objet et une valeur est associée aux signaux. Cette valeur est la même pour tous les objets.

L'instant suivant est déterminé par la réaction des objets à la valeur des signaux. On désignera le n^{e}terme d'une horloge **H** par : « **H** à l'instant n ».

Certains signaux ne font partie d'aucun objet : ce sont les entrées et les sorties de l'horloge. Il est important de ne pas confondre « entrée » (respectivement « sortie ») et « signal d'entrée » (respectivement « signal de sortie »).

Un signal d'entrée est un signal qui peut être connecté à un signal de sortie dont il tire sa valeur. Un signal de sortie est un signal qui est susceptible de fournir sa valeur à des signaux d'entrée. Ceci ne présage pas de ce que représente un tel signal pour celui qui l'utilise.

Ainsi, les signaux d'entrée d'un objet synchrone sont effectivement des entrées pour lui. Ils lui fournissent les valeurs auxquelles il réagit. Dans le cas d'une horloge, les entrées sont les valeurs disponibles pour les objets de l'horloge. Elles apparaissent donc comme des signaux de sortie, afin que l'on puisse y connecter les signaux d'entrée des objets. De même, les sorties de l'horloge sont des valeurs produites par les objets. Elles apparaissent donc comme des signaux d'entrée, qui peuvent être connectés aux signaux de sortie des objets de l'horloge.

On peut aussi considérer que les entrées de l'horloge sont perçues par les objets comme des sorties du monde extérieur, de même que les sorties de l'horloge sont utilisées comme entrées par le monde extérieur.

Définition IV.2 *On appelle* horloge *une suite* **H** *de triplets*

$$\mathbf{H} = (\mathbf{H}_n)_{n\in\mathbf{N}} = (\langle s_I, s_O, \mathbf{T}\rangle_n)_{n\in\mathbf{N}}$$

tels que :

(i) $s_I \text{et} s_O \in \bullet\mathscr{P}_f(\mathscr{N})$

(ii) $\mathbf{T} \in \mathscr{P}_f(\mathscr{M})$

(iii) $\forall i, j \in \mathbf{N}, j > i$
$\mathbf{M} \in \mathbf{H}_i.\mathbf{T}, \mathbf{M} \in \mathbf{H}_j.T \Rightarrow \forall t \in [i, j], \mathbf{M} \in \mathbf{H}_t.\mathbf{T}$

Le point i) permet de nommer les signaux à un instant de l'horloge de façon distincte des signaux des objets car $\mathscr{N}$ ne contient pas le mot vide. Il faut noter que s_I et s_O ne sont pas forcément constants et peuvent changer d'un instant à l'autre de l'horloge.

Le point ii) indique que chaque terme de l'horloge contient un nombre fini d'objets.

Le point iii) exprime le fait qu'un objet ne peut pas disparaître de l'horloge à un instant et y reparaître plus tard. La vie d'un objet dans une horloge est donc un intervalle $[i, j]$.

Nous allons maintenant exprimer le fait qu'à chaque instant d'une horloge, les objets ont un état, et que leurs signaux et ceux de l'horloge prennent une valeur, et peuvent être interconnectés afin que ces valeurs soient partagées par plusieurs signaux.

Définition IV.3 *On appelle évolution d'une horloge* $\mathbf{H}$ *une suite* $\mathscr{S}_{\mathbf{H}}$ *de fonctions d'état vérifiant :*

$$\mathscr{S}_{\mathbf{H}} = (\mathscr{S}_{H_n})_{n\in\mathbf{N}} / \quad \begin{array}{l} \forall n \in \mathbf{N}, \mathscr{S}_{h_n} : \mathbf{H}_n.\mathbf{T} \rightarrow \mathbf{N} \\ \forall \mathbf{M} \in \mathbf{H}_n.\mathbf{T}, \mathscr{S}_{\mathbf{H}_n}(\mathbf{M}) \in \mathbf{M}.\Sigma \end{array}$$

La fonction d'état de l'instant n de l'horloge $\mathbf{H}$ associe son état à chaque objet de l'horloge. Cet état appartient à l'ensemble des états de l'objet.

Soient $s_I\mathbf{H}_n = \mathbf{H}_n.s_I \cup \bigcup_{\mathbf{M}_i\in\mathbf{H}_n.\mathbf{T}} \mathbf{M}_i.s_I$ l'ensemble de tous les signaux d'entrée présents dans $\mathbf{H}_n$, $s_O\mathbf{H}_n = \mathbf{H}_n.s_O \cup \bigcup_{\mathbf{M}_i\in\mathbf{H}_n.\mathbf{T}} \mathbf{M}_i.s_O$ l'ensemble de tous les signaux de sortie présents dans $\mathbf{H}_n$, et $s\mathbf{H}_n = s_I\mathbf{H}_n \cup s_O\mathbf{H}_n$ l'ensemble de tous les signaux présents dans $\mathbf{H}_n$.

Pour une horloge $\mathbf{H}$, on définit une suite de fonctions de connexion $\mathscr{C}_{\mathbf{H}}$ et une suite de fonctions d'évaluation $\mathscr{E}_{\mathbf{H}}$. La fonction de connexion indique à quels signaux de sortie sont connectés les signaux d'entrée présents dans l'horloge :

Définition IV.4 *On appelle connexion d'une horloge* $\mathbf{H}_n$ *une suite* $\mathscr{C}_{\mathbf{H}}$ *de fonctions vérifiant :*

$$\mathscr{C}_{\mathbf{H}} = (\mathscr{C}_{\mathbf{H}_n})_{n\in\mathbf{N}} / \forall n \in \mathbf{N}, \mathscr{C}_{\mathbf{H}_n} : s_I\mathbf{H}_n \rightarrow s_O\mathbf{H}_n$$

D'après le profil des $\mathscr{C}_{\mathbf{H}_n}$, on voit qu'un signal d'entrée ne peut être connecté qu'à un signal de sortie. Les $\mathscr{C}_{\mathbf{H}_n}$ ne sont pas nécessairement surjectives : plusieurs signaux d'entrée peuvent être connectés à un même signal de sortie. Elles ne sont pas nécessairement injectives : un signal de sortie peut n'être la source d'aucun signal d'entrée.

La fonction d'évaluation d'une horloge à un instant associe à chaque signal sa valeur et un booléen indiquant la présence du signal. L'évaluation d'une horloge est la suite des fonctions d'évaluation de cette horloge à chacun de ses instants :

Définition IV.5 *On appelle évaluation d'une horloge* $\mathbf{H}$ *la suite de fonctions définie par :*

$$\mathscr{E}_{\mathbf{H}} = (\mathscr{E}_{\mathbf{H}_n})_{n\in\mathbf{N}} / \forall n \in \mathbf{N}, \mathscr{E}_{\mathbf{H}_n} : s\mathbf{H}_n \rightarrow \mathscr{V} \times \mathscr{B}$$

Sur la figure IV.1, on voit une horloge $\mathbf{H}$ à un instant n où elle contient deux objets O_1 et O_2 et a un signal de sortie 'a' et un signal d'entrée 'b'. Les signaux d'entrée $O_2 \bullet e$ et $O_1 \bullet g$ sont connectés au signal de sortie $\bullet a$, et le signal d'entrée $\bullet b$ est connecté au signal de sortie $O_2 \bullet n$. Les flèches sur les traits de connexion indiquent le sens de circulation de l'information.

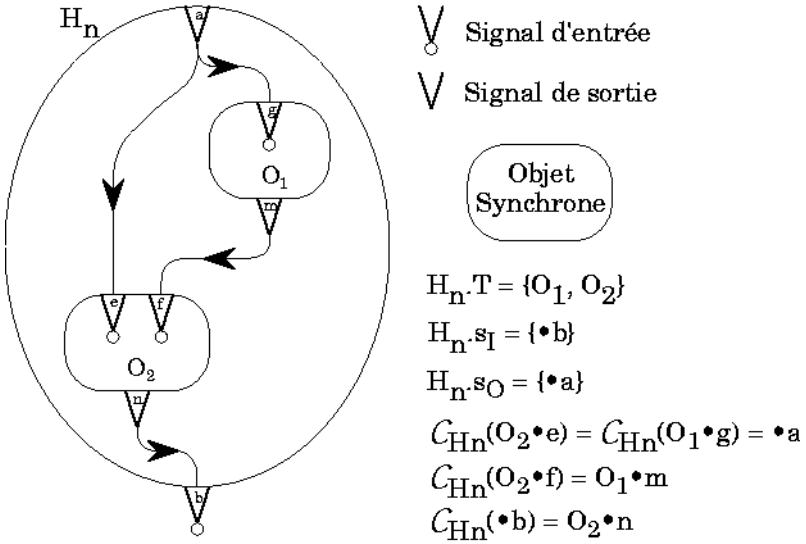

FIG. IV.1 – Horloge H à l'instant n

L'évaluation de l'horloge n'est pas représentée sur cette figure.

Les objets synchrones d'un système ne sont pas forcément tous sur la même horloge. Un système a donc un ensemble d'horloges qui doivent vérifier certaines propriétés pour que le système soit valide.

Définition IV.6 *On considère dans la suite un ensemble $\mathscr{H}$ d'horloges vérifiant :*

$$\forall \mathbf{H}, \mathbf{H}' \in \mathscr{H}, \mathbf{H} \neq \mathbf{H}' \Rightarrow \forall i, j \in \mathbf{N}, \mathbf{H}_i \cap \mathbf{H}'_j = \emptyset$$

Cette condition exprime le fait qu'un objet synchrone ne peut appartenir qu'à une seule horloge. On pourrait penser qu'un ensemble d'horloges convenables peut être obtenu en interdisant simplement à un objet d'appartenir au même instant à deux horloges. L'objet commencerait ainsi son activité dans une horloge, puis arriverait avec toute son histoire dans une autre horloge pour y poursuivre son activité.

Malheureusement, il n'est pas possible d'exprimer qu'un objet n'appartient pas à deux horloges au même instant. Les indices de deux horloges ne sont en effet aucunement liés, l'instant i d'une horloge pouvant correspondre à l'instant j ou k d'une autre horloge selon le temps de référence choisi pour la comparaison. La seule façon de garantir qu'un objet n'est jamais partagé par deux horloges est donc d'interdire qu'il puisse apparaître dans les deux.

Nous introduisons maintenant la notion de type d'un signal, les valeurs prises par un signal étant toutes de même type.

Propriété IV.1 *On note $p_{\mathscr{V}}$ la projection sur $\mathscr{V}$, et $p_{\mathscr{B}}$ la projection sur $\mathscr{B}$ d'un élément de $\mathscr{V} \times \mathscr{B}$. On peut étendre la relation τ à $\mathscr{V} \times \mathscr{B}$:*

$$\forall v, w \in \mathscr{V} \times \mathscr{B}, v \tau w \Leftrightarrow p_{\mathscr{V}}(v) \tau p_{\mathscr{V}}(w)$$

La valeur d'un signal est toujours du même type :

$$\forall \mathbf{H} \in \mathscr{H}, \forall n, n' \in \mathbf{N}, \forall \mathbf{M} \in \mathbf{H}_n.\mathbf{T} \cap \mathbf{H}'_n.\mathbf{T}, \forall s \in \mathbf{M}.s_I \cup \mathbf{M}.s_O, \mathscr{E}_{\mathbf{H}_n}(s) \tau \mathscr{E}_{\mathbf{H}_{n'}}(s)$$

Deux signaux connectés ont même type et même valeur, ce qui s'exprime par la propriété suivante sur $\mathscr{E}_{\mathbf{H}}$ et $\mathscr{C}_{\mathbf{H}}$:

Propriété IV.2 *Deux signaux connectés ont même type et même valeur :*

$$\begin{array}{l} \forall \mathbf{H} \in \mathscr{H}, \forall n \in \mathbf{N}, \forall s \in s_I\mathbf{H}_n, \forall s' \in s_O\mathbf{H}_n \\ \mathscr{C}_{\mathbf{H}_n}(s) = s' \Rightarrow \left\{ \begin{array}{lcl} \mathscr{E}_{\mathbf{H}_n}(s) & \tau & \mathscr{E}_{\mathbf{H}_n}(s') \\ \mathscr{E}_{\mathbf{H}_n}(s) & = & \mathscr{E}_{\mathbf{H}_n}(s') \end{array} \right. \end{array}$$

D'après la définition de la fonction de réaction d'un objet, on voit que la fonction d'évaluation doit être connue pour les signaux d'entrée afin de pouvoir calculer la fonction d'évaluation des signaux de sortie. Pour calculer la fonction d'évaluation totale, il va falloir calculer la réaction des objets dans un certain ordre. Notre but est maintenant de montrer qu'un tel ordre existe et qu'il est donc possible de calculer la fonction d'évaluation $\mathscr{E}_{\mathbf{H}_n}$.

On définit une relation de précédence $\prec$ entre les objets synchrones d'une horloge à un instant donné :

Définition IV.7 *On appelle précédence entre objets synchrones la relation $\prec$ définie par :*

$$\begin{array}{l} \forall \mathbf{H} \in \mathscr{H}, \forall n \in \mathbf{N}, \forall \mathbf{M}, \mathbf{M}' \in \mathbf{H}_n.\mathbf{T} \\ \mathbf{M} \prec \mathbf{M}' \Leftrightarrow \exists s \in \mathbf{M}'.s_I / \mathscr{C}_{\mathbf{H}_n}(s) \in \mathbf{M}.s_O \end{array}$$

$\mathbf{M} \prec \mathbf{M}'$ signifie que $\mathbf{M}'$ dépend directement de $\mathbf{M}$ puisqu'au moins un de ses signaux d'entrée est connecté à un signal de sortie de $\mathbf{M}$.

Soit $\dot{\prec}$ la fermeture transitive de $\prec$. $\dot{\prec}$ est transitive par construction, et on a de plus la propriété suivante :

Propriété IV.3 *La fermeture transitive $\dot{\prec}$ de $\prec$ est un ordre strict partiel sur $\mathbf{H}_n.\mathbf{T}$*

L'antisymétrie de $\dot{\prec}$ interdit toute boucle de plus d'un objet dans le graphe orienté $\mathscr{G}_{\dot{\prec}}$ de $\dot{\prec}$. Le fait que $\dot{\prec}$ soit un ordre strict interdit toute boucle d'un objet vers lui-même dans ce même graphe. On a donc :

Corollaire IV.1 *$\mathscr{G}_{\dot{\prec}}$ est un graphe orienté acyclique.*

La relation de précédence ne permet pas de prendre en compte le fait qu'un signal de sortie d'un objet ne dépend pas d'un de ses signaux d'entrée. Si le retard était un objet synchrone dans notre modèle, on ne pourrait pas l'utiliser pour faire circuler de l'information le long d'une boucle. Cette boucle apparaîtrait en effet dans $\mathscr{G}_{\dot{\prec}}$. Nous définissons donc le retard comme une paire de signaux de l'horloge :

Définition IV.8 *On appelle retard d'une horloge* **H** *un couple de signaux* $\langle s, s_z \rangle$ *tel que :*

(i) $s \in \mathbf{H}_n.s_I, s_z \in \mathbf{H}_{n+1}.s_O$

(ii) $\mathscr{E}_{\mathbf{H}_{n+1}}(s_z) = \mathscr{E}_{\mathbf{H}_n}(s)$

Le point i) exprime le fait que les signaux du retard sont des signaux de l'horloge aux instants où ils sont émis.

Le point ii) exprime le fait que le signal retardé prend la valeur qu'avait le signal d'origine à l'instant précédent de l'horloge.

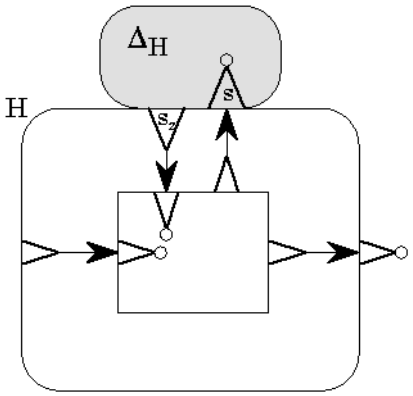

FIG. IV.2 – Retard d'une horloge

Pour connaître l'état du système, il faut pouvoir évaluer les horloges, c'est-à-dire calculer les suites $\mathscr{E}_{\mathbf{H}}$. Pour une horloge **H** à un instant n, on connaît $\mathscr{E}_{\mathbf{H}_n}/\mathbf{H}_n.s_O$ qui est une donnée du monde extérieur. Pour chaque objet synchrone **M** de $\mathbf{H}_n.\mathbf{T}$, on connaît la fonction de réaction $\mathscr{R}$ qui permet d'obtenir $\mathscr{E}_{\mathbf{H}_n}/\mathbf{M}.s_O$ à partir de $\mathscr{E}_{\mathbf{H}_n}/\mathbf{M}.s_I$, cette dernière se déduisant d'une fonction partielle d'évaluation de signaux de sortie grâce à $\mathscr{C}_{\mathbf{H}_n}$.

On va donc calculer $\mathscr{E}_{\mathbf{H}_n}$ à partir de ses fonctions partielles sur les signaux de l'horloge et des objets synchrones. Nous allons pour cela construire une suite d'ensembles d'objets synchrones telle que les fonctions partielles sur les signaux des objets d'un ensemble ne dépendent que des fonctions partielles sur les signaux des objets du précédant. Nous montrerons ensuite que cette suite permet de calculer $\mathscr{E}_{\mathbf{H}_n}$ en un nombre fini d'étapes.

Les $\mathbf{H}_n.\mathbf{T}$ étant finis, l'ensemble des éléments minimaux pour $\dot{\prec}$ de $\mathbf{H}_n.\mathbf{T}$ est non-vide (sauf dans le cas trivial ou $\mathbf{H}_n.\mathbf{T}$ est vide). On peut donc définir la suite d'ensembles σ_n , σ_{n+1} étant l'ensemble des objets synchrones qui peuvent réagir une fois que ceux de σ_n ont produit leurs signaux de sortie :

Définition IV.9 *Pour une horloge* **H** *à l'instant n, on définit la suite* $\sigma = (\sigma_k)_{k \in \mathbf{N}}$ *par :*

$$\begin{aligned} \kappa_0 &= \mathbf{H}_n.\mathbf{T} \\ \sigma_0 &= \{\mathbf{M} \in \kappa_0 / \{\mathbf{M}' \in \kappa_0 / \mathbf{M}' \dot{\prec} \mathbf{M}\} = \emptyset\} = \min \kappa_0 \\ \kappa_{k+1} &= \kappa_k \setminus \sigma_k \\ \sigma_{k+1} &= \min \kappa_{k+1} \end{aligned}$$

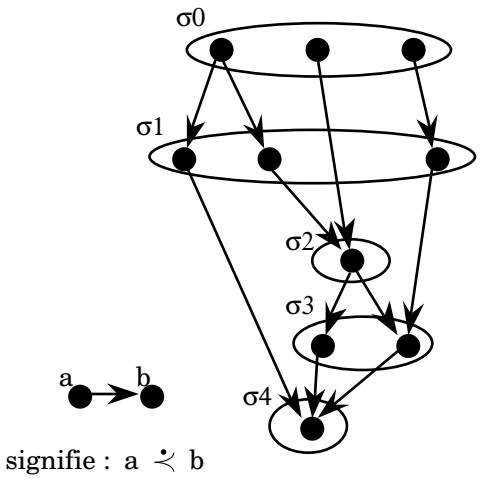

FIG. IV.3 – Les ensembles σ_k

Théorème 1 *Le calcul de* $\mathscr{E}_{\mathbf{H}_n}$ *se réduit au calcul des* $\mathscr{E}_{\mathbf{H}_n}/\sigma_k$ *et de* $\mathscr{E}_{\mathbf{H}_n}/\mathbf{H}_n.s_I$. *Ce calcul se fait en au plus* Card($\mathbf{H}_n$.**T**) *étapes.*

Preuve κ_k est l'ensemble des objets dont on n'a pas encore calculé la réaction après k étapes. Si κ_k est non vide, comme il est fini, l'ensemble $\sigma_k = \min \kappa_k$ des objets dont on peut calculer la réaction après k étapes est non vide, donc de cardinal au moins égal à 1. Si κ_k est vide, le calcul est terminé.

A chaque étape, on peut donc calculer la réaction d'au moins un objet, il faut alors au maximum Card($\mathbf{H}_n$.**T**) étapes pour calculer celle de tous les objets de $\mathbf{H}_n$.**T**.

Partant de $\mathscr{E}_{\mathbf{H}_n}/\mathbf{H}_n.s_O$, on obtient $\mathscr{E}_{\mathbf{H}_n}/\sigma_0.s_I$ grâce à $\mathscr{C}_{\mathbf{H}_n}$. Les fonctions de réaction des objets de σ_0 fournissent $\mathscr{E}_{\mathbf{H}_n}/\sigma_0.s_O$. En appliquant à chaque fois la fonction de connexion pour obtenir la fonction partielle sur les signaux d'entrée de σ_k, puis les fonctions de réaction pour obtenir la fonction partielle sur les sortie de σ_k, on obtient de proche en proche en au plus Card($\mathbf{H}_n$.**T**) toutes les fonctions partielles sur les σ_k. On connaît donc $\mathscr{E}_{\mathbf{H}_n}$ pour tous les signaux des objets de l'horloge et pour les signaux de sortie de l'horloge. Il reste à calculer $\mathscr{E}_{\mathbf{H}_n}/\mathbf{H}_n.s_I$ qui est en fait fournie par $\mathscr{C}_{\mathbf{H}_n}$.

IV.3 Communication asynchrone

Nous allons maintenant spécifier la manière dont une horloge communique avec une autre horloge ou avec le monde asynchrone. Pour cela, nous avons besoin d'une référence temporelle globale permettant de dater les événements. Cette référence temporelle peut aussi bien être le temps physique compté à partir d'une date de référence quelconque, qu'un temps logique progressant au fil des interactions entre les entités du système, similaire aux horloges de Lamport [LAM-78].

On exprime le rapport de cause à effet entre deux événements par la relation *est une conséquence de* qui est notée $\lhd$. Cette relation est un ordre partiel sur les événements.

La référence temporelle globale se traduit par une fonction de datation qui associe un entier à chaque événement. L'ordre induit sur les événements par la datation doit être compatible avec l'ordre induit par les rapports de cause à effet :

Définition IV.10 *On appelle datation une fonction δ qui à tout événement e associe un entier et qui vérifie :*

$$e' \lhd e \Leftrightarrow \delta(e') \geq \delta(e)$$

Pour des événements entre lesquels n'existe aucun lien causal, deux datations différentes peuvent donner des ordres différents. La causalité n'est en effet qu'une relation d'ordre partiel sur les événements.

Un instant d'une horloge est un événement. On peut donc parler de la date de l'instant n de l'horloge **H**, que l'on note $\delta(\mathbf{H}_n)$.

Cet instant est la conséquence de l'instant précédent puisque l'état des objets qui s'y trouve dépend de leur état antérieur. Il est aussi la conséquence d'événements extérieurs à l'horloge qui se manifestent à travers les entrées de cette horloge (qui sont en fait des signaux de sortie comme nous l'avons vu).

Cet instant peut aussi avoir des conséquences à l'extérieur de l'horloge par le biais de ses sorties (qui sont des signaux d'entrée). La communication entre une horloge **H** et les entités extérieures à **H** vérifie :

Propriété IV.4 *Pour toute fonction de datation δ, pour toute horloge* **H** *et pour tout événement e extérieur à l'instant n de* **H**, *on a :*

(i) $e \lhd \mathbf{H}_n \Rightarrow \delta(e) > \delta(\mathbf{H}_n)$

(ii) $\mathbf{H}_n \lhd e \Rightarrow \delta(e) \leq \delta(\mathbf{H}_n)$

Le point i) indique que les conséquences d'un instant d'une horloge sont strictement postérieures à cet instant.

Le point ii) indique qu'un instant de l'horloge peut dépendre d'un événement antérieur ou simultané à cet instant.

Si un signal est bouclé sur une horloge **H**, son occurrence à l'instant $\mathbf{H}_n$ ne peut donc être perçue en entrée qu'à l'instant $\mathbf{H}_{n+1}$ au plus tôt.

Les langages synchrones s'appuient sur l'hypothèse synchrone. Cette hypothèse suppose que la réaction d'un système synchrone à un événement est instantanée. Dans la pratique, il suffit que le système réagisse en un temps inférieur à celui qui sépare deux événements.

En s'appuyant sur les propriétés temporelles des communications asynchrones, on peut donc exprimer l'hypothèse synchrone comme suit :

Définition IV.11 *On dit que l'hypothèse synchrone est* vérifiée strictement *pour une horloge* **H** *vis à vis d'une suite d'événements* $(e_n)_{n\in\mathbf{N}}$ *si pour toute fonction de datation δ on a :*

$$\forall m, n \in \mathbf{N}, \delta(\mathbf{H}_n) < \delta(e_m) \leq \delta(\mathbf{H}_{n+1}) \Rightarrow \delta(\mathbf{H}_{n+1}) < \delta(e_{m+1})$$

Lorsque l'hypothèse synchrone est vérifiée pour une horloge vis-à-vis d'une suite d'événements, il ne peut arriver au plus qu'un événement de la suite entre deux instants de l'horloge.

Il peut être intéressant de rester dans le cadre de l'hypothèse synchrone même si elle n'est pas vérifiée à chaque instant. Ceci est possible lorsque l'hypothèse synchrone est vérifiée en moyenne :

Définition IV.12 *On dit que l'hypothèse synchrone est vérifiée en moyenne pour une horloge* **H** *vis-à-vis d'une suite d'événements* $(e_n)_{n\in\mathbf{N}}$ *si pour toute fonction de datation* δ *on a :*

- $\exists (k_t)_{t\in\mathbf{N}}, \exists \mathbf{K} \in \mathbf{N}, \forall\, t \in \mathbf{N}, k_t \in \mathbf{N}, k_{t+1} - k_t \leq K$, *et*
- $\langle \delta(\mathbf{H}_{n+1}) - \delta(\mathbf{H}_n) \rangle_{[k_t, k_{t+1}[} \leq \langle \delta(e_{n+1}) - \delta(e_n) \rangle_{[k_t, k_{t+1}[}$

où $\langle v \rangle_I$ *désigne la moyenne temporelle de v sur l'intervalle I.*

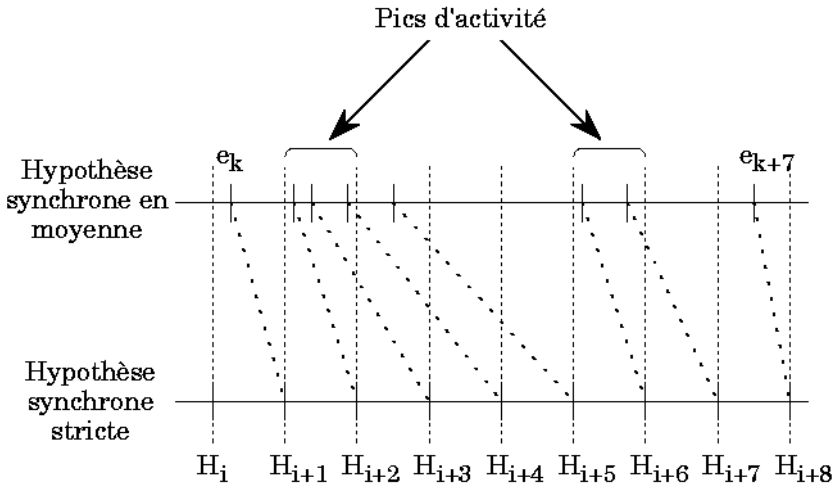

FIG. IV.4 – Hypothèse synchrone stricte ou en moyenne

Cette hypothèse signifie que l'on n'a jamais besoin de plus de **K** instants de l'horloge pour rétablir l'hypothèse synchrone stricte en redistribuant les événements dans le temps.

On peut ainsi transformer une suite d'événements pour laquelle l'hypothèse synchrone est vérifiée en moyenne en une suite d'événements pour laquelle l'hypothèse synchrone est vérifiée strictement, sans perdre un seul événement.

L'hypothèse synchrone en moyenne permet de traiter les « arrivées sporadiques par rafales » [NAS-92] dans le cas où le nombre d'événements est borné sur tout intervalle de temps.

Ceci est intéressant dans le cas où l'exactitude du comptage des occurrences d'un événement est importante. Par exemple, dans un système de surveillance d'un musée, on veut compter les visiteurs afin de pouvoir vérifier que leur nombre est bien nul à la fermeture.

Connaissant les lois statistiques des entrées et des sorties de visiteurs, on peut déterminer un rythme moyen qui permet de dimensionner le système informatique utilisé pour la surveillance.

Afin d'éviter les fausses alertes, ou au contraire, les présences à la fermeture non détectées, il est important de ne rater aucun visiteur, même si temporairement le rythme des entrées et/ou des sorties s'accélère.

Pour que ceci soit possible, il faut que le modèle statistique décrivant la distribution des entrées et sorties de visiteurs permette de déterminer la constante **K** intervenant dans la définition de l'hypothèse synchrone en moyenne. Cette constante détermine en effet la taille des files où seront placés les événements en attente de traitement.

V Modèle objet

Nous présentons ici un modèle des classes d'objets synchrones, de l'héritage entre ces classes et donnons les propriétés des objets synchrones induites par la définition de leur classe.

V.1 Classes synchrones

Une classe est un gabarit d'objet. Elle décrit la structure et le comportement d'un ensemble d'objets appelés ses instances. Chaque objet a son propre état, mais la manière de passer d'un état à un autre est décrite dans la classe.

Définition V.1 *On appelle classe synchrone un quintuplé* $\mathbf{C} = \langle s_I, s_O, \Upsilon, \Sigma, \mathscr{R} \rangle$ *tel que :*

- $s_I \text{et} s_O \in \mathscr{P}_f(\mathscr{N})$
- $\Upsilon : s_I \cup s_O \rightarrow \mathscr{T}$
- $\Sigma = [0, k], k \in \mathbf{N}$ *est l'*ensemble des états *des instances*
- $\mathscr{R} : (\mathscr{V} \times \mathscr{B})^{s_I} \times \Sigma \rightarrow (\mathscr{V} \times \mathscr{B})^{s_O} \times \Sigma$
  $\mathscr{R}$ *est la* fonction de réaction *de la classe.*

s_I et s_O sont les ensembles des noms des signaux d'entrée et de sortie de la classe. Ces noms sont les noms locaux des signaux des instances.

La fonction Υ donne le type des signaux des instances. Le nom local et le type des signaux sont donc des caractéristiques de la classe. Ils définissent complètement l'interface des instances de cette classe vis-à-vis des autres objets.

La fonction de réaction est aussi une caractéristique de la classe. Son profil est identique à celui de la fonction de réaction d'une instance, excepté qu'il fait appel à l'évaluation des signaux à partir de leur nom local.

Il est important de noter qu'une classe synchrone n'étant pas un objet synchrone, elle n'appartient à aucune horloge. Il n'y a donc pas de fonction d'évaluation pour ses signaux qui n'ont par conséquent ni valeur, ni type. La fonction Υ permet de spécifier que le signal de nom local `sig` d'une instance de la classe aura le type Υ(`sig`).

Il faut maintenant préciser comment la structure et le comportement d'un objet synchrone se déduisent de sa classe.

Propriété V.1 *Une instance* $\mathbf{M} = \langle \eta, s_I, s_O, \Sigma, \mathscr{R} \rangle$ *d'une classe synchrone* $\mathbf{C} = \langle s_{I_c}, s_{O_c}, \Upsilon, \Sigma_c, \mathscr{R}_c \rangle$ *vérifie :*

(i) $s_I = \eta \bullet s_{I_c}, s_O = \eta \bullet s_{O_c}$

(ii) $\forall s \in s_{I_c} \cup s_{O_c}, \Upsilon(s) = typ(\eta \bullet s)$

(iii) $\Sigma = \Sigma_c$

(iv) $\mathscr{R} = \pi_M \circ \mathscr{R}_c \circ \pi_M^r$

Où π_M *est la fonction de* $(\mathscr{V} \times \mathscr{B})^{s_I} \times \Sigma$ *sur* $(\mathscr{V} \times \mathscr{B})^{s_{I_c}} \times \Sigma_c$ *qui associe à un couple* (f, ξ) *le couple* (f', ξ) *tel que* $f'(s) = f(\eta \bullet s)$*, et* π_M^r *la fonction de* $(\mathscr{V} \times \mathscr{B})^{s_{O_c}} \times \Sigma_c$ *sur* $(\mathscr{V} \times \mathscr{B})^{s_O} \times \Sigma$ *qui associe à un couple* (f, ξ) *le couple* (f', ξ) *tel que* $f'(\eta \bullet s) = f(s)$.

Le point i) indique que le nom d'un signal d'une instance se construit en préfixant le nom local du signal par le nom de l'instance.

Le point ii) exprime le fait que le signal dont le nom est ainsi construit doit avoir le type déclaré dans la classe grâce à la fonction Υ.

Le point iii) indique que l'ensemble des états de l'instance est celui défini dans la classe. Enfin, le point iv) permet de calculer la réaction d'une instance à partir de la fonction de réaction de sa classe.

On voit donc que le seul élément qui soit propre à une instance est son nom, le nom de ses signaux se déduisant de son nom et de sa classe, l'ensemble de ses états étant celui de sa classe et sa fonction de réaction se déduisant de celle de sa classe. On peut donc décrire un objet synchrone par son nom et sa classe, c'est-à-dire comme une paire $\langle \eta, \mathbf{C} \rangle$.

Les fonctions π_M et π_M^r permettent de créer un contexte d'évaluation pour la fonction de réaction $\mathscr{R}$ de la classe, c'est-à-dire une fonction d'évaluation des noms locaux des signaux et un état.

La fonction de réaction de la classe travaille en effet sur les noms locaux des signaux et sur un état. Les fonctions π_M et π_M^r font correspondre l'état utilisé par la fonction de réaction de la classe à l'état de l'objet et associent aux noms locaux des signaux la valeur des signaux correspondants dans l'objet.

La fonction de réaction de la classe décrit le comportement générique des instances. Sa composition avec π_M et π_M^r applique ce comportement dans le contexte (signaux et état) d'une instance particulière et donne ainsi la fonction de réaction de cette instance :

$$\mathscr{R} : (\mathscr{V} \times \mathscr{B})^{s_I} \times \Sigma \xrightarrow{\pi_M} (\mathscr{V} \times \mathscr{B})^{s_{I_c}} \times \Sigma_c \xrightarrow{\mathscr{R}_c} (\mathscr{V} \times \mathscr{B})^{s_{O_c}} \times \Sigma_c \xrightarrow{\pi_M^r} (\mathscr{V} \times \mathscr{B})^{s_O} \times \Sigma$$

V.2 Héritage entre classes synchrones

Dans un langage à objets, si une classe **B** hérite (on dit aussi : dérive) d'une classe **A**, le protocole de **B** (l'ensemble des messages auxquels répondent les instances de **B**) inclut celui de **A**, et le schéma d'instance de **B** contient celui de **A**. Par contre, la sémantique des méthodes héritées peut être modifiée.

L'héritage n'entraîne donc que des propriétés structurelles sur les classes, on ne peut rien en déduire sur leur comportement. Il en est de même pour les classes synchrones. La propriété suivante exprime cette contrainte structurelle :

Propriété V.2 *Si une classe synchrone* $\mathbf{B} = \langle s_I, s_O, \Upsilon, \Sigma, \mathscr{R} \rangle$ *dérive d'une autre classe synchrone* ***A****, on a :*

- $\mathbf{A}.s_I \subseteq \mathbf{B}.s_I, \mathbf{A}.s_O \subseteq \mathbf{B}.s_O$
- $\mathbf{A}.s_I \cap \mathbf{B}.s_O = \emptyset, \mathbf{A}.s_O \cap \mathbf{B}.s_I = \emptyset$
- $\forall s \in (\mathbf{A}.s_I \cup \mathbf{A}.s_O) \cap (\mathbf{B}.s_I \cup \mathbf{B}.s_O), \mathbf{A}.\Upsilon(s) = \mathbf{B}.\Upsilon(s)$

Comme nous l'avons exposé lors de la présentation informelle de notre système, nous considérons l'héritage comme un moyen de spécialiser le comportement d'une classe. Pour définir l'héritage entre classes synchrones, nous devons trouver un moyen de spécialiser le comportement d'une telle classe. Or ce comportement est pour nous une boîte noire produite par un langage synchrone, et nous n'en voyons que les entrées et les sorties.

Nous choisissons donc de spécialiser le comportement d'une classe synchrone en filtrant ses entrées et ses sorties à l'aide d'objets synchrones. Ce mécanisme de construction de sous-classes synchrones sera présenté dans le chapitre « Outils de Développement ».

VI Relations entre le Modèle Temporel et le Modèle Objet

Dans le modèle temporel de notre système, nous avons présenté les horloges comme des suites d'instants, chaque instant ayant une fonction de connexion et une fonction d'évaluation des signaux. Ce modèle permet d'exprimer des propriétés sur les instants mais n'indique pas comment passer d'un instant au suivant.

Dans le modèle objet nous avons présenté la notion de classe synchrone et exprimé les propriétés statiques d'un objet synchrone en fonction de sa classe.

Nous allons maintenant indiquer comment les actions effectuées dans le cadre du modèle objet s'expriment dans le modèle temporel, c'est-à-dire comment évolue une horloge d'un instant au suivant sous l'effet des opérations effectuées sur les objets.

Les actions principales sont l'instanciation d'une classe, la destruction d'un objet synchrone, la connexion de deux signaux et la réaction d'un objet synchrone. Ces opérations déterminent des propriétés de l'instant suivant de l'horloge dans laquelle elles ont lieu. Si elles sont effectuées à l'instant i d'une horloge $\mathbf{H}$, leurs effets se font sentir à l'instant $i+1$ de $\mathbf{H}$. On peut en effet considérer ces actions comme des événements auxquels l'horloge réagit en modifiant son état. Cette réaction de l'horloge correspond à la détermination de l'instant suivant de l'horloge.

VI.1 Instanciation d'une classe synchrone

L'instanciation d'une classe synchrone est la création d'un objet de cette classe dans une horloge.

L'objet fera partie de l'instant de l'horloge suivant celui auquel sa création a été demandée. Il y persistera jusqu'à ce qu'une requête de destruction de cette objet soit formulée.

Chaque instanciation d'une classe produit un objet distinct de tous ceux qui existent ou ont existé auparavant.

Définition VI.1 *L'instanciation d'une classe synchrone* $\boldsymbol{C}$ *dans une horloge* $\boldsymbol{H}$ *à l'instant n de cette horloge, que l'on note* $\mathrm{new}_{H_n}\mathbf{C}$ *est la production d'un objet synchrone* $\mathbf{M} = \langle \eta, \xi, \mathbf{C} \rangle$ *tel que :*

(i) $\mathbf{M} \notin \mathbf{H}_n.\mathbf{T}$

(ii) $\mathbf{M} \in \mathbf{H}_{n+1}.\mathbf{T}$

(iii) $\mathscr{S}_{\mathbf{H}_{n+1}}(\mathbf{M}) = 0$

Le point i) exprime le fait que l'instanciation produit un nouvel objet, différent de ceux qui existent à l'instant n dans l'horloge.

Le point ii) indique que le nouvel objet apparaît dans l'horloge à l'instant $n+1$.

La combinaison de ces deux points, de la définition des horloges et de la définition de l'ensemble $\mathscr{H}$ des horloges, implique que le nouvel objet est différent de tous ceux qui ont appartenu aux $\mathbf{H}_i$ pour $i \leq n$ et de tous les objets appartenant à des instants quelconques à d'autres horloges.

Le point iii) précise que l'état initial de l'objet est 0.

Il est important de noter que l'instanciation, bien que provoquée par la réaction d'un objet à l'instant n de l'horloge, ne survient qu'à l'instant $n+1$. L'ensemble des objets présents dans une horloge à un instant est en effet parfaitement défini et ne peut pas varier au cours de l'instant puisque qu'un instant à théoriquement une durée nulle.

On peut considérer un instant d'une horloge comme une photographie (notion de snapshot). Un objet ne peut pas apparaître ou disparaître au cours d'un instant car cela rendrait la photographie floue.

On pourrait traiter ce problème de la même manière que l'émission des signaux : si un objet doit apparaître à un instant, il est présent dans l'horloge à cet instant. Mais la présence de cet objet pourrait modifier l'instant de façon à ce que la cause de son instanciation disparaisse. On aurait alors une erreur de causalité.

Les erreurs de causalité sont détectées par les compilateurs de langages synchrones car les signaux et les modules y sont définis statiquement. Il est donc possible d'envisager a priori toutes les conséquences de l'émission d'un signal, et donc de repérer celles qui entraînent l'absence d'émission de ce signal [GON-88, HAL-93]. Dans le cadre de notre système, où l'instanciation est dynamique, la prise en compte de l'existence d'un objet à l'instant même de la cause de son instanciation nous obligerait à calculer par itérations le point fixe de la fonction d'évaluation. L'absence de point fixe traduirait une erreur de causalité qu'il faudrait alors traiter à l'exécution.

Or la correction d'erreurs de causalité dans un langage synchrone où tout est statique est déjà une chose difficile. Il nous semble donc peu intéressant d'alourdir notre modèle d'exécution avec un calcul de point fixe qui pourrait détecter des erreurs dont le programmeur serait incapable de trouver la cause.

Nous choisissons donc de supprimer toute erreur de causalité potentielle en découplant la cause de l'instanciation et l'apparition de l'objet dans l'horloge. On peut aussi considérer que l'instanciation est un phénomène asynchrone de durée minimale, tout comme le retard.

Les modifications du réseau d'interconnexion entre objets d'une horloge sont traitées de la même façon.

VI.2 Destruction d'un objet synchrone

La destruction d'un objet synchrone fait disparaître cet objet de l'instant suivant de l'horloge.

Les signaux qui sont connectés à une sortie de l'objet détruit ne seront connectés à aucun signal à l'instant suivant, sauf bien sûr si on les connecte explicitement à un autre signal à l'instant auquel on demande la destruction de l'objet.

Définition VI.2 *La destruction d'un objet synchrone **M** à l'instant n d'une horloge **H**, que l'on note $delete_{H_n}\mathbf{M}$ fait disparaître **M** des instants de **H** ultérieurs à n :*

(i) $\mathbf{M} \in \mathbf{H}_n.\mathbf{T}$

(ii) $\mathbf{M} \notin \mathbf{H}_{n+1}.\mathbf{T}$

(iii) $\forall s \in s_I\mathbf{H}_n, \mathscr{C}_{\mathbf{H}_n}(s) \in \mathbf{M}.s_O \Rightarrow \mathscr{C}_{\mathbf{H}_{n+1}}$ *n'est pas définie pour s, sauf si on la définit explicitement par une nouvelle connexion.*

Le point i) impose que l'objet à détruire existe à l'instant auquel on le détruit.

Le point ii) exprime le fait que l'objet détruit ne fait plus partie de l'horloge à l'instant suivant. Comme un objet ne peut appartenir à une horloge que sur un intervalle d'instants, **M** n'appartient à aucun instant de **H** de rang supérieur ou égal à $n+1$.

Le point iii) indique que la fonction de connexion à l'instant $n+1$ n'est pas définie pour les signaux qui étaient connectés à un signal de **M** à l'instant de sa destruction. Ceci n'est vrai que si l'on ne définit pas explicitement $\mathscr{C}_{\mathbf{H}_{n+1}}$ pour ces signaux par une connexion (voir paragraphe suivant).

VI.3 Connexion de deux signaux

La connexion de deux signaux dans une horloge permet de modifier la fonction de connexion de l'instant suivant de l'horloge.

Définition VI.3 *Soit **H** une horloge et les signaux $s_i \in s_I\mathbf{H}_{n+1}$, et $s_o \in s_O\mathbf{H}_{n+1}$. La connexion de s_i à s_o à l'instant n de l'horloge **H**, que l'on note $s_i \ll_{H_n} s_o$ est l'opération qui contraint $\mathscr{C}_{\mathbf{H}_{n+1}}$ de façon à ce que : $\mathscr{C}_{\mathbf{H}_{n+1}}(s_i) = s_o$*

VI.4 Réaction d'un objet synchrone

La réaction d'un objet synchrone modifie son état et fournit une contrainte sur la fonction d'évaluation de l'horloge.

Définition VI.4 *La réaction d'un objet synchrone **M** appartenant à l'horloge **H** à l'instant n, que l'on note* $\mathbf{M}.react_{H_n}$ *est l'opération qui modifie l'état de **M** et contraint* $\mathscr{E}_{H_n}$ *de façon à ce que :*

Si $\mathscr{S}_{H_n}(\mathbf{M}) = \xi$ *et* $\mathbf{M}.\mathscr{R}(\mathscr{E}_{H_n/M.s_I}, \xi) = (\mathscr{E}_{M_o}, \xi')$ *alors :*

(i) $\mathscr{S}_{H_{n+1}}(\mathbf{M}) = \xi'$

(ii) $\mathscr{E}_{H_n/M.s_O} = \mathscr{E}_{M_o}$

Le point i) indique que l'état de l'objet à l'instant $n+1$ est celui calculé par sa fonction de réaction à l'instant n.

Le point ii) indique que la valeur des signaux de sortie de l'objet est celle qui est fournie par sa fonction de réaction. Une conséquence du point ii) est que la valeur des signaux de sortie est fournie instantanément, contrairement au changement d'état qui n'apparaît qu'à l'instant suivant.

VII Modèle d'exécution

Nous présentons ici la manière dont nous faisons réagir les objets synchrones au sein d'une horloge afin que l'exécution d'un programme synchrone respecte le modèle que nous venons d'exposer.

VII.1 Objets synchrones et Contrôleurs

Un objet synchrone peut être utilisé « nu » dans un programme, mais en règle générale, et dès qu'il y a plusieurs objets sur une même horloge, un objet synchrone est associé à un contrôleur. Ce contrôleur est chargé de faire réagir l'objet cycliquement et de le synchroniser avec les autres objets de l'horloge.

Dans chaque cycle, le contrôleur commence par invalider les signaux de sortie de l'objet. La valeur qu'ils portent correspond en effet au cycle précédent et ne peut plus être utilisée par un autre objet. Le contrôleur fait ensuite réagir l'objet qu'il contrôle. Cette réaction se déroule en deux temps.

Dans une première phase, l'objet cherche à évaluer ses signaux d'entrée. Il demande pour cela à chacun de ces signaux de se mettre à jour. Un signal d'entrée se met à jour en demandant la valeur du signal de sortie auquel il est connecté. Cette opération peut être bloquante si le signal de sortie ne connaît pas encore sa valeur.

Dans la deuxième phase, l'objet exécute son code synchrone, change d'état et produit ses signaux de sortie.

Lorsque l'objet a terminé sa réaction, le contrôleur sait que les signaux de sortie sont valides. Les éventuelles requêtes de valeur par des signaux d'entrée d'autres objets, qui étaient suspendues jusqu'alors, peuvent être satisfaites.

Le contrôleur permet donc à un objet synchrone de calculer la fonction d'évaluation restreinte à ses signaux d'entrée pour pouvoir ensuite calculer sa fonction de réaction.

Chacun de ces cycles correspond à un instant de l'horloge. Les cycles des contrôleurs des objets d'une même horloge doivent être synchronisés, et entrelacés de façon à satisfaire les dépendances entre objets. Cette tâche est confiée à un ordonnanceur.

VII.2 Horloge et Ordonnanceur

L'ordonnanceur correspond à la notion d'horloge dans le modèle. Il indique aux contrôleurs le début et la fin des instants. Il peut fonctionner périodiquement ou à la demande.

Dans le premier mode, les instants de l'horloge sont séparés par un certain intervalle de temps compté sur l'horloge du système. Dans le second mode, les instants de l'horloge ne surviennent que lorsque l'on sollicite l'ordonnanceur.

Le mode périodique est adapté aux systèmes qui échantillonnent leur environnement et supposent que les échantillons sont périodiques. C'est le cas des systèmes continus discrétisés.

Le mode déclenché convient mieux aux systèmes discrets, puisqu'il évite de faire réagir les objets lorsqu'aucun événement significatif n'est présent.

VII.3 Ordonnanceur et Contrôleurs

L'ordonnanceur gère l'exécution en parallèle des contrôleurs des objets de l'horloge, des contrôleurs des objets qui calculent les signaux de sortie de l'horloge, et des contrôleurs de ceux qui utilisent ses signaux d'entrée[1].

Il indique le début de l'instant aux contrôleurs qui entament alors un cycle de réaction. La phase de mise à jour des signaux d'entrée débute donc en même temps pour tous les objets. Seuls ceux qui ne dépendent d'aucun autre terminent cette phase sans être bloqués. Ils passent ensuite en phase de réaction et signalent à leur contrôleur que leurs signaux de sortie sont valides.

Ceci débloque la phase de mise à jour des objets qui en dépendent et qui peuvent ainsi réagir à leur tour. Comme il n'y a pas de boucle dans le graphe de dépendance, tous les objets finissent par réagir.

Lorsque tous les contrôleurs ont signalé à l'ordonnanceur qu'ils ont terminé leur cycle, ce dernier déclare la fin de l'instant. Les contrôleurs invalident alors les signaux de sortie de leur objet et attendent le début de l'instant suivant.

L'ordonnanceur, en collaboration avec les contrôleurs des objets de l'horloge, détermine donc la suite des ensembles σ_k permettant le calcul de la fonction d'évaluation de l'horloge.

De façon a respecter le modèle de l'instanciation d'une classe synchrone et de la connexion de deux signaux, les connexions, instanciations ou disparitions d'objets qui sont demandées à un instant sont traitées par l'ordonnanceur entre la fin de cet instant et le début de l'instant suivant. Les nouvelles connexions et les instanciations apparaîtront bien ainsi à l'instant suivant celui auquel elles ont été demandées.

VII.4 Discussion du modèle

Nous exposons ici les raisons qui nous ont conduits à certains choix dans notre modèle d'exécution. Il y a en effet d'autres modèles d'exécution compatibles avec notre modèle temporel. Notre expérience avec l'implémentation

[1] Se référer à la discussion sur la différence entre « signal de sortie » et « sortie », et entre « signal d'entrée » et « entrée » dans le modèle temporel lors de la définition des horloges.

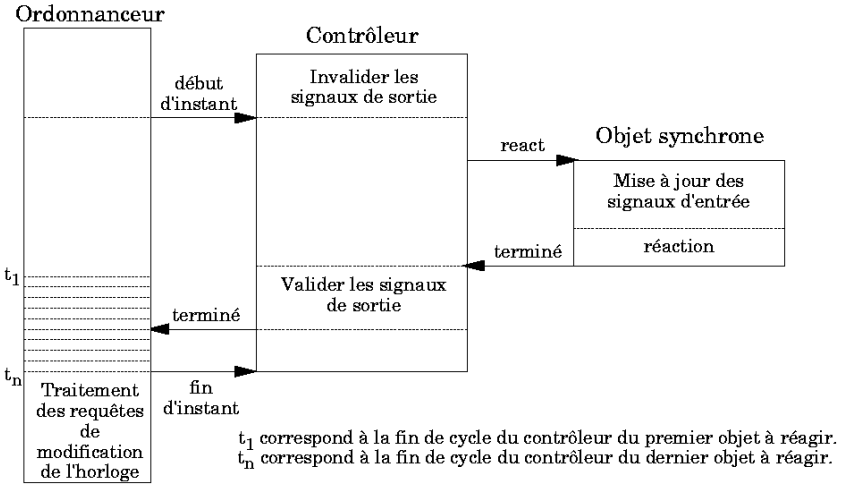

FIG. VII.1 – Ordonnanceur, Contrôleur et objet synchrone

de ce premier modèle d'exécution nous a conduits à en concevoir un nouveau qui sera à la base de la prochaine implémentation de notre système.

VII.4.1 Calcul des signaux de l'horloge

Dans ce modèle, les objets liés aux signaux d'entrée et de sortie de l'horloge sont traités comme les objets synchrones, alors qu'ils n'apparaissent pas dans l'horloge dans le modèle temporel. Ces objets ne sont en effet pas des objets synchrones au sens propre bien qu'ils portent des signaux. Ils fournissent une interface entre l'horloge et le monde asynchrone. Cette double nature leur interdit le statut d'objet synchrone, mais leur composante productrice ou consommatrice de signaux doit s'exécuter dans le contexte des instants de l'horloge.

L'état de ces objets est modifié de façon asynchrone par des comportements qui n'entrent pas dans leur interface vis-à-vis de l'horloge, et ils traduisent ces changements d'états en événements pour les objets synchrones à chaque instant de l'horloge. Leur état peut aussi être modifié par l'occurrence d'un événement dans l'horloge, ce qui modifie leur comportement vis-à-vis du monde asynchrone.

Nous reviendrons sur les différentes fonctionnalités fournies par ces objets. Ces fonctionnalités incluent notamment le retard, déjà présenté dans le modèle temporel.

VII.4.2 Ordonnancement statique ou dynamique

L'ordonnancement des objets synchrones en fonction de leurs dépendances est fait dynamiquement à chaque instant. De plus, la solution retenue met en oeuvre plusieurs flots de contrôle en concurrence pour leur exécution. Tout ceci peut paraître assez lourd, il nous faut donc justifier ce choix et présenter les alternatives que nous avons prévues.

Tout d'abord, étant donné que le nombre d'objets et leurs interconnexions peuvent varier à chaque cycle de l'horloge, on ne peut pas se contenter d'un ordonnancement statique. L'ordonnancement doit être fait à chaque cycle, et nous avons deux possibilités pour le faire : l'ordonnanceur peut analyser les dépendances entre objets, déterminer un ordre de réaction et faire réagir les objets suivant cet ordre, ou il peut les faire réagir tous ensemble et les laisser s'ordonnancer par des contraintes sur l'accès aux valeurs des signaux.

Nous avons choisi la deuxième solution pour la simplicité de son implémentation sur un système temps réel où la création de plusieurs flots de contrôle (ou threads) est peu coûteuse. Nous envisageons toutefois d'implémenter aussi la première solution pour des raisons d'efficacité. En effet, il est peu probable que le système modifie son organisation à chaque instant de l'horloge. Si toutes les requêtes de modification sont traitées par l'ordonnanceur, ce dernier peut déterminer quand les dépendances entre objets ont changé et ne recalculer un ordre d'exécution que dans ce cas.

Pour les horloges dont la configuration est statique, nous proposons un outil de construction d'objets composites qui détermine un ordre de réaction correct une fois pour toutes. On n'est donc pas forcé de supporter l'ordonnancement dynamique lorsqu'on a une configuration statique. Nous reviendrons sur ce point lors de la présentation des outils de développement.

VII.4.3 Transmission de la valeur des signaux

Dans notre modèle, les signaux d'entrée demandent leur valeur aux signaux de sortie. Or habituellement, on propage les modifications des signaux de sortie vers les signaux d'entrée. Cette deuxième solution a l'avantage de ne provoquer une mise à jour des signaux d'entrée que lorsqu'il y a effectivement un changement de valeur.

La solution retenue ici fait cette mise à jour avant chaque réaction. Ce choix a été fait pour deux raisons. La première est que les processus asynchrones (et notamment le monde réel) n'ont pas la notion d'instant d'une horloge. Ils n'ont donc aucun moyen de savoir quand un signal doit être émis, et l'échantillonnage doit donc se faire à la demande de l'objet synchrone. Cet argument n'est toutefois plus valable dans le modèle actuel puisque les objets n'ont plus directement accès au monde asynchrone et obtiennent leurs signaux au travers d'entités particulières qui sont chargées d'échantillonner les valeurs pour eux.

La deuxième raison tient à la dynamicité des connexions entre objets. Si les valeurs sont propagées des signaux de sortie vers les signaux d'entrée, chaque signal de sortie doit maintenir une liste de tous les signaux d'entrée qui lui sont connectés. Si au contraire, ce sont les signaux d'entrée qui demandent la valeur qu'ils doivent prendre, il suffit que chaque signal d'entrée comporte une référence sur le signal de sortie auquel il est connecté, ce qui

est beaucoup plus simple.

Enfin, une dernière raison nous pousse à conserver ce mécanisme de transmission bien que la première raison (historique) soit maintenant caduque : il s'agit du mécanisme d'ordonnancement dynamique. En effet, si seuls les signaux émis sont mis à jour, se pose alors le problème de faire la différence entre un signal non émis et un signal non encore calculé. La solution consiste à renoncer à l'ordonnancement par blocage de threads sur la non-disponibilité de la valeur d'un signal. Ceci rejoint le projet d'implémentation d'un ordonnanceur qui ne recalcule un ordre de réaction que lorsque la configuration du système a changé.

VII.5 Communications entre une horloge et le monde extérieur

Nous présentons ici les outils de base de la communication entre une horloge et le monde extérieur à cette horloge, considéré comme asynchrone.

Le monde réel (c'est-à-dire l'interface que les capteurs et les actionneurs nous présentent de ce monde) est considéré comme un processus asynchrone. Il n'y a donc pour nous aucune différence entre une entrée/sortie et une communication avec une tâche asynchrone s'exécutant sur notre processeur.

VII.5.1 Trois modes de communication

Nous proposons trois modes de communication du monde asynchrone vers le monde synchrone : un mode échantillonné, un mode avec mémorisation de la valeur la plus récente et un mode avec mémorisation de l'historique.

Nous envisageons aussi un mode de communication qui tienne compte des relations entre signaux, mais ce mode n'est pas encore implémenté.

Mode échantillonné

Dans ce mode, la valeur d'un signal est directement liée à l'état du monde asynchrone au début de l'instant. Ce comportement correspond à ce que l'on attend d'un système de traitement du signal. A chaque top de l'horloge, le signal physique est échantillonné et la valeur de l'échantillon donne celle du signal.

Mode mémorisé avec valeur la plus récente

Ce mode correspond au fonctionnement d'un bloqueur : à chaque fois qu'un événement survient dans le monde asynchrone, il est mémorisé, et chaque nouvelle occurrence remplace la précédente. La valeur du signal synchrone correspond à l'occurrence la plus récente de l'événement asynchrone.

Ce mode est utile lorsque seule la dernière occurrence d'un signal fugitif est importante, par exemple le signal « Secondes » émis par un chronomètre lorsqu'il est pris en compte par un module d'affichage.

Mode mémorisé avec historique

Dans ce mode, les occurrences successives d'un événement asynchrone sont mémorisées dans une file. La valeur du signal synchrone correspond à la tête de la file, c'est-à-dire à la dernière occurrence non prise en compte. Ce mode est utile lorsque toute l'histoire du signal asynchrone est significative, ce qui est le cas par exemple pour des impulsions de comptage. Ce mode de communication suppose que l'hypothèse synchrone est vérifiée en moyenne.

Mode avec relations entre signaux

Ce mode prend en compte plusieurs signaux et leurs relations d'exclusion ou de simultanéité. Si deux signaux doivent toujours être émis en même temps et que l'événement asynchrone correspondant au premier signal survient sans celui correspondant au deuxième, on attendra l'occurrence du second événement pour émettre les deux signaux. Si au contraire les deux signaux sont en exclusion et que les événements asynchrones correspondant surviennent en même temps, les deux signaux seront émis l'un après l'autre afin de créer deux événements synchrones distincts respectant les relations entre signaux.

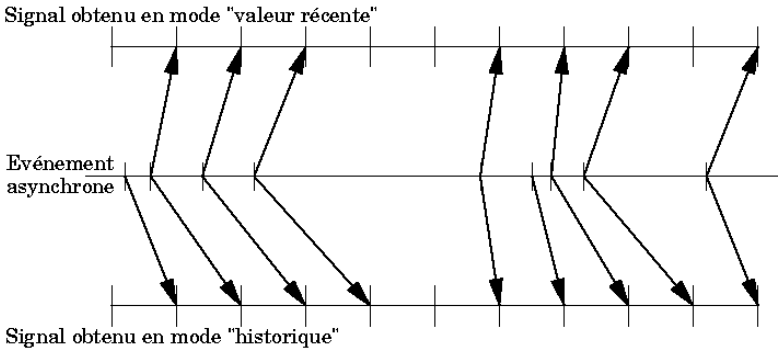

FIG. VII.2 – Mode « récent » et mode « historique »

VII.6 Autre modèle d'exécution envisagé

L'implémentation actuelle de ce premier modèle d'exécution a servi de base au développement de notre système en démontrant sa faisabilité. L'expérience que nous a apporté son utilisation nous amène à envisager un nouveau modèle d'exécution, plus simple, et dans lequel nous essayons d'optimiser les performances sans perdre de souplesse ni de possibilités de vérification des propriétés du modèle. C'est ce nouveau modèle que nous présentons ici, bien qu'il ne soit pas encore implémenté.

Dans ce modèle, la notion de contrôleur disparaît et il n'y a plus d'évaluation concurrente de la réaction des objets. L'ordonnanceur maintient une liste des objets de l'horloge classés par ordre de précédence, et les signaux de sortie maintiennent une liste des signaux d' entrée qui leur sont connectés.

A chaque cycle de l'horloge, l'ordonnanceur commence par vérifier que sa liste d'objets est à jour. Si ce n'est pas le cas, il recalcule un ordre de réaction. Il fait ensuite réagir les objets qui calculent les signaux de sortie de l'horloge (ces objets ne dépendent d'aucun objet de l'horloge). Ces signaux de sortie propagent leur nouvelle valeur aux signaux d'entrée des objets de l'horloge.

C'est ensuite aux objets de l'horloge de réagir, dans l'ordre de la liste. Les valeurs des signaux de sortie se propagent vers les signaux d'entrée qui ont ainsi la bonne valeur quand leur objet réagit.

L'ordonnanceur fait ensuite réagir les objets qui utilisent les signaux d'entrée de l'horloge pour agir sur le monde asynchrone (aucun objet de l'horloge ne dépend d'eux). L'instant est alors terminé.

Enfin, l'ordonnanceur traite les requêtes de modification de l'horloge (nouvelles connexions, instanciations ou disparition d'objets). Ces éventuelles modifications invalident la liste des objets de l'horloge que l'ordonnanceur devra donc ordonner de nouveau au prochain cycle.

La figure VII.3 montre le déroulement d'un instant de l'horloge pour ce modèle d'exécution. Les objets d'interface n'y sont pas représentés. Ils sont traités de la même façon que les objets synchrones et viennent se placer avant le premier objet où après le dernier selon qu'il produisent des signaux de sortie ou utilisent des signaux d'entrée.

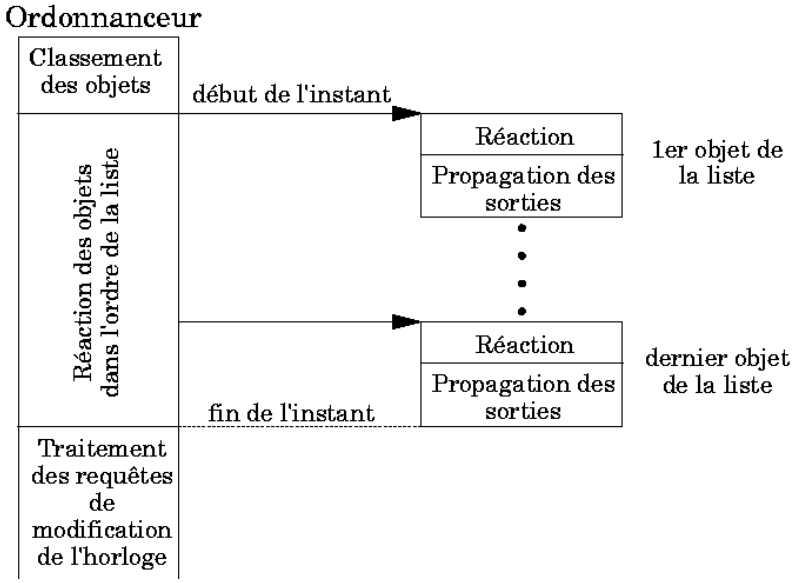

FIG. VII.3 – Autre modèle d'exécution

Ce modèle d'exécution sera retenu pour la prochaine version du système, mais ce changement ne devrait être perceptible pour l'utilisateur que dans le cadre de la conception des objets d'interface synchrone-asynchrone.

VIII 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.

VIII.1 Occ++

Occ++ est un traducteur d'oc en C++. Son nom a été choisi pour qu'il soit reconnu par le compilateur Esterel, ce qui permet de choisir C++ comme langage cible sur la ligne de commande grâce à l'option `-Lc++`.

Ce traducteur n'a été testé qu'avec du code oc généré par le compilateur Esterel V3_20_PRL. Le code C++ produit utilise des templates, le compilateur C++ utilisé doit donc être compatible avec au minimum la version 3 de celui d'ATT.

Occ++ produit une classe pour chaque module présent dans le fichier oc. Il ne crée pas de classe pour les sous-modules. Ces classes sont décrites par deux fichiers : l'un (extension .H) décrit l'interface de la classe, l'autre (extension .C) contient son code.

L'option `-mdf` produit un fichier supplémentaire pour chaque module. Ce fichier décrit l'interface du module (d'où l'extension .mdf pour Module Description File) sous une forme qui pourra être utilisée par le constructeur de modules composites que nous présentons plus loin.

VIII.1.1 Exemple

Prenons un exemple simple pour illustrer le fonctionnement d'occ++. Si nous avons un fichier Esterel demo.strl avec le contenu suivant :

```
module ex_a:
  input i(integer);
  output o;

    ...
    ...

end module

module ex_b:
  input i;
  output o;

    ...
    ...
```

```
end module
```

La commande `esterel -Lc++ demo.strl` produit les fichiers demo.H et demo.C. demo.H contient :

```
//
//  demo.H
//  Created by occ++ v3.2.7 on Tuesday October 12 1993 at 11:57:02
from oc v3 source file oc2312.oc

#include <Esterel/libSync.H>

//
//  Interface of module ex_a
//    root module: ex_a, from file: demo.strl

class ex_a : public Esterel {

    ...

};


//
//  Interface of module ex_b
//    root module: ex_b, from file: demo.strl

class ex_b : public Esterel {

    ...

};
```

La classe Esterel dont dérivent les classes ex_a et ex_b 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.

VIII.2 Mdlc

Mdlc est un outil de construction de modules synchrones. Le module à construire est décrit dans un langage simple : mdl, pour « Module Description Language ».

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. 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.

Ainsi, avec l'option `-mdf`, la ligne de commande esterel de l'exemple précédent devient : `esterel -Lc++ :"-mdf" demo.strl` et produit deux fichiers supplémentaires : ex_a.mdf et ex_b.mdf. Ces fichiers décrivent l'interface des modules ex_a et ex_b en mdl (Module Description Language). A titre d'exemple, le fichier ex_a.mdf contient :

```
//
//  ex_a.mdf
// Created by occ++ v3.2.7 on Tuesday October 12 1993 at 12:54:54
//
#source "demo.strl"
#oc "/tmp/oc2389.oc"
#c++ "demo.H"

ex_a {
  input:
    integer i;
  output: o;
}
```

Ces fichiers peuvent être utilisés pour construire des modules composites ou dérivant d'un autre module. La ligne `#source` indique le fichier contenant le source du module, la ligne `#oc` indique le fichier oc qu'occ++ a traduit (ici, il s'agit d'un fichier temporaire créé par esterel) et la ligne `#c++` indique dans quel fichier se trouve l'interface de la classe synchrone correspondant au module.

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.

VIII.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.

La description d'un module composite se fait en mdl (Module Description Language) de la manière suivante :

```
nom_module : {
    ...
    ...          // Déclarations
    ...
}
```

Les déclarations peuvent être des 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. Un exemple de déclaration de signaux est :

```
input : integer a, b, boolean t ;
```

Ceci déclare un signal d'entrée 'a' de type integer, un signal d'entrée pur (sans valeur) 'b', et un signal d'entrée de type boolean 't'. La déclaration des signaux de sortie se fait de même en remplaçant le mot-clé 'input :' par le mot-clé 'output :'.

Les déclarations de modules indiquent le nom et le type des modules utilisés pour construire le comportement du module composite. Le type du sous-module doit être connu du compilateur. Pour cela, il suffit que le fichier de description du sous-module soit fourni comme argument lors de la compilation du module composite. Ainsi, la déclaration suivante indique que deux modules mod1 et mod2 de type ex_a sont utilisés : `ex_a mod1, mod2 ;`

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. La déclaration suivante connecte l'entrée du sous-module mod2 à la sortie du sous-module mod1 : `mod2.i << mod1.o ;`

Les déclarations d'identité permettent de spécifier qu'un signal du module composite est en fait un signal d'un sous-module. Si l'on veut définir un module composite compo construit à l'aide de deux modules l'un de type ex_a, l'autre de type ex_b, et ayant un signal d'entrée e de type entier et un signal de sortie pur s on déclare :

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

  ex_a mod1;         // Sous-modules utilisés
  ex_b mod2;

  mod1.i = e;        // e est en fait l'entrée i de mod1
  s = mod2.o;        // s est en fait la sortie o de mod2

  mod2.i << mod1.o;  // L'entrée i de mod2 est connectée à
                     // la sortie o de mod1.
}
```

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 VIII.1 :

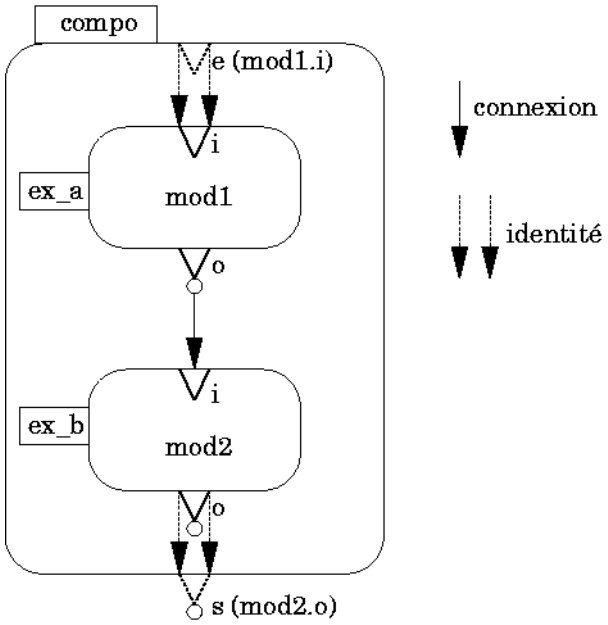

FIG. VIII.1 – Exemple de module compositee

Si la déclaration du module compo est dans un fichier compo.mdl, la ligne de commande : `mdlc compo.mdl ex_a.mdf ex_b.mdf` produit les fichiers compo.H et compo.C. Si l'on passe l'option `-mdf`, le fichier de description compo.mdf est créé :

```
//
//  compo.H
//  Created by mdlc v3.2.0 on Wednesday October 13 1993 at 09:50:12
from mdl source file "compo.mdl"
//

#include <Esterel/libSync.H>

#include "demo.H"

//
//  Interface of module compo
//

class compo : public Composite {

    ...

};
```

La classe synchrone compo dérive cette fois 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.

Le fichier de description compo.mdf décrit l'interface du module composite produit, ce module pouvant à son tour être utilisé pour construire d'autres modules composites :

```
//
//  compo.mdf
// Created by mdlc v3.2.0 on Wednesday October 13 1993 at 10:04:16
//
#source "compo.mdl"
#c++ "compo.H"

compo {
  input: integer e;
  output: s;
}
```

VIII.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 dans le modèle objet, 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. Ainsi, un module dérivé est un module composite dans lequel une copie du super-module est implicite.

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. Son comportement est tel que si le signal `commande` est connecté à une pédale de frein et qu'on relie l'entrée d'un étrier au signal `pression` on obtient le comportement habituel d'un frein : plus on appuie sur la pédale, plus on freine. La description de ce module donne :

```
//
//  frein.mdf
//
#source "frein.strl"
#c++ "frein.H"

frein {
  input: integer commande;
  output: integer pression;
}
```

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 particuliè re 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, ce qui donne :

```
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 :

```
//
//  filtre_ABS.mdf
//
#source "filtre_ABS.strl"
#c++ "filtre_ABS.H"

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

La commande `mdlc -mdf frein_ABS.mdl frein.mdf filtre_ABS.mdf` produit les fichiers frein_ABS.H, frein_ABS.C et frein_ABS.mdf. Le fichier frein_ABS.H contient :

```
//
//  frein_ABS.H
//  Created by mdlc v3.2.0 on Wednesday October 13 1993 at 13:36:12
from mdl source file "frein_ABS.mdl"
//

#include <Esterel/libSync.H>

#include "filtre_ABS.H"

//
//  Interface of module frein_ABS
//

class frein_ABS : public frein, public Composite {

    ...

};
```

On remarque que la classe frein_ABS dérive bien de la classe frein, mais aussi de la classe Composite puisque sa réaction fait appel à celle d'autres objets synchrones. La figure VIII.2 montre la structure des instances de frein_ABS :

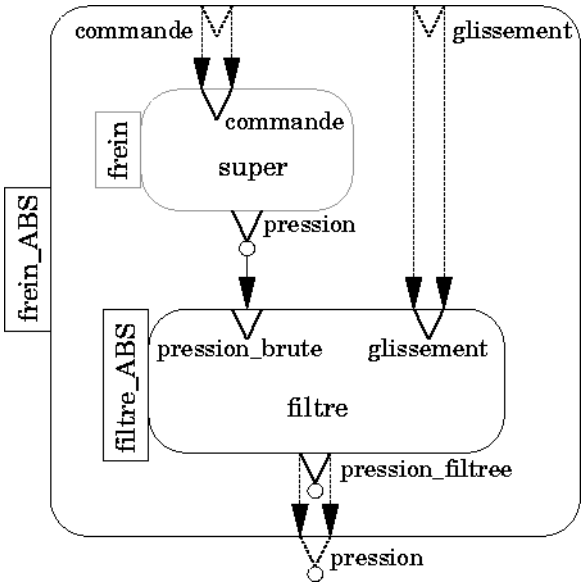

FIG. VIII.2 – Structure des instances de la classe frein_ABS

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'.

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. Comme il est impossible de connecter un signal d'entrée d'un sous-module à un signal de sortie du module, le signal `pression` de la première expression ne peut être que le signal de sortie de 'super'. Dans la deuxième expression, une identité relie toujours un signal d'un sous-module et un signal du module, `pression` ne peut donc être que le signal de sortie du module.

VIII.3 Chaîne de développement

Occ++ permet de traduire des modules synchrones en classes synchrones. Mdlc permet de construire des modules composites et de définir une relation d'héritage entre modules, ce qui permet de hiérarchiser leurs comportements de manière intuitive.

Mdlc traduit directement ces modules en classes synchrones, l'héritage entre modules impliquant l'héritage entre classes.

Le troisième élément fondamental de notre système est la bibliothèque de classes synchrones. Cette bibliothèque contient tous les mécanismes qui implémentent le modèle d'exécution.

Ces trois éléments, le compilateur C++ et les compilateurs de langages synchrones constituent notre système de développement. Il s'agit d'un minimum auquel il serait souhaitable d'ajouter des outils méthodologiques, autant pour l'aspect synchrone que pour l'aspect objet. Dans ce système minimal, le développement d'une application intégrant du code synchrone se fait selon la chaîne de la figure VIII.3 :

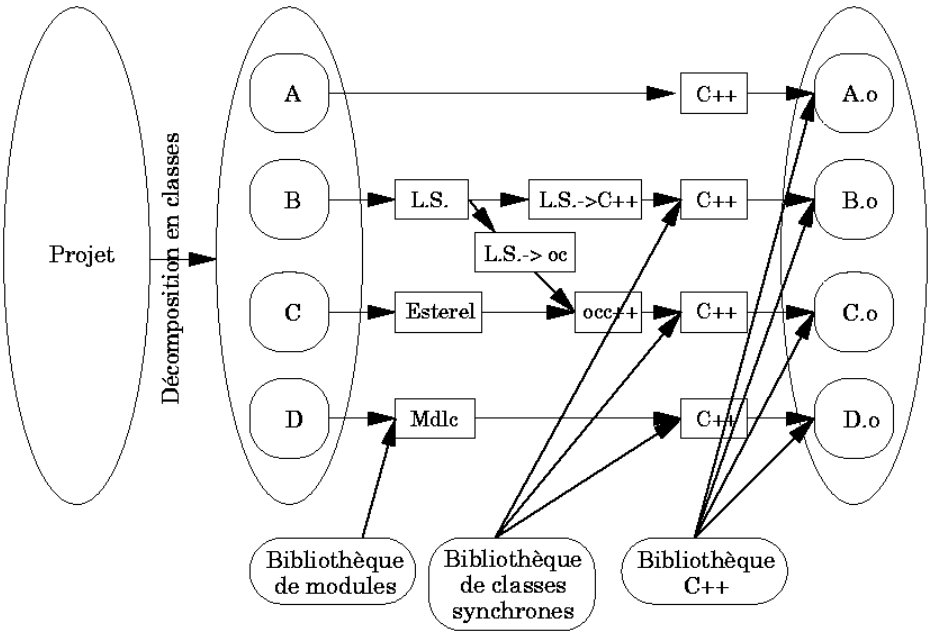

FIG. VIII.3 – Chaîne de développement

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.

Le code synchrone traduit en C++ fait appel à des fonctionnalités de la bibliothèque de classes synchrones. De même, le code C++ utilise des fonctionnalités décrites dans la bibliothèque C++.

IX Implémentation

Nous présentons ici l'implémentation actuelle de notre système d'intégration de modules synchrones en C++. Nous indiquerons parfois les mécanismes que nous comptons implémenter, et il sera alors bien précisé que cela ne fait pas partie de l'implémentation actuelle.

Notre système fonctionne actuellement sur DECstation 5000 sous Ultrix V4.3 et sur une carte MVME-147S-1 de Motorola sous DECelx V1.0 (version de VxWorks par DIGITAL). Les outils de développement occ++ et mdlc fonctionnent sur DECstation 5000 sous Ultrix V4.3 et sur Apple Macintosh sous MPW[1].

Nous avons porté le compilateur C++ version 3 d'ATT sous Ultrix V4.3 afin qu'il génère du code pour le MC68030 de la carte MVME-147S-1 sous DECelx.

La version des outils décrits ici est :

- occ++ version 3.2.8
- mdlc version 3.2.0
- bibliothèque synchrone libSynch version 3.2.0.

Le code de libSynch fait appel à des threads, ou flots de contrôle, pour permettre à plusieurs activités de se dérouler en partageant le temps d'exécution d'un processus. Ce mécanisme étant fortement dépendant du système et du processeur, nous avons créé une bibliothèque de classes permettant d'avoir la même interface pour les threads, événements et sémaphores sous Ultrix et sous DECelx. Cette bibliothèque est le seul élément de notre système qui n'ait pas le même source pour les deux systèmes d'exploitation.

La description complète des outils occ++ et mdlc est donnée en annexe. Nous ne décrivons ici que l'interface des classes de la bibliothèque synchrone et des classes synchrones générées par occ++ ou mdlc.

IX.1 Aperçu des classes de libSynch

La figure IX.1 donne la hiérarchie des classes de libSynch. La classe Scheduler correspond à l'ordonnanceur du modèle d'exécution. Les classes Controlleur et SynControlleur fournissent l'environnement temporel pour la réaction des objets synchrones en liaison avec l'ordonnanceur.

La classe Synchronous est la classe des objets qui savent réagir à des signaux d'entrée en produisant des signaux de sortie. Ceci ne correspond pas exactement à la notion d'objet synchrone puisque cette définition englobe aussi les objets d'interface avec le monde asynchrone.

[1] Macintosh Programmer's Workshop, système de développement pour Macintosh

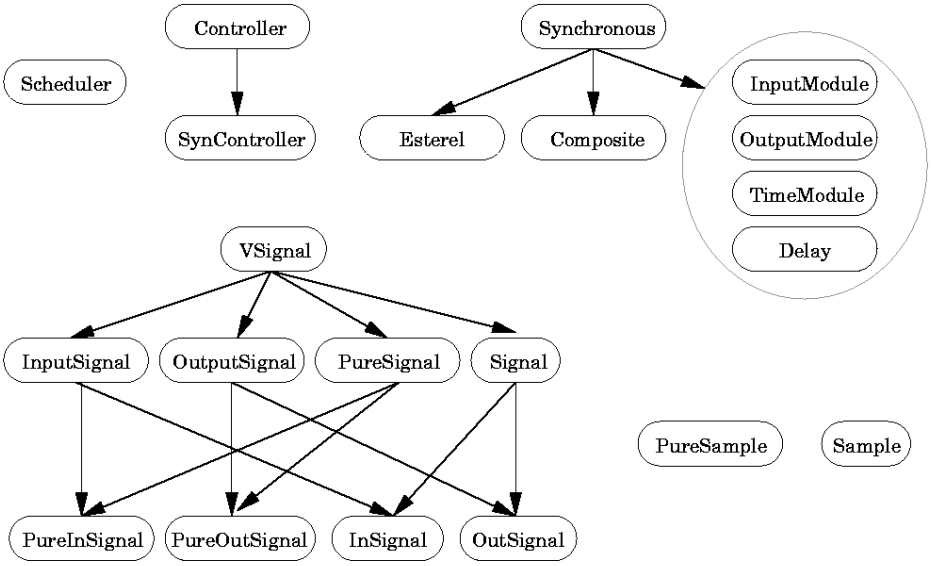

FIG. IX.1 – Les classes de libSynch

De Synchronous dérivent Composite, classe des objets synchrones composés, Esterel, classe des objets synchrones produits par occ++, Delay, classe des retards, et trois classes d'interface : InputModule et OutputModule qui donnent une interface synchrone aux fichiers ouverts en lecture ou en écriture, et TimeModule qui donne une interface synchrone à l'horloge système.

La hiérarchie des signaux est un peu plus complexe. La classe VSignal est une classe abstraite qui implémente le protocole minimal de tous les signaux. Les signaux sont ensuite divisés en quatre catégories : signaux d'entrée (InputSignal), de sortie (OutputSignal), signal pur (PureSignal) ou valué (Signal). Les classes usuelles de signaux sont obtenues par héritage multiple, un lien d'héritage précisant leur nature pure ou valuée, l'autre indiquant leur caractère d'entrée ou de sortie.

Enfin, les classes Sample et PureSample correspondent aux échantillons véhiculés respectivement par les signaux valués et les signaux purs.

Nous commençons la description de ces classes par l'entité la plus élé mentaire : l'échantillon. Nous poursuivrons avec les signaux, puis les objets synchrones, les contrôleurs et enfin l'ordonnanceur.

IX.1.1 Les échantillons

Un signal est une suite d'échantillons. L'échantillon est un instantané du signal à un instant de l'horloge. Il est caractérisé par deux attributs : la présence et la valeur.

L'attribut de présence indique si le signal est émis à l'instant considéré. L'attribut de valeur donne la valeur du signal à cet instant. La valeur ne peut changer que lorsque le signal est émis, c'est-à-dire lorsque l'attribut de présence est vrai.

Nous considérons deux classes principales d'échantillons : les échantillons purs qui ne véhiculent pas de valeur et sont donc simplement présents ou absents, et les échantillons valués, qui portent une valeur d'un type déterminé. Ces deux types d'échantillons sont implémentés par les classes PureSample et Sample.

PureSample

Les échantillons purs ont un seul attribut : la présence. Le protocole de PureSample permet de tester cet attribut, de réinitialiser l'échantillon et de lui affecter l'état d'un autre échantillon (ce qui est utilisé pour les communications). L'interface de la classe PureSample est donc :

```
class PureSample {
  private:
    boolean fPresent;
  public:
    PureSample();
    virtual boolean isPresent() const;
    virtual void reset();
    virtual PureSample& operator=(const boolean& newVal);

    friend istream& operator>>(istream& i, PureSample& s);
    friend ostream& operator<<(ostream& o, PureSample& s);
};
```

L'attribut booléen de présence fPresent est classiquement déclaré privé afin qu'il ne puisse être modifié qu'à travers le protocole de la classe.

La partie publique de l'interface comprend un constructeur PureSample(), chargé d'initialiser l'objet lors de l'instanciation. Ici, l'initialisation se limite à marquer l'échantillon non présent.

La méthode isPresent() permet de tester l'état de l'échantillon. Elle est qualifiée de const car elle ne modifie pas l'état de l'objet. Le mot-clé virtual est rendu nécessaire par la manière curieuse dont C++ considère le polymorphisme. Voir à ce sujet l'introduction sur les langages à objets.

La méthode reset() remet l'échantillon à l'état non présent.

L'opérateur d'affectation est défini ici pour permettre l'affectation d'un booléen à un PureSample. Si ps est une instance de PureSample, on peut donc écrire `ps = true ;` ou `ps = qr.isPresent() ;` si qr est une autre instance de PureSample.

Les deux derniers opérateurs >> et << permettent de placer ou d' extraire une représentation textuelle d'un PureSample sur un flot de caractères. Ce ne sont pas des opérateurs de la classe puisqu'ils s'appliquent à des instances d'istream et ostream. C'est pourquoi ils doivent être déclarés friend afin d'avoir accès à la partie privée du schéma d'instance des PureSample. Cette manière de définir les opérateurs d'entrée/sortie est classique en C++.

Sample

La classe Sample est particulière car elle définit en fait toute une famille de classes. Il s'agit d'un gabarit de classes, ou template. En définissant Sample, nous définissons ce que doit faire un échantillon quel que soit le type de donnée qu'il véhicule. Ce type est donc ici un type muet. Donnons l'interface de Sample pour illustrer ce mécanisme :

```
template<class sigType> class Sample {
  private:
    boolean fPresent;
    sigType fValue;
  public:
    Sample();
    virtual boolean isPresent() const;
    virtual void reset();
    virtual Sample& operator=(const sigType& newVal);
    virtual const sigType& value() const;

    friend istream& operator>>(istream& i, Sample<sigType>& s);
    friend ostream& operator<<(ostream& o, Sample<sigType>& s);
};
```

Le schéma d'instance de Sample comporte un membre privé supplémentaire : fValue. Ce membre est du type muet sigType et contient la valeur de l'échantillon. Le protocole comporte toujours l'opérateur d'affectation qui permet cette fois d'affecter une valeur de type sigType à l'échantillon. Cette affectation à pour effet de ranger la nouvelle valeur dans fValue, mais aussi de marquer l'échantillon comme présent en mettant true dans fPresent. Il est donc ici très important que le schéma d'instance soit déclaré privé afin qu'on ne puisse pas modifier la valeur de l'échantillon sans le rendre présent.

Une méthode supplémentaire est apparue : value(). Cette méthode permet d'obtenir la valeur de l'échantillon. Elle est qualifiée de const car elle ne modifie pas l'objet. La valeur qu'elle rend est elle même qualifiée de const afin qu'il soit impossible de la modifier, ce qui modifierait l'échantillon. Si l'on veut modifier la valeur rendue par value(), il faut en faire une copie.

Lorsqu'on utilise ce gabarit de classes pour un type particulier (on dit qu'on instancie le template pour ce type), le compilateur génère le code C++ correspondant en remplaçant le type muet par le type de l'instanciation. Pour utiliser des échantillons qui véhiculent des caractères, il suffit donc de déclarer : `Sample<char> mySample ;`.

IX.1.2 Les signaux

Les signaux sont les vecteurs de communication entre objets synchrones. Ils peuvent être vus sous deux aspects : la connexion et le transport d'échantillons.

L'aspect connexion couvre la vérification de la validité du réseau d'interconnexion des objets synchrones d'une horloge. Sous cet aspect, le signal

traduit la relation de précédence définie dans le modèle temporel. L'aspect transport d'échantillons couvre la mise à jour des signaux d'entrée à partir de leur source (le signal de sortie auquel ils sont connectés) et la manipulation des échantillons.

VSignal

Cette classe définit l'interface commune à tous les signaux. Chaque signal a en effet un nom, un propriétaire (qui est une instance de Synchronous) et sait s'afficher sur un flot de caractères. L'interface de VSignal est la suivante :

```
class VSignal {
  public:
    VSignal(Synchronous*, const char* name);
    virtual const char* name();
    virtual void chown(Synchronous*);
    friend ostream& operator<<(ostream&, VSignal&);
  protected:
    VSignal();
    Synchronous* fOwner;
    const char* name;
};
```

Le constructeur public de VSignal prend deux arguments : un objet synchrone et une chaîne de caractères. Un signal a donc dès son instanciation un propriétaire et un nom. Le constructeur sans argument dans la partie protégée de l'interface est rendu nécessaire par l'héritage multiple qui sera utilisé pour construire les classes de signaux.

La méthode name() donne le nom du signal. La méthode chown(Synchronous*) permet de changer le propriétaire du signal. Elle est utilisée pour implémenter l'identification de signaux dans les objets composites.

L'opérateur d'affichage << donne une représentation du signal sous la forme nom_propriétaire.nom_signal. La partie protégée de l'interface n'est accessible qu'aux classes qui dérivent de VSignal.

OutputSignal

Cette classe implémente l'aspect connexion des signaux de sortie. Son interface est :

```
class OutputSignal : virtual public VSignal {
  public:
    OutputSignal(Synchronous* s, const char* name);
    virtual boolean checkLoopFor(Synchronous* s);
  protected:
    OutputSignal();
};
```

On retrouve le constructeur avec propriétaire et nom, ainsi que le constructeur protégé sans arguments pour l'héritage multiple. On remarque

que OutputSignal dérive virtuellement de VSignal. Ceci signifie que le schéma d'instance d'OutputSignal ne sera présent qu'une fois dans celui d'une classe dérivée, même si OutputSignal apparaît plusieurs fois dans le graphe d'héritage, ce qui peut arriver avec l'héritage multiple.

La méthode checkLoopFor(Synchronous* s) permet de vérifier que le signal de sortie ne dépend pas de l'objet s. L'algorithme utilisé est relativement simple : partant du propriétaire du signal, on demande à chacun de ses signaux d'entrée de faire la même vérification. On fait ainsi un parcours en profondeur d'abord qui est interrompu dès qu'on rencontre l'objet s. On a alors détecté une boucle. Si on parcourt tout le graphe sans rencontrer s, on sait que le graphe ne contient pas de boucle passant par s.

InputSignal

Cette classe implémente l'aspect connexion des signaux d'entrée. Son interface est :

```
class InputSignal : virtual public VSignal {
  public:
    InputSignal(Synchronous* s, const char* name);
    virtual boolean checkLoopFor(Synchronous* s);
    virtual boolean checkLoopThrough(OutputSignal& o);
  protected:
    InputSignal();
    OutputSignal* source;
};
```

La méthode checkLoopFor(Synchronous* s) complète celle de la classe OutputSignal. Elle explore le réseau d'interconnexion à partir du propriétaire du signal de sortie auquel le signal est connecté. La méthode checkLoopThrough(OutputSignal& o) permet de vérifier si la connexion du signal au signal de sortie o ferme une boucle ou non. Elle active pour cela la méthode checkLoopFor du signal o avec le propriétaire du signal comme argument.

On remarque un nouveau membre dans la partie protégée du schéma d'instance : source. Il s'agit du signal de sortie auquel est connecté le signal et qui est donc sa source d'échantillons. Comme seul l'aspect connexion est traité à ce niveau, le type du signal n'est pas encore précisé. Cette information n'est en effet pas nécessaire pour le parcours du réseau d'interconnexion. Elle ne sera indispensable que lorsqu'il s'agira d'effectuer une connexion, l'égalité des types des signaux à connecter devant alors être vérifiée.

PureSignal

Cette classe implémente l'aspect transport d'échantillons pour les signaux purs, c'est-à-dire les signaux qui ne véhiculent pas d'autre information que leur présence ou leur absence. L'interface de cette classe est :

```
class PureSignal : virtual public VSignal {
```

```
  public:
    PureSignal(Synchronous*, const char*);
    virtual PureSample& value();
  protected:
    PureSignal();
    PureSample fValue;
};
```

La méthode value() permet d'obtenir l'échantillon courant du signal. Il s'agit ici d'une instance de PureSample puisque le signal est pur.

Signal

On retrouve ici un gabarit de classes, comme lors de la définition de Sample :

```
template class<sigType> class Signal : virtual public VSignal {
  public:
    Signal(Synchronous*, const char*);
    virtual Sample<sigType>& value();
  protected:
    Signal();
    Sample<sigType> fValue;
};
```

La méthode value() retourne cette fois un Sample<sigType> puisque le signal véhicule des données de type sigType. On retrouve l'échantillon courant dans la partie protégée du schéma d'instance.

PureOutSignal

Cette classe correspond à des signaux « réels », c'est-à-dire qu'elle implémente, grâce à l'héritage multiple, à la fois l'aspect connexion et l'aspect transport d'échantillons. Il s'agit ici d'échantillons purs :

```
class PureOutSignal : virtual public PureSignal,
                      public OutputSignal {
  public:
    PureOutSignal(Synchronous*, const char*);
    virtual PureSample& sample();
  protected:
    PureOutSignal();
};
```

La méthode sample() donne l'échantillon du signal correspondant à l'instant courant de l'horloge. Elle n'est donc pas identique à la méthode value() qui donne l'échantillon courant. Nous avons en effet vu que les objets de l'horloge réagissent dans un ordre compatible avec leurs relations de dépendance. Lors de l'activation de la méthode value(), l'échantillon courant peut ne pas être celui de l'instant courant de l'horloge si le propriétaire du signal n'a pas encore réagit. La méthode sample() attend que l'échantillon soit à jour pour l'instant courant de l'horloge avant de le rendre. Ce mécanisme de synchronisation fait intervenir le contrôleur du propriétaire du signal.

OutSignal

Cette classe est la contrepartie de PureOutSignal pour les signaux valués. Il s'agit donc d'un gabarit de classes :

```
template<class sigType> class OutSignal :
                 virtual public Signal<sigType>,
                 public OutputSignal {
  public:
    OutSignal(Synchronous*, const char*);
    virtual Sample<sigType>& sample();
  protected:
    OutSignal();
};
```

Au type muet près, les méthodes sont les mêmes que pour PureOutSignal.

PureInSignal

Cette classe correspond aux signaux d'entrée purs :

```
class PureInSignal : virtual public PureSignal,
                     public InputSignal {
  public:
    PureInSignal(Synchronous*, const char*);
    virtual void operator<<(PureOutSignal&);
    virtual void update();
  protected:
    PureInSignal();
};
```

On trouve enfin ici l'opérateur de connexion <<. Son prototype assure qu'un signal d'entrée pur ne peut être connecté qu'à un signal de sortie pur. Cette vérification s'appuyant sur les types, elle est faite statiquement à la compilation.

La connexion à un signal o n'est effectuée que si checkLoopThrough(o) retourne faux, c'est-à-dire si cette connexion ne ferme pas une boucle. Cette vérification ne peut être faite qu'à l'exécution. Comme il n'y a au départ aucune connexion et que chaque nouvelle connexion n'est effectuée que si elle ne ferme pas de boucle, on est assuré de n'avoir aucune boucle dans le réseau d'interconnexion.

La méthode update() fournit le mécanisme de mise à jour d'un signal d'entrée à partir du signal de sortie auquel il est connecté. Cette méthode active la méthode sample() du signal source et range le ré sultat dans l'échantillon du signal. L'utilisation de la méthode sample() au lieu de la méthode value() garantit que l'on obtient l'échantillon de l'instant courant de l'horloge.

InSignal

Cette classe correspond à PureInSignal pour des signaux valués. Il s'agit donc d'un gabarit de classes :

```
template<class sigType> class InSignal :
                  virtual public Signal<sigType>,
                  public InputSignal {
  public:
    InSignal(Synchronous*, const char*);
    virtual void operator<<(OutSignal<sigType>&);
    virtual void update();
  protected:
    InSignal();
};
```

Cette fois, le prototype de l'opérateur de connexion << assure qu'un signal d'entrée véhiculant des données de type sigType ne peut être connecté qu'à un signal de sortie véhiculant des données du même type. De même que pour PureInSignal, la connexion n'est effectuée que si elle ne ferme pas de boucle.

Récapitulatif

Le fait que les signaux soient des suites d'échantillons d'un type donné assure le respect de la propriété du modèle qui affirme que la valeur d'un signal est toujours du même type.

Les signaux assurent, par le prototype de leurs méthodes de connexion, qu'un signal d'entrée ne peut être connecté qu'à un signal de sortie véhiculant des valeurs de même type. Ceci permet de vérifier les propriétés correspondantes du modèle. La vérification de ces propriétés s'appuie sur la vérification de la correspondance des types par le compilateur C++ et est donc statique.

Les propriétés suivantes sont vérifiées dynamiquement au cours de l'exécution. La vérification de l'absence de boucle lors des connexions permet de respecter la propriété du modèle qui affirme que le graphe de dépendance des objets synchrones est acyclique. Le comportement du système peut conduire à des tentatives de violation de cette propriété. Il s'agit alors d'une erreur dans la programmation du système d'objets synchrones composant l'horloge.

Le respect de l'égalité des signaux connectés s'appuie sur la synchronisation des objets grâce aux contrôleurs. Le comportement du système ne peut pas conduire à une violation de cette propriété (à moins qu'il y ait une erreur dans libSynch) qui peut donc être toujours considérée comme valide.

IX.1.3 Les objets synchrones

Les objets synchrones sont les briques de base pour la construction de la partie synchrone d'une application. Ils comprennent les objets synchrones au sens strict du modèle, c'est-à-dire les objets qui ne communiquent que par signaux et qui ne changent d'état qu'aux instants de l'horloge.

Mais la hiérarchie des classes issues de Synchronous comprend aussi les classes d'objets qui interfacent l'horloge avec le monde asynchrone. Ces objets ont une partie synchrone, activée aux instants de leur horloge, et une partie asynchrone qui peut être activée en dehors de ces instants par le monde asynchrone.

Dans ce schéma, la classe Delay, qui implémente le retard, a un statut particulier : bien qu'elle ne comporte pas de code activable de manière asynchrone, il ne s'agit pas d'une classe synchrone à proprement parler. Ses instances attendent en effet la fin de l'instant de l'horloge pour calculer le signal retardé, ce qui permet de réaliser le « plus petit asynchronisme » décrit dans le modèle. Cet asynchronisme permet de couper la relation de dépendance entre le signal retardé et le signal d'origine, et il est rendu suffisamment petit pour que le signal retardé soit disponible à l'instant suivant de l'horloge.

Synchronous

Synchronous est la classe de base pour toutes les classes qui implémentent un comportement synchrone. Nous allons voir sur quels mécanismes elle s'appuie en détaillant son protocole :

```
class Synchronous {
  public:
    Synchronous(const char* name, const boolean sched = true);
    virtual ~Synchronous();
    virtual const char* name();
    virtual void react();
    virtual InputSignal** inSigList() = 0;
    virtual void synchronizeOn(Controller* ctrl);
    virtual void synchronize();
    virtual void endInstant();
    virtual void killSched();
  protected:
    Controller* fCtrl;
    virtual void resetOutputs() = 0;
    virtual void setInputs() = 0;
    virtual void activate() = 0;
  private:
    const char* name;
};
```

Le constructeur prend pour arguments le nom de l'objet et un booléen indiquant si l'objet doit être créé dans l'horloge courante, ce qui est le cas par défaut. La notion d'horloge courante permet d'éviter de spécifier l'horloge lors de chaque instanciation d'un objet synchrone.

~Synchronous() est le destructeur. Cette méthode est appelée automatiquement lorsque l'objet est détruit et permet de libérer les ressources allouées à l'objet.

name() retourne sans surprise le nom de l'objet. Ce nom, ainsi que celui des signaux, est particulièrement utile pour retrouver la source de l'erreur lorsque la fermeture d'une boucle dans le réseau d'interconnexion est détectée.

La méthode react() fait réagir l'objet synchrone à ses signaux d'entrée. Elle se décompose en trois phases : l'invalidation des signaux de sortie, la mise à jour des signaux d'entrée et enfin l'activation du code synchrone de l'objet.

Les méthodes qui traitent chacune de ces phases sont déclarées dans la partie protégée de l'interface en tant que méthodes virtuelles pures. Elles font en effet partie du protocole des objets synchrones, mais leur corps ne peut être défini que pour une classe particulière.

La méthode inSigList() retourne une liste des signaux d'entrée dont dépend le comportement de l'objet à cet instant de l'horloge. Pour les objets synchrones construits à partir d'un module d'un langage synchrone, il s'agit de la liste des signaux d'entrée puisque nous considérons le comportement synchrone d'un tel objet comme une boîte noire. Tous les signaux de sortie dépendent a priori de tous les signaux d'entrée. Un exemple d'objet pour lequel la liste retournée par inSigList() n'est pas la liste des signaux d'entrée de l'objet est le retard. Dans un retard, le signal de sortie ne dépend pas du signal d'entrée, mais de la valeur du signal d'entrée à l'instant précédent de l'horloge. Ainsi, malgré la présence d'un signal d'entrée, la liste rendue par la méthode d'un retard inSigList() est vide.

La méthode synchronizeOn(Controller* ctrl) permet de synchroniser l'objet sur un contrôleur particulier. Lorsque l'objet est directement créé dans une horloge, ce contrôleur est fourni automatiquement par l'ordonnanceur de l'horloge.

La méthode synchronize() permet d'attendre que les signaux de sortie de l'objet soient calculés. Cette méthode fait appel au contrôleur de l'objet et suspend l'exécution de l'appelant jusqu'à ce que la réaction de l'objet ait eu lieu. C'est grâce à cette méthode que se fait l'ordonnancement dynamique à chaque instant de l'horloge, la mise à jour des signaux faisant appel à cette synchronisation.

La méthode endInstant() est activée pour signaler à l'objet que l'instant de l'horloge vient de se terminer. Son corps est vide pour les objets synchrones au sens strict. Elle est utilisée par le retard pour déterminer quand la valeur de son signal de sortie peut changer.

La méthode killSched() a un statut particulier. Elle n'est pas indispensable au fonctionnement du système mais permet d'écrire des programmes qui s'arrêtent. Lorsque cette méthode est activée à un instant, l'ordonnanceur s'arrête à la fin de l'instant. Lorsque tous les ordonnanceurs d'un programme s'arrêtent, le programme peut s'arrê ter. Sans ce mécanisme, le seul moyen d'arrêter un programme serait de tuer son processus au niveau du système d'exploitation. Cette solution peut être inacceptable dans certains cas où le système piloté par l'application doit être amené dans une certaine configuration avant d'être arrêté.

La partie protégée de l'interface contient un pointeur sur le contrôleur de l'objet et les trois méthodes virtuelles pures nécessaires à la réaction de l'objet. La méthode resetOutputs() met les signaux de sortie dans l'état non émis. L'émission ou non d'un signal sera déterminée par la réaction de l'objet.

setInputs() provoque la mise à jour des signaux d'entrée. Comme nous l'avons vu, cette mise à jour se fait par la méthode update() des signaux qui

active la méthode sample() du signal source. sample() provoque la synchronisation de l'objet avec son contrôleur (méthode synchronize() de l'objet) ce qui peut être bloquant si le signal source n'est pas encore disponible.

Lorsque les signaux d'entrée ont été mis à jour, la méthode activate() exécute le code synchrone de l'objet, ce qui a pour effet de le placer dans un nouvel état et de donner une valeur à ses signaux de sortie.

Pour se replacer dans le contexte du modèle, setInputs() permet d'obtenir la fonction partielle d'évaluation $\mathscr{E}_{H_n}/\mathbf{M}.s_I$. resetOutputs() et activate() correspondent à la fonction de réaction $\mathscr{R}$, resetOutputs() initialisant les signaux de sortie de façon à ce que activate() ne modifie que les signaux émis.

Esterel

La classe Esterel définit le protocole commun à toutes les classes synchrones produites par occ++. On y trouve le moteur de l'automate produit à partir du code oc.

```
class Esterel : virtual public Synchronous {
  public:
    typedef unsigned short tStateNumber;
    typedef unsigned short tActionNumber;
    static const tActionNumber kGotoAction;
    Esterel(const tStateNumber *const *const s);
  protected:
    const tStateNumber *const *const cStateTable;
    tStateNumber fState;
    virtual void activate();
    virtual void doAction(tActionNumber a, tActionNumber *i)=0;
    void branchIfFalse(const boolean c, tActionNumber *i);
};
```

Dans la partie publique de l'interface, on définit deux types : tStateNumber et tActionNumber qui servent à représenter les états et les actions de l'automate. Ces deux types doivent être définis de la même façon car les états et les actions sont rangés dans le même tableau. La définition de deux types permet toutefois de préciser comment un élément de ce tableau est interprété par une méthode.

kGotoAction est la valeur réservée à l'action de changement d'état. Il s'agit d'une constante, et elle est déclarée static car sa valeur est partagée par toutes les instances d'Esterel.

Le constructeur d'Esterel prend pour argument la table des états et actions de l'automate. Le nom de l'objet ne lui est pas fourni car le nommage des objets synchrones relève du protocole de Synchronous.

Dans la partie protégée de l'interface, on trouve la table des états et actions cStateTable. Les trois const intervenant dans sa déclaration indiquent qu'il s'agit d'un pointeur constant de pointeurs constants sur des constantes. Ceci permet d'assurer que la table de l'automate ne peut pas être modifiée

une fois que l'automate est créé. Nous avons en effet vu dans le modèle que la fonction de réaction d'un objet synchrone est une donnée statique de cet objet.

Le membre fState contient l'état courant de l'automate et correspond à la valeur de la fonction d'état de l'horloge pour l'objet synchrone.

La méthode activate() est le moteur de l'automate. Elle interprète la table de l'automate et provoque l'exécution des actions ainsi que les changements d'état.

doAction(tActionNumber a, tActionNumber *i) exécute le code associé à une action. Il s'agit d'une méthode virtuelle pure car elle doit faire partie du protocole d'Esterel mais ne peut être définie que pour une classe synchrone particulière, le code associé à une action dépendant de l'objet synchrone considéré. a est le code de l'action à exécuter et i pointe sur la prochaine action. i est passé par référence car la prochaine action à exécuter peut être modifiée dans le cas où l'action courante est un branchement.

branchIfFalse(const boolean c, tActionNumber *i) implémente le branchement. Si la condition c est vérifiée, i est incrémenté pour passer à l'action suivante. Dans le cas contraire, i prend la valeur correspondant à l'action à laquelle il faut sauter.

Composite

La classe Composite définit le protocole des classes produites par mdlc. Ces classes correspondent à des modules composés de sous-modules qui peuvent hériter d'autres modules.

```
class Composite : virtual public Synchronous {
  protected:
    static char* subname(const char* mod, const char* sub);
    Composite();
};
```

Cette classe n'apporte rien à l'interface publique de Synchronous. Dans la partie protégée de l'interface, elle définit la méthode subname qui permet aux classes de modules composés de construire le nom de leurs composants.

Le constructeur sans arguments Composite() est nécessaire pour l'héritage multiple.

IX.1.4 Le retard

La classe Delay implémente le retard tel qu'il a été défini dans le modèle. Les instances de cette classe ont un signal d'entrée et un signal de sortie qui prend à chaque instant la valeur qu'avait le signal d'entrée à l'instant précédent. Il est possible de spécifier la valeur initiale du retard, c'est-à-dire la valeur du premier échantillon de son signal de sortie.

```
template<class sigType>
  class Delay : virtual public Synchronous {
    public:
      Delay(const char* name, const boolean sched = true,
            const sigType* initValue = 0);
      InputSignal** inSigList();
      virtual InSignal<sigType>& input();
      virtual OutSignal<sigType>& output();
      virtual void synchronize();
      virtual void endInstant();
    protected:
      void resetOutputs();
      void setInputs();
      void activate();
    private:
      InSignal<sigType> fInput;
      OutSignal<sigType> fOutput;
      Sample<sigType> lastValue;
      Event delayReady;
};
```

Delay implémente le retard pour n'importe quel type de signal, sa définition est donc un gabarit de classe, le type des valeurs véhiculées par le signal étant un type muet. Son constructeur prend un argument supplémentaire : initValue qui est la valeur initiale du retard. Si cet argument n'est pas fourni, la valeur initiale n'est pas définie et le signal de sortie du retard ne sera pas émis au premier instant.

La méthode inSigList() retourne ici une liste vide puisque le signal de sortie du retard ne dépend d'aucun signal à l'instant auquel il est émis.

La méthode input() rend une référence sur le signal d'entrée du retard, et la méthode output() en rend une sur son signal de sortie. Le fonctionnement particulier du retard nous oblige à redéfinir les méthodes synchronize() et endInstant(). En effet, synchronize() est appelée lorsqu'on cherche à obtenir la valeur du signal de sortie du retard. Le comportement hérité de Synchronous pour cette méthode consiste à se synchroniser avec le contrôleur, c'est-à-dire à attendre que l'objet ait fini de réagir. Dans le cas du retard, il faut simplement attendre que la valeur du signal d'entrée à l'instant précédent ait été recopiée dans le signal de sortie.

La méthode endInstant(), qui n'est pas utilisée par les objets synchrones, est ici utilisée pour déterminer quand la valeur du signal de sortie n'a plus besoin d'être valide.

La méthode resetOutputs() est habituellement utilisée pour initialiser les signaux de sortie à l'état non émis, la méthode de réaction ne modifiant que les signaux émis. Ici, il faut donner au signal de sortie la valeur qu'avait le signal d'entrée à l'instant précédent, la méthode de réaction ne faisant que mémoriser la valeur du signal d'entrée.

setInputs() et activate() ont leur rôle habituel : la première cherche à obtenir la valeur du signal d'entrée, ce qui peut être bloquant si cette valeur n'est

pas encore disponible, et la seconde fait réagir le retard, ce qui revient ici à mémoriser la valeur obtenue pour le signal d'entrée.

La partie privée de l'interface contient le schéma d'instance qui se compose du signal d'entrée, du signal de sortie, de l'échantillon contenant la valeur du signal d'entrée à l'instant précédent, et d'un événement permettant de savoir si le signal de sortie a pris sa valeur. C'est cet événement qu'utilise la méthode synchronize().

IX.1.5 Les classes d'interface

Les classes d'interface décrivent les objets qui permettent à une horloge de communiquer avec le monde asynchrone. Nous présentons ici les trois classes d'interface qui ont été développée dans la bibliothèque synchrone pour accéder aux fichiers et à l'horloge du système.

D'autres classes ont été développées pour des besoins spécifiques (interfaces d'une carte d'entrée/sortie pour notre machine temps-réél), et l'utilisateur de notre système sera amené à développer les classes d'interface propres à ses besoins.

Les classes d'interface fournies dans la bibliothèque synchrone sont celles qui correspondent à des besoins standard. Parmi celles que nous comptons implémenter lorsque nous adopterons le nouveau modèle d' exécution se trouvent :

- Interface avec un processus asynchrone
- Interface entre deux signaux de deux horloges distinctes (en mode « historique » et en mode « valeur récente »[2])
- Interface avec un protocole réseau pour permettre le développement d'applications distribuées à gros grain.

InputModule

Cette classe permet d'accéder en lecture au contenu d'un fichier. Elle a été spécifiquement prévue pour fournir une interface pratique lors de la simulation du comportement d'objets synchrones.

```
template<class sigType>
  class InputModule : virtual public Synchronous {
    public:
      InputModule(const char* name, const boolean sched =true,
                  const char* prompt = "", const char* tail = "");
      virtual void setStreams(istream& i, ostream& o = cout,
                              ostream& log = nullDev);
      InputSignal** inSigList();
      virtual OutSignal<sigType>& input();
```

[2]Voir à ce sujet le paragraphe « Communications entre une horloge et le monde extérieur » dans le modèle d'exécution.

```
    protected:
      void resetOutputs();
      void setInputs();
      void activate();
    private:
      istream* fInputStream;
      ostream* fOutputStream;
      ostream* fLogStream;
      const char* fPrompt;
      const char* fTail;
      OutSignal<sigType> fInput;
};
```

Le constructeur prend, en plus des arguments habituels, deux chaînes de caractères : prompt et tail. prompt est affichée sur le fichier de sortie lorsqu'une valeur est requise sur le fichier d'entrée. tail est affichée sur le fichier de sortie lorsqu'une valeur a été lue sur le fichier d'entrée. Nous avons donc un couplage entre le fichier d'entrée sur lequel les valeurs sont lues, et un fichier de sortie qui permet d'informer l'utilisateur de ce qui se passe dans le cas notamment d'une simulation.

La méthode setStreams permet de choisir ces fichiers. Dans ses arguments, i est le fichier sur lequel sont lues les valeurs du signal, o est le fichier sur lequel sont affichées les informations pour l'utilisateur, d'où sa valeur par défaut cout (qui correspond à la console), et log est le fichier sur lequel sera affiché l'historique de la session sous la forme "prompt <valeur lue> tail".

Par défaut, le fichier d'historique n'existe pas, ce qui correspond à la valeur nullDev (/dev/null sous Unix).

inSigList() retourne ici une liste vide puisque les instances d'InputModule n'ont pas de signal d'entrée.

La méthode input() donne une référence sur le signal de sortie de l'objet. La valeur de ce signal est en effet une entrée de l'horloge puisqu'il s'agit d'information allant du monde extérieur vers les objets de l'horloge (voir la définition de l'horloge dans le modèle pour une discussion sur la différence entre « entrée » et « signal d'entrée »).

resetOutputs() a le comportement standard. setInputs() et activate() ont un comportement un peu particulier. activate() ne fait rien, la valeur du signal de sortie étant obtenue dans setInputs(). C'est en effet dans le corps de cette méthode que l'objet lit le fichier d'entrée. Cette lecture est l'équivalent de la mise à jour des signaux d'entrée pour un objet synchrone, et peut être bloquante si le fichier d'entrée est la console (cin en C++) et que l'utilisateur n'a pas encore tapé de valeur.

Notons que la transformation de la chaîne de caractère lue sur ce fichier en valeur de type sigType se fait grâce à l'opérateur d' extraction << de C++, qui doit donc avoir été défini pour ce type. Le caractère '!' placé avant la valeur est interprété par l'opérateur d'extraction défini dans Sample comme indiquant l'absence de signal.

La partie privée de l'interface décrit le schéma d'instance qui comprend un pointeur sur le fichier d'entrée, un pointeur sur le fichier de sortie (pour prompt et tail), un pointeur sur le fichier d'historique de la session, et le signal de sortie de l'objet d'interface.

OutputModule

Cette classe est similaire à InputModule, mais écrit la valeur d'un signal dans un fichier.

```
template<class sigType>
  class OutputModule : virtual public Synchronous {
    public:
      OutputModule(const char* name,const boolean sched =true,
                   const char* banner = "",
                   const boolean always = true,
                   const char* tail = "\n");
      virtual void setStreams(ostream& o,
                              ostream& l = nullDev);
      InputSignal** inSigList();
      virtual InSignal<sigType>& probe();
    protected:
      void resetOutputs();
      void setInputs();
      void activate();
    private:
      ostream* fOutputStream;
      ostream* fLogStream;
      const char* fBanner;
      const char* fTail;
      const boolean fAlways;
      InSignal<sigType> fProbe;
};
```

Le paramètre banner du constructeur est la chaîne de caractères qui sera affichée avant la valeur du signal, et le paramètre tail, celle qui sera affichée après. La valeur par défaut "\n" correspond à un passage à la ligne après chaque écriture d'une valeur.

Le paramètre always indique si la valeur du signal doit être écrite dans le fichier à chaque instant ou seulement quand le signal est émis.

setStreams permet de choisir le fichier dans lequel les valeurs sont écrites, ainsi qu'un fichier d'historique.

inSigList() retourne ici une liste contenant le signal d'entrée probe de l'objet. On obtient une référence sur ce signal par la méthode probe().

Les méthodes resetOutputs(), setInputs() et activate() ont le comportement habituel, l'écriture dans le fichier se faisant dans activate().

La partie privée de l'interface contient le schéma d'instance.

TimeModule

Cette classe fournit une interface avec l'horloge du système :

```
class TimeModule : virtual public Synchronous {
  public:
    TimeModule(const char* name, const boolean sched = true);
    InputSignal** inSigList();
    virtual OutSignal<int>& milliSecond();
    virtual OutSignal<int>& centiSecond();
    virtual OutSignal<int>& deciSecond();
    virtual OutSignal<int>& second();
    virtual OutSignal<int>& minute();
    virtual OutSignal<int>& hour();
  protected:
    void resetOutputs();
    void setInputs();
    void activate();
  private:
    OutSignal<int> fMilliSecond;
    OutSignal<int> fCentiSecond;
    OutSignal<int> fDeciSecond;
    OutSignal<int> fSecond;
    OutSignal<int> fMinute;
    OutSignal<int> fHour;
    timeb fClock;
};
```

Les instances de TimeModule ont cinq signaux véhiculant des valeurs entières. Ces signaux donnent le nombre de millièmes de seconde, centièmes de seconde, dixièmes de seconde, secondes, minutes et heures à l'horloge du système.

Dans setInputs(), l'objet échantillonne l'horloge du système. Dans activate(), il calcule la nouvelle valeur des signaux et émet ceux qui ont changé. Comme il s'agit d'un échantillonnage de l'horloge système, il n'est pas garanti qu'un signal comme milliSecond soit émis à chaque milliseconde. Tout dépend de la fréquence d'échantillonnage. Par contre, quand hour est émis, tous les autres signaux sont émis puisque leur occurrence est par définition synchrone avec la sienne.

IX.1.6 Les contrôleurs

Les contrôleurs sont chargés de synchroniser les objets synchrones sur les instants définis par l'ordonnanceur. Quand un objet disparaît, son contrôleur passe dans un état spécial qui indique à l'ordonnanceur qu'il ne contrôle plus rien. L'ordonnanceur le supprime alors de sa liste. Cet état particulier des contrôleurs est appelé stub pour rappeler qu'il n'est plus que la souche (ou le chicot) d'une structure qui comprenait un objet synchrone.

Controller

Cette classe définit le protocole standard des contrôleurs qui sont par défaut des stubs :

```
class Controller {
```

```
  public:
    Controller();
    virtual ~Controller();
    virtual void synchronize();
    virtual boolean isStub();
    virtual void becomeStub();
};
```

La méthode synchronize() retourne immédiatement et la méthode isStub() retourne true puisque par défaut, un contrôleur ne contrôle rien. La méthode becomeStub() permet de transformer un contrôleur en stub. Ici, il n'y a rien à faire puisque le contrôleur est déjà un stub.

SynController

La classe SynController est celle des contrôleurs qui synchronisent un objet avec l'ordonnanceur. Ce ne sont donc plus des stubs par défaut.

```
class SynController : public Controller {
  public:
    SynController(Synchronous* module);
    virtual ~SynController();
    virtual boolean isStub();
    virtual void becomeStub();
    virtual void synchronize();
    virtual void tideUp();
    virtual void tideLow();
    virtual void kill();
  private:
    Synchronous *const theModule;
    Event outReady;
    Semaphore tic;
    Semaphore tac;
    Semaphore ready;
    boolean amIaStub;
    Thread<SynController*, int> myLife;
    static int manage(SynController* s);
};
```

Le constructeur prend une instance de Synchronous pour argument. Cet objet est celui que le contrôleur va prendre en charge. Le constructeur active la méthode synchronizeOn() de l'objet pour lui indiquer qu'il doit se synchroniser sur le nouveau contrôleur, et crée un nouveau flot de contrôle (ou thread) pour exécuter la méthode statique manage de la classe SynController.

Cette méthode est la boucle d'activation de l'objet synchrone que gère le contrôleur. Elle commence par signaler le sémaphore ready qui indique que l'on est prêt pour un nouvel instant. Elle se met ensuite en attente du sémaphore tic. Ce dernier indique le début de l'instant. Quand il est signalé, la boucle d'activation fait réagir l'objet synchrone. Comme la réaction comprend la mise à jour des signaux d'entrée, cette étape peut être bloquante.

Lorsque la réaction est terminée, la boucle signale l'événement outReady dont la présence indique que les signaux de sortie de l'objet sont valides. Elle

se met ensuite en attente du sémaphore tac qui indique la fin de l'instant. Lorsque ce sémaphore est signalé, on remet l'événement outReady à l'état absent et on active la méthode endInstant() de l'objet pour lui indiquer que l'instant est terminé. Les sémaphores tic et tac sont signalés par les méthodes tideUp() et tideLow() qui sont activées par l'ordonnanceur. Avant de signaler tic, la méthode tideUp() attend le sémaphore ready qui indique que le contrôleur est prêt à débuter un nouvel instant.

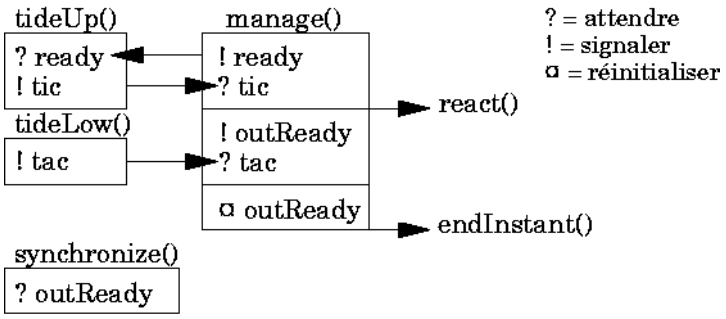

FIG. IX.2 – Interactions entre les méthodes d'un contrôleur

La figure IX.2 montre les relations entre les méthodes de SynController et ses événements et sémaphores, ainsi que le moment auquel est activée la méthode de réaction de l'objet contrôlé.

La méthode kill() arrête le flot de contrôle associé au contrôleur. Elle est utilisée par la méthode becomeStub() qui transforme le contrôleur en stub et doit donc tuer le thread associé. Elle est aussi utilisée par le destructeur qui doit s'assurer que le thread est bien mort, de façon à éviter qu'il ne cherche à accéder aux champs du contrôleur quand ce dernier n'existera plus.

La partie privée de l'interface contient le schéma d'instance qui comprend les événements et sémaphores utilisés pour synchroniser le contrôleur avec l'ordonnanceur et l'objet synchrone avec le contrôleur, ainsi qu'un booléen indiquant si le contrôleur est un stub ou pas.

Le champ myLife est un flot d'exécution pour une fonction prenant un pointeur sur SynController comme argument et retournant un entier. Cette fonction est en fait la méthode manage(), et c'est le constructeur de SynController qui crée myLife pour exécuter manage.

Le gabarit de classes Thread est défini dans la bibliothèque de threads qui n'est pas décrite ici car elle ne sert que d'interface portable aux appels de différents systèmes d'exploitation. Cette bibliothèque contient aussi la définition des classes Semaphore et Event qui implémentent respectivement les sémaphores et les évé nements en s'appuyant sur les mécanismes fournis par le système d' exploitation.

IX.1.7 L'ordonnanceur

La classe Scheduler implémente l'ordonnanceur qui est chargé de définir les instants de l'horloge et de faire réagir les objets synchrones dans un ordre compatible avec leurs dépendances. Il communique pour cela avec les contrôleurs des objets.

Dans l'implémentation actuelle, le traitement des requêtes de modification de l'horloge n'est pas implémenté. Il le sera en même temps que le nouveau modèle d'exécution.

```
class Scheduler {
  public:
    Scheduler(const char* name = 0, double period = 0);
    virtual ~Scheduler();
    virtual void manage(SynController* s);
    virtual void manage(Synchronous* s);
    virtual void start();
    virtual void join();
    virtual void stop();
  private:
    struct SLink {
      SynController *s;
      SLink* next;
    };
    Clock myClock;
    boolean iShouldStop;
    SLink* modList;
    Thread<Scheduler*, int> sched;
    static int schedProc(Scheduler* self);
};
```

Le constructeur prend comme argument une chaîne de caractère qui sera le nom du flot de contrôle exécutant le code de l'ordonnanceur, et un réel qui est la période de l'horloge, c'est-à-dire l'intervalle de temps, compté sur l'horloge du système, qui sépare deux instants.

Le code de l'ordonnanceur s'exécute dans un nouveau flot de contrôle afin de permettre à plusieurs ordonnanceurs de tourner sur le même processeur, ainsi que pour laisser le flot de contrôle principal exécuter le code non synchrone du programme.

Les deux méthodes manage() permettent d'ajouter un objet synchrone dans l'horloge, soit en passant le contrôleur de l'objet s'il existe, soit en passant l'objet s'il n'a pas encore de contrôleur. Dans ce dernier cas, un contrôleur sera automatiquement associé à l'objet.

La méthode start() fait démarrer l'ordonnanceur. En effet, le constructeur crée le flot de contrôle, mais n'autorise pas son exécution. Ceci permet d'initialiser et de configurer tous les ordonnanceurs d'une application avant de les laisser s'exécuter.

La méthode join() permet d'attendre que l'ordonnanceur s'arrête. Elle sera par exemple appelée par le flot de contrôle principal si ce dernier n'a rien d'autre à faire pendant que l'ordonnanceur tourne.

La méthode stop() permet d'arrêter l'ordonnanceur à la fin d'un instant de l'horloge. Cette méthode est appelée par la méthode killSched() de la classe Synchronous.

Dans la partie privée de l'interface, on trouve la définition de la structure SLink qui permet de chaîner les contrôleurs des objets de l'horloge. La liste de ces contrôleurs est pointée par le champ modList. Le champ myClock contient une horloge dont le temps de référence est celui du système, et qui bat à la fréquence correspondant à l'argument period du constructeur. L'ordonnanceur utilise cette horloge pour déterminer quand il doit commencer un nouvel instant. La classe Clock est définie dans la bibliothèque de threads.

Le booléen iShouldStop indique à l'ordonnanceur s'il doit s'arrêter à la fin de l'instant. Ce champ prend la valeur true quand la méthode stop() est activée. Le champ sched contient le flot de contrôle dans lequel s'exécute le code de l'ordonnanceur. Ce code est celui de la méthode schedProc() qui prend pour argument un pointeur sur l'ordonnanceur. L'exécution de cette méthode dans un nouveau flot de contrôle ne peut se faire que si l'on a un pointeur sur elle, ce qui impose qu'elle soit déclarée statique. Or une méthode statique n'a pas de référence implicite sur l'objet pour lequel elle est activée (elle peut même n'être activée pour aucun objet), il faut donc lui fournir explicitement cette référence dans notre cas, d'où l'argument de la méthode schedProc().

Le coeur de l'ordonnanceur est donc la méthode schedProc() qui active périodiquement les contrôleurs des objets de l'horloge. Un cycle débute par l'élimination des contrôleurs qui sont devenus des stubs, et qui étaient donc associés à des objets qui ont disparu de l'horloge. On active ensuite la méthode tideUp() de chacun des contrôleurs de la liste, puis on attend que les objets associés aient fini de réagir en se synchronisant avec chacun des contrôleurs. Cette phase revient à attendre que tous les événements outReady des contrôleurs soient signalés. On active ensuite la méthode tideLow() de chacun des contrôleurs pour leur indiquer la fin de l'instant. Il ne reste alors qu'à attendre le prochain top de l'horloge de l'ordonnanceur pour faire débuter un nouvel instant.

Pour traiter les requêtes de modification de l'horloge (instanciations et destruction d'objets synchrones et connexions de signaux), il faudrait ajouter deux méthodes au protocole de Scheduler : une méthode de requête de destruction et une méthode de requête de connexion. Les instanciations peuvent en effet se faire n'importe quand, l'ordonnanceur n'ajoutant le nouvel objet dans sa liste que juste avant le début d'un instant. Les requêtes seraient rangées dans deux listes, une pour les destructions et l'autre pour les connexions. Pour une destruction, il suffit de garder un pointeur sur l'objet à détruire, et pour une connexion, un pointeur sur chacun des deux signaux à connecter. Le contenu de ces listes serait traité dans la phase actuelle de nettoyage de la liste des contrôleurs.

IX.2 Classes créées par occ++

Pour présenter l'interface des classes créées par occ++, nous allons prendre un exemple qui présente toutes les caractéristiques que l'on peut trouver dans un module synchrone écrit en Esterel.

Le module screensave est un économiseur d'écran qui attend 2 minutes avant d'émettre le signal saveScreen avec la valeur true. A chaque fois que la souris bouge, le délai d'attente est remis à zéro et le signal screenSave est émis avec la valeur false. De plus, si la souris est amenée dans le coin supérieur gauche de l'écran, l'économiseur d'écran s'arrête.

Cet exemple permet d'illustrer l'utilisation de types, constantes, procédures et fonctions externes à Esterel, et montre comment les compteurs d'événements et les exceptions valuées apparaissent dans la classe générée.

```
module screensave:
  type Point;
  constant UpperLeft:Point;
  procedure killSched()();
  function And(boolean, boolean):boolean;

  input mouse(Point);
  input second;
  output saveScreen(combine boolean with And);

  loop
    trap mouseMove(integer) in
      every mouse do
        if ?mouse = UpperLeft then
          exit mouseMove(1)
        else
          exit mouseMove(0)
        end if
      end every
      ||
      await 120 second;
      emit saveScreen(true);
    handle mouseMove do
      if ??mouseMove = 1 then
        call killSched()();
      else
        emit saveScreen(false);
      end if
    end trap
  end loop
end module
```

La constante UpperLeft correspond au coin supérieur gauche de l'écran. La procédure killSched() est celle qui est définie dans la classe Synchronous. La fonction And() est utilisée pour combiner les valeurs du signal saveScreen lorsqu'il est émis plusieurs fois dans un instant (ce qui arrive lorsque l'on bouge la souris après exactement 2 minutes).

Le signal mouse donne la position de la souris. Il est émis à chaque fois

qu'elle bouge. Le signal second est émis toute les secondes et le signal saveScreen est émis avec la valeur true lorsque l'économiseur d'écran doit être activé, et avec la valeur false lorsque l'état normal de l'écran doit être restauré. Ses émissions multiples sont combinées par un ET logique, ce qui donne la priorité à la restauration de l'état normal de l'écran.

Dans le corps du module, on utilise une exception valuée pour signaler que la souris bouge dans le seul but de montrer comment elle apparaît dans la classe synchrone. Lorsque cette exception est levée avec la valeur 1, le programme doit s'arrêter. Lorsqu'elle est levée avec la valeur 0, il faut restaurer l'état normal de l'écran.

La commande `esterel -Lc++ screensave.strl` produit les fichiers screensave.H et screensave.C. Le contenu de screensave.H donne l'interface de la classe synchrone screensave :

```
//
//  screensave.H
//  Created by occ++ v3.2.7 on Thursday November 04 1993 at
//  10:09:17 from oc v3 source file oc15940.oc

#include <Esterel/libSync.H>

//
//  Interface of module screensave
//       root module: screensave, from file: screensave.strl

#include "screensaveDef.H"

class screensave : public Esterel {
  protected:
    enum {  // Automaton action numbers
      kSequence,
      kBranchAlways,
      kPresent_mouse_2,
      kPresent_second_3,
      kOutput_saveScreen_4,
      kIf_5,
      kCall__assign_integer_6,
      kCall__assign_integer_7,
      kCall__assign_integer_8,
      kDsz_9,
      kCombine_saveScreen_10,
      kIf_11,
      kCall_killSched_12,
      kCombine_saveScreen_13
    };
    static const Esterel::tStateNumber *const *const cscreensaveStates;

    static const Point UpperLeft;

    void resetOutputs();
    void setInputs();
    void doAction(tActionNumber op, Esterel::tActionNumber *i);

    InSignal<Point> fmouse;
```

```
    PureInSignal fsecond;
    OutSignal<boolean> fsaveScreen;
// Variables of module screensave
    integer valExc_mouseMove_5;  // Valued exception.
    integer _compteur_1;

  public:
    virtual InSignal<Point>& mouse();
    virtual PureInSignal& second();
    virtual OutSignal<boolean>& saveScreen();
    virtual InputSignal** inSigList();  // Returns a null
                      // terminated list of input signals
    screensave(const char* name,
               const boolean schedule = true); // Constructor
};
```

Le module faisant appel à des définitions externes (types et fonctions qui ne sont pas prédéfinis en Esterel), le fichier screensaveDef.H est automatiquement inclus en plus du fichier d'interface de la bibliothèque synchrone libSync.H. Ce fichier est censé être fourni par le programmeur et doit contenir les déclarations des entités externes à la classe.

Dans notre cas, les entités externes du module sont : le type Point, la constante UpperLeft, la procédure killSched() et la fonction And(). Les constantes sont transformées en constantes de la classe par occ++, UpperLeft est donc déclarée dans l'interface de screensave, il faudra donc simplement déclarer sa valeur dans un fichier C++.

killSched() est une méthode héritée de Synchronous à travers Esterel, elle ne doit donc pas être redéclarée. Les seules entités externes à la classe screensave sont donc le type Point et la fonction And(). Ils doivent donc être déclarés dans screensaveDef.H.

Examinons maintenant la déclaration de la classe screensave. On voit tout d'abord qu'elle hérite bien de la classe Esterel qui donne le comportement et la structure des objets synchrones qui utilisent un automate pour réagir.

La partie protégée de l'interface contient tout ce qui touche à la mécanique interne du module. On y trouve les numéros des actions sous la forme d'une énumération, ce qui les rend plus lisibles. Les deux premières actions sont présentes dans toutes les classes synchrones produites par occ++ : kSequence correspond à l'action vide, et kBranchAlways correspond au branchement inconditionnel. Les autres actions sont particulières au module screensave, le nom de l'action étant postfixé par son numéro afin d'obtenir des identificateurs uniques.

On trouve ensuite la déclaration de la table des états de l'automate sous le nom cscreensaveStates, puis la constante UpperLeft dont il faudra définir la valeur comme nous l'avons vu. Les méthodes resetOutputs() et doAction() sont définies (resetOutputs() est une virtuelle pure de Synchronous, et doAction() est une virtuelle pure de Esterel) car on connait maintenant les signaux, et on sait quel comportement associer à chaque action.

Suivent les déclarations des signaux : fmouse, signal d'entrée de type Point correspond au signal mouse du module ; fsecond, signal d'entrée pur correspond au signal second et fsaveScreen, signal de sortie de type booléen correspond à savescreen.

On trouve ensuite les variables du module : valExc_mouseMove_5 contient la valeur de l'exception mouseMove (une exception non valuée n'aurait pas utilisé de variable, son émission aurait été codée dans l'automate) et _compteur_1 contient la valeur du compteur d'occurrence associé à l'expression await 120 second.

La partie publique est beaucoup moins chargée. On y trouve les trois méthodes d'accès aux signaux des instances. Ces méthodes portent simplement le nom du signal auquel elles permettent d'accéder. inSigList() retourne la liste des deux signaux d'entrée mouse et second, et le constructeur screensave() permet de créer une instance de la classe qui aura un nom et sera par défaut placée dans l'horloge courante.

Avant de pouvoir instancier cette classe, nous devons compléter sa définition par le fichier screensaveDef.H qui doit, comme nous l'avons vu, définir le type Point et la fonction And() :

```
#include "Point.H"

boolean And(boolean a, boolean b);
```

La définition de la classe Point se trouve dans le fichier Point.H qui est donc inclus, et la fonction And() prend deux arguments booléens et retourne un booléen, comme cela avait été déclaré en Esterel. Le fichier screensaveDef.C contient le code associé à cette déclaration, ainsi que la valeur de la constante UpperLeft :

```
#include "screensave.H"

const Point screensave::UpperLeft = {0,0};

boolean And(boolean a, boolean b) {
  return boolean(a&&b);
}
```

UpperLeft est donc définie comme étant le point de coordonnées (0,0) et la fonction And() retourne bien le ET logique de ces arguments.

A titre d'exemple, et pour montrer comment sont codées les actions, nous donnons le contenu de screensave.C, certains passages étant supprimés :

```
      ...
#include "screensave.H"
      ...
static const Esterel::tStateNumber cscreensaveState0ops[] = {
  0,0
};
      ...
```

```
static const Esterel::tStateNumber cscreensaveState3ops[] = {
  2,27,5,13,6,11,5,12,8,0,2,13,8,4,0,2,7,11,5,
      ...
};

static const Esterel::tStateNumber *const cscreensaveStateOps[] = {
  cscreensaveState0ops,
  cscreensaveState1ops,
  cscreensaveState2ops,
  cscreensaveState3ops
};

// Table of automaton action numbers
const Esterel::tStateNumber *const *const
    screensave::cscreensaveStates = cscreensaveStateOps;

InSignal<Point>& screensave::mouse() {
  return fmouse;
}
      ...

void screensave::resetOutputs() {
  fsaveScreen.value().reset();
}

void screensave::setInputs() {
  fmouse.update();
  fsecond.update();
}

InputSignal** screensave::inSigList() {
  InputSignal** inList;
  inList = new (InputSignal*[3]);
  inList[0] = &fmouse;
  inList[1] = &fsecond;
  inList[2] = 0;
  return inList;
}

screensave::screensave(const char* name,
                       const boolean schedule)    // Constructor
          : Synchronous(name, schedule),
            Esterel(cscreensaveStates),
            fmouse(this, "mouse"),
            fsecond(this, "second"),
            fsaveScreen(this, "saveScreen")
{
  fState = 1;    // Set initial state
}

void screensave::doAction(tActionNumber _op,
                          Esterel::tActionNumber *_i) {
  switch (_op) {
    case kSequence:
      break;
    case kBranchAlways:
      branchIfFalse(false,_i);
```

```
      break;
    case kPresent_mouse_2:
      branchIfFalse(fmouse.value().isPresent(), _i);
      break;
        ...
    case kIf_5:
      branchIfFalse(boolean(
              (fmouse.value().value() == UpperLeft)), _i);
      break;
    case kCall__assign_integer_6:
      valExc_mouseMove_5 = 1;
      break;
        ...
    case kDsz_9:
      branchIfFalse(boolean(-- _compteur_1 <= 0), _i);
      break;
    case kCombine_saveScreen_10:
      if (fsaveScreen.value().isPresent())
        fsaveScreen.value() =
                  And(fsaveScreen.value().value(),true);
      else
        fsaveScreen.value() = true;
      break;
    case kIf_11:
      branchIfFalse(boolean((valExc_mouseMove_5 == 1)), _i);
      break;
    case kCall_killSched_12:
      killSched();
      break;
        ...
  }
}
```

On remarque que dans le constructeur de screensave, les signaux sont construits en leur passant un pointeur sur l'objet synchrone (this) et leur nom. C'est ainsi que les signaux peuvent connaître l'objet synchrone auquel ils appartiennent.

Les expressions du type signal.value().value() peuvent surprendre à première vue. Le premier appel à la méthode value() est traité par le signal qui donne en retour son échantillon. Le deuxième appel à value() est adressé à cet échantillon qui donne alors sa valeur. On notera aussi que l'émission du signal saveScreen se fait (dans le traitement des actions kCombine_saveScreen_x) en affectant une valeur booléenne à son échantillon. On pourra se référer à la présentation de l'opérateur d'affectation qui a été faite lors de la description de la classe Sample.

IX.3 Classes créées par mdlc

Pour présenter l'interface des classes créées par mdlc, nous allons reprendre l'exemple du module frein_ABS que nous avons déjà utilisé lors de la présentation de mdlc au chapitre « Outils de Développement ».

Nous allons définir frein_ABS de deux manières, tout d'abord en tant que module composé d'un frein et d'un filtre_ABS, puis en tant que module dérivant de frein par l'utilisation d'un filtre_ABS.

IX.3.1 Classes de modules composés

Rappelons l'interface des modules frein et filtre_ABS :

```
#source "frein.strl"
#c++ "frein.H"

frein {
  input: integer commande;
  output: integer pression;
}
#source "filtre_ABS.strl"
#c++ "filtre_ABS.H"

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

On définit alors frein_ABS en tant que module composé par :

```
frein_ABS : {
  input: glissement, integer commande;
  output: integer pression;

  frein freinStandard;
  filtre_ABS filtre;

  freinStandard.commande = commande;
  filtre.glissement = glissement;
  filtre.pression_brute << freinStandard.pression;
  pression = filtre.pression_filtree;
}
```

Ce module regroupe un module frein (freinStandard) et un module filtre_ABS (filtre) pour obtenir son comportement de frein_ABS. L'entrée commande du frein ABS est celle du frein, l'entrée glissement est celle du filtre, l'entrée pression_brute du filtre est connectée à la sortie du frein standard et la sortie du frein ABS est celle du filtre.

Après compilation de ces fichiers par mdlc, on obtient les fichiers frein_ABS.H et frein_ABS.C qui contiennent respectivement l'interface et le code de la classe frein_ABS. L'interface est la suivante :

```
class frein_ABS : public Composite {
  private:
    frein freinStandard;
    filtre_ABS filtre;
  protected:
    void resetOutputs();
    void setInputs();
```

```
    void activate();
  public:
    virtual PureInSignal& glissement();
    virtual InSignal<integer>& commande();
    virtual OutSignal<integer>& pression();
    virtual InputSignal** inSigList();

    frein_ABS(const char* name, const boolean schedule = true);
};
```

Le module frein_ABS étant composé, la classe frein_ABS dérive de la classe Composite qui donne le comportement et la structure des objets synchrones qui utilisent d'autres objets synchrones pour réagir. Dans la partie privée de l'interface, on trouve les instances de frein et de filtre_ABS qui sont utilisées par le frein ABS.

Cette interface ne fait pas apparaître de variables pour les signaux, ces derniers sont en effet déjà présents dans les instances de frein et de filtre_ABS. Le reste est similaire à ce que l'on trouve dans l'interface d'une classe générée par occ++, mis à part la déclaration de la méthode activate() qui n'a bien entendu pas le même comportement que celle des objets synchrones à automate.

A titre d'exemple, et pour indiquer comment se contruisent et ré agissent les instances de frein_ABS, nous donnons le contenu de frein_ABS.C :

```
    ...
//  Code of module frein_ABS
//

void frein_ABS::resetOutputs() {
  pression().value().reset();
}

void frein_ABS::setInputs() {
}

void frein_ABS::activate() {
  freinStandard.react();
  filtre.react();
}

PureInSignal& frein_ABS::glissement() {
  return filtre.glissement();
}

InSignal<integer>& frein_ABS::commande() {
  return freinStandard.commande();
}

OutSignal<integer>& frein_ABS::pression() {
  return filtre.pression_filtree();
}

InputSignal** frein_ABS::inSigList() {
  InputSignal** inList;
```

```
  inList = new (InputSignal*[3]);
  inList[0] = &glissement();
  inList[1] = &commande();
  inList[2] = 0;
  return inList;
}

frein_ABS::frein_ABS(const char* name, const boolean schedule)
        : Synchronous(name, schedule),
          freinStandard(subname(name,"freinStandard"), false),
          filtre(subname(name,"filtre"), false) {
  freinStandard.commande().chown(this);
  filtre.glissement().chown(this);
  filtre.pression_brute() << freinStandard.pression();
  filtre.pression_filtree().chown(this);
}
```

On remarque que la méthode setInputs() ne fait rien. Les signaux d'entrée sont en effet mis à jour par les instances de frein et de filtre_ABS auxquelles ils appartiennent. La méthode activate() fait réagir le frein, puis le filtre. Mdlc a en effet déterminé lors de la compilation que le filtre dépend du frein.

Les effets de l'identification des signaux par l'opérateur = de mdlc se traduisent dans les méthodes d'accès aux signaux : le signal retourné par la méthode commande() est celui du frein standard par exemple. Ces mêmes effets se traduisent dans le constructeur par l'appel de la méthode chown() qui permet de donner un nouveau propriétaire à un signal. On remarquera l'utilisation de la méthode subname() de Composite pour créer les noms de freinStandard et de filtre.

IX.3.2 Classes de modules dérivés

Nous définissons maintenant frein_ABS en tant que module dérivé de frein :

```
frein_ABS : frein {
  input: glissement;
  filtre_ABS filtre;

  filtre.pression_brute << super.pression;
  filtre.glissement = glissement;
  pression = filtre.pression_filtree;
}
```

Ainsi définit, frein_ABS utilise un filtre_ABS et un signal supplémentaire : glissement pour raffiner le comportement de frein. La compilation de cette définition par mdlc produit la classe frein_ABS dont l'interface est la suivante :

```
class frein_ABS : public frein, public Composite {
  private:
    filtre_ABS filtre;
  protected:
    void resetOutputs();
    void setInputs();
```

```
    void activate();
  public:
    virtual PureInSignal& glissement();
    virtual OutSignal<integer>& pression();
    virtual InputSignal** inSigList();

    frein_ABS(const char* name, const boolean schedule = true);
};
```

Cette fois, la classe frein_ABS dérive de frein et de Composite puisqu'elle correspond à un module composé (il utilise filtre_ABS pour réagir) dérivé de frein.

La partie privée de l'interface contient l'instance de filtre_ABS utilisée par le frein ABS. Le signal glissement n'y apparaît pas puisqu'il est apporté par le filtre. Pour le reste, on retrouve l'interface de la classe frein_ABS obtenue par composition de modules.

Il est intéressant de voir comment les instances de frein_ABS s'initialisent et réagissent, nous donnons donc un extrait de frein_ABS.C :

```
void frein_ABS::activate() {
  frein::resetOutputs();
  frein::setInputs();
  frein::activate();
  filtre.react();
}

PureInSignal& frein_ABS::glissement() {
  return filtre.glissement();
}

OutSignal<integer>& frein_ABS::pression() {
  return filtre.pression_filtree();
}

frein_ABS::frein_ABS(const char* name, const boolean schedule)
        : Synchronous(name, schedule),
          frein(name, schedule),
          filtre(subname(name,"filtre"), false) {
  filtre.glissement().chown(this);
  filtre.pression_brute() << frein::pression();
  filtre.pression_filtree().chown(this);
}
```

On remarque que la méthode activate() fait appel aux méthodes de réaction de la classe frein. On ne peut pas se contenter d'invoquer la méthode react() de la classe frein, car cette dernière, qui est définie dans la classe Synchronous, ferait appel à la méthode activate() de frein_ABS, amorçant ainsi une récursion sans fin.

On note aussi que cette méthode fait réagir le filtre après le frein, ce qui correspond à l'ordre de dépendance déterminé par mdlc lors de la compilation. Seules les méthodes d'accès aux signaux glissement et pression sont redéfinies. Le signal commande étant celui de la superclasse, la méthode d'accès héritée convient à frein_ABS.

Dans le constructeur, on notera l'utilisation de la méthode d'accès au signal pression de la superclasse pour réaliser la connexion de l'entrée du filtre à la sortie du module hérité. La méthode pression() de frein_ABS retourne en effet le signal de sortie du filtre puisque le signal pression a été identifié au signal pression_filtree de ce filtre.

X Exemple d'utilisation d'objets synchrones

Nous donnons ici un exemple d'utilisation d'objets synchrones dans une application. Cet exemple très simple permet de montrer comment instancier un objet synchrone dans une horloge et le connecter à d'autres objets synchrones.

Pour cet exemple, nous utilisons un module Esterel très simple qui calcule la différence entre ses deux entrées :

```
module diff:
  input a(integer);
  input b(integer);
  output s(integer);

  loop
    await
      case a
      case b
    end await;
    emit s(?a - ?b)
  end loop
end module
```

Le programme suivant utilise ce module, les entrées et les sorties se faisant sur la console grâce à des objets d'interface :

```
#include "diff.H"

int main() {
  theScheduler = new Scheduler("Ordonnanceur",0.1);
  diff soustracteur("soustracteur");
  InputModule<int> A("entree a",true,"a:");
  InputModule<int> B("entree b",true,"b:");
  OutputModule<int> S("sortie",true,"s=");

  soustracteur.a() << A.input();
  soustracteur.b() << B.input();
  S.probe() << soustracteur.s();

  theScheduler->start();
  theScheduler->join();

  return 0;
}
```

La variable globale theScheduler pointe sur l'ordonnanceur de l'horloge courante. On commence donc par créer un ordonnanceur de nom « Ordonnanceur » et de période d'activation 0,1 seconde. Les objets synchrones instanciés par la suite seront créés dans l'horloge de cet ordonnanceur.

L'objet soustracteur, instance de diff, et les objets d'interface A, B et C, instances d'InputModule<int> et d'OutputModule<int> sont donc tous sur la même horloge. On connecte ensuite le signal a du soustracteur à la sortie de

l'objet A, le signal b à la sortie de l'objet B, et l'entrée de l'objet d'interface C à la sortie s du soustracteur. Ces connexions sont possibles puisque les types des signaux sont identiques. La configuration de l'horloge ainsi établie sera celle du premier instant puisque l'ordonnanceur n'a pas été lancé. Comme ces objets ne modifient pas la topologie de leur horloge, cette configuration sera celle de tous les instants de l'horloge. Nous avons donc bien affaire à un système statique.

On lance ensuite l'ordonnanceur, et comme nous n'avons aucune tâche asynchrone à effectuer, nous nous mettons immédiatement en attente de son arrêt. Comme rien n'est prévu pour arrêter l'ordonnanceur dans les objets que nous utilisons, l'instruction return 0 ne sera jamais exécutée. L'exécution de ce programme donne :

```
a:1
b:2
s=#! 0
a:1
b:2
s=-1
a:!1
b:3
s=-2
a:!1
b:!3
s=#! -2
a:10
b:5
s=5
a:7
b:!5
s=2
a:^C
```

Le premier cycle correspond à l'initialisation de l'automate et ne produit donc aucun résultat. Les caractères '#' et/ou '!' indiquent qu'un signal n'est pas émis. La sortie s est bien égale à la différence a - b et n'est émise que lorsque l'un des signaux a ou b est émis.

Si nous introduisons maintenant une boucle dans les dépendances entre objets synchrones grâce au programme suivant :

```
#include "diff.H"

int main() {
  theScheduler = new Scheduler("Ordonnanceur",0.1);
  diff soustrac1("soustrac1");
  diff soustrac2("soustrac2");
  InputModule<int> A("entree a",true,"a:");
  InputModule<int> B("entree b",true,"b:");
  OutputModule<int> S("sortie",true,"s=");

  soustrac1.a() << A.input();
  soustrac1.b() << soustrac2.s();
  soustrac2.a() << soustrac1.s();  // Closing loop here
  soustrac2.b() << B.input();
  S.probe() << soustrac1.s();
```

```
  theScheduler->start();
  theScheduler->join();

  return 0;
}
```

La compilation se passe normalement puisque les vérifications statiques ne détectent pas les boucles. Par contre à l'exécution, on obtient :

```
# Loop detected while attempting to connect soustrac2.a to soustrac1.s
## Connection aborted !
```

Le signal a du soustracteur soustrac2 a détecté que la connexion qu'on lui demande de faire provoque la fermeture d'une boucle. Cette connexion est donc refusée. Le système étant statique, nous aurions pu le construire avec mdlc grâce au fichier mdl suivant :

```
loop : {
  input: integer a, integer b;
  output: integer s;

  diff soustrac1;
  diff soustrac2;

  soustrac1.a = a;
  soustrac1.b << soustrac2.s;
  soustrac2.a << soustrac1.s;  // Closing loop here
  s = soustrac1.s;
}
```

Mdlc refuse de compiler ce fichier en indiquant :

```
#  ### Error : Dependency loop in submodules.
#  soustrac2.a << soustrac1.s
#  soustrac1.b << soustrac2.s
### mdlc - Fatal : mdl source error
 # Loop detected in dependency graph of submodules
```

Mdlc nous indique le chemin qui, partant de soustrac2, a permis de revenir à soustrac2, fermant ainsi la boucle. Il faut savoir que mdlc n'indique que la première boucle détectée, il ne suffit donc pas toujours de supprimer une des connexions pour supprimer la boucle.

On voit ici l'avantage qu'il y a à utiliser mdlc pour construire des systèmes statiques : toutes les vérifications sont faites à la compilation. On peut ainsi être sûr qu'aucune erreur de ce type ne peut se produire à l'exécution.

XI Exemple de communications asynchrones

Pour cet exemple, nous avons développé deux classes d'interface pour la communication asynchrone entre horloges. Ces classes ne sont pas encore intégrées à la bibliothèque synchrone.

Il s'agit de simuler le comportement de quatre tortues qui se suivent, la première suivant la deuxième, et ainsi de suite, la quatrième suivant la première. Le module Esterel suivant donne le comportement d'une tortue :

```
module tortue:
  type Point;
  function nouvPos(Point, Point):Point;
  function dist(Point, Point):integer;
  constant iPos:Point;
  input cible(Point);
  output pos(Point);

  var npos:=iPos:Point in
    emit pos(npos);
    every tick do
      if (dist(npos, ?cible) > 1) then
        npos := nouvPos(npos, ?cible);
        emit pos(npos);
      end if
    end every
  end
end module
```

A chaque instant, la tortue calcule la distance qui la sépare de sa cible, et si cette distance est supérieure à 1, elle calcule sa nouvelle position et l'émet. Une tortue ne peut se déplacer que d'une unité de distance à la fois dans chaque direction.

Dans un premier temps, nous allons instancier ces quatre tortues sur une même horloge avec des communications synchrones. Comme il y a une boucle de dépendance entre ces tortues, nous les faisons communiquer à travers un retard, ce qui permet de casser la boucle. Afin d'obtenir un résultat graphique, nous avons construit une classe d'interface tortueX qui permet d'afficher les valeurs successives d' un signal de type Point dans une fenêtre XWindow.

On obtient la classe tortue par la commande `esterel -Lc++ :"-param" tortue.strl`. L'option -param indique que les constantes du module tortue sont des paramètres de l'instanciation au lieu d'être des constantes de la classe. Les fonctions nouvPos et dist sont définies dans le fichier "tortueDef.C" contenant les définitions auxiliaires de la classe tortue.

L'instanciation de quatre tortues sur une même horloge se fait comme suit :

```
#include "tortue.H"
#include "tortueX.H"
int main() {
  tortueConsts tConsts;
  theScheduler = new Scheduler("Ordonnanceur", 0.1);
      ...                        // Créer la fenêtre etc...
  tConsts.iPos = Point(0,0);  // Position initiale (0,0)
  tortue t1("t1", tConsts);   // pour la tortue t1
  tortueX xt1("xt1",w,...);   // xt1 affiche dans la fenêtre w
  Delay<Point> d1("d1", &(tConsts.iPos));
                              // Retard pour casser la boucle
  xt1.probe() << t1.pos();    // xt1 affiche la position de t1
  t1.cible() << d1.output();  // cible obtenue par un retard
      ...                        // Idem pour t2, t3 et t4
  d1.input() << t2.pos();     // cible de t1 = t2
      ...                        // idem pour t2 et t3
  d4.input() << t1.pos();     // cible de t4 = t1
  theScheduler->start();      // Démarrage de l'horloge
      ...                        // Gestion de l'affichage
  theScheduler->join();       // Attendre la fin
      ...                        // Fermer la fenêtre etc...
}
```

Ceci correspond au schéma de la figure XI.1 où les carrés représentent les retards, les cercles 1, 2, 3 et 4 les tortues, et les cercles X les objets d'interface chargés de l'affichage.

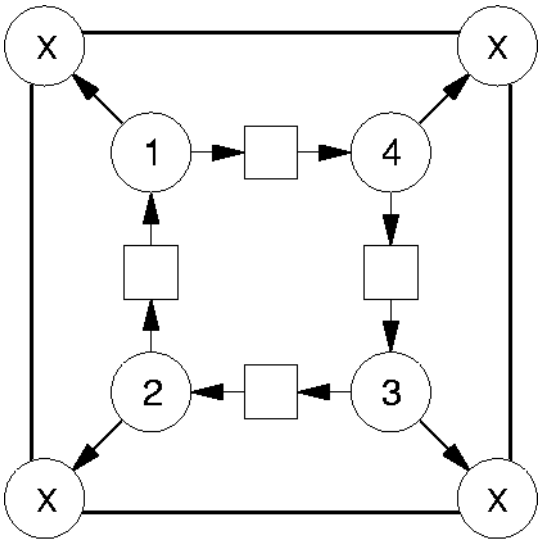

FIG. XI.1 – Quatre tortues sur une même horloge

L'exécution de ce programme donne le résultat de la figure XI.2 : les quatre tortues convergent au centre de la figure et leurs trajectoires sont symétriques. Plusieurs exécutions de ce programme donnent toujours le même résultat puisque le comportement des tortues est synchrone, donc déterministe.

Nous allons maintenant instancier chaque tortue sur une horloge distincte. Les communications entre tortues seront donc asynchrones et se feront grâce à des instances d'asyncEmit et d'asyncRecept, respectivement

FIG. XI.2 – Trajectoires des tortues en communication synchrone

émetteur asynchrone et récepteur asynchrone. La période d'activation de chaque horloge pourra être donnée sur la ligne de commande :

```
#include "tortue.H"
#include "tortueX.H"
int main(int argc, char** argv) {
      ...                          // Récupération des périodes
  Scheduler ord1("h1",per[0]);     // Premier ordonnanceur
      ...                          // Idem pour les 3 autres
  theScheduler = &ord1;            // Sélection de la 1ere horloge
  tConsts.iPos = Point(0,0);       // Position initiale de t1
  tortue t1("t1",tConsts);         // Création de t1 sur ord1
  tortueX xt1("xt1",w,...);        // Affichage sur la fenêtre w
  asyncRecept ar1("ar1");          // Récepteur asynchrone
  asyncEmit("ae1");                // Emetteur asynchrone
  xt1.probe() << t1.pos();         // xt1 affiche la position de t1
  t1.cible() << ar1.output();      // cible obtenue par le récepteur
  ae1.input() << t1.pos();         // position émise sur l'émetteur
  theScheduler = &ord2;            // Idem pour les autres tortues
      ...                          // chacune sur son horloge
  ar1.getFrom(&ae2);               // Accord de ar1 sur ae2 car t1
      ...                          // suit t2, idem pour les autres.
  ord1.start();                    // démarrage des horloges
      ...
  ord4.start();
      ...                          // Gestion de l'affichage
  ord1.join();                     // Attendre la fin des 4 horloges
      ...
  ord4.join().
      ...                          // Fermer la fenêtre etc...
}
```

Ce qui correspond au schéma de la figure XI.3 sur lequel les flèches ondulées représentent les communications asynchrones entre les horloges :

On obtient alors les trajectoires de la figure XI.4 dans le cas où les com-

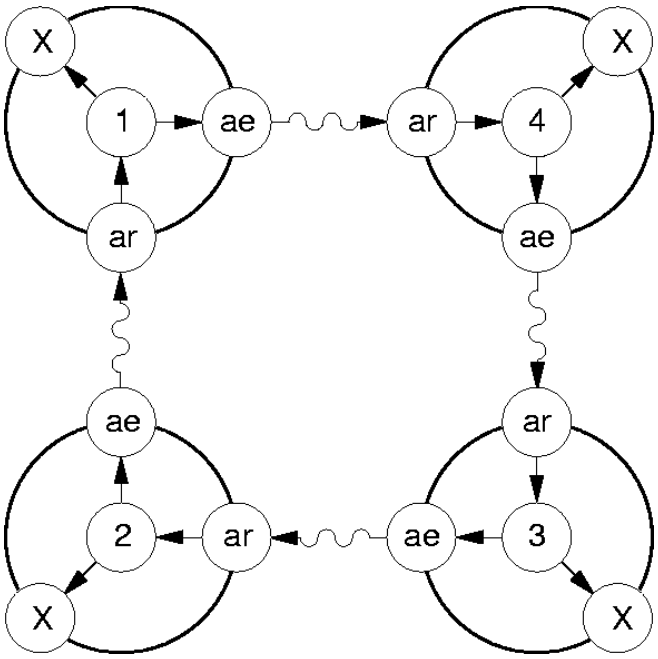

FIG. XI.3 – Quatre tortues, chacune sur son horloge

munications asynchrones se font par valeur la plus récente. Pour la première exécution, les périodes d'horloge sont respectivement 0,12s 0.1s 0.12s et 0.1s pour les tortues 1, 2, 3 et 4. Les trajectoires des tortues qui ont la même période d'horloge semblent symétriques. Elles sont effectivement similaires, mais la symétrie n'est pas garantie, comme le montrent les deux dernières exécutions. Ces deux exécutions se font en effet avec une période de 0.2 seconde pour chacune des horloges, or elles ne donnent pas exactement les mêmes trajectoires : le programme n'est pas déterministe. Le temps physique intervient dans le déroulement de ce programme et une tortue peut ne pas recevoir la position de sa cible à un instant si cette valeur n'est pas encore arrivée sur son récepteur asynchrone. Les instants des quatre horloges se partagent le temps d'exécution du programme et rien n'est fait pour synchroniser leur exécution. Le processeur peut donc être alloué à une horloge au beau milieu d'un instant d'une autre horloge.

L'utilisation de communications asynchrones en mode historique permet de faire disparaître ce non-déterminisme. Puisque toute l'histoire du signal est conservée, il devient impossible de rater un échantillon. Ainsi, quelque soit l'entrelacement des instants des horloges, chaque tortue recevra toute la progression de sa cible. Comme nous le voyons sur la figure XI.5, ce mode de communication n'est pas adapté à notre problème car on souhaite que chaque tortue se dirige le plus rapidement possible vers sa cible. Le mode de communication avec historique dirige la tortue vers une position que sa cible occupait précédemment, la poursuite prend alors la forme d'une spirale dès que les vitesses des tortues sont différentes. Par contre, les tortues convergent au centre de la figure, ce qui n'était pas toujours le cas avec la communication par valeur la plus récente.

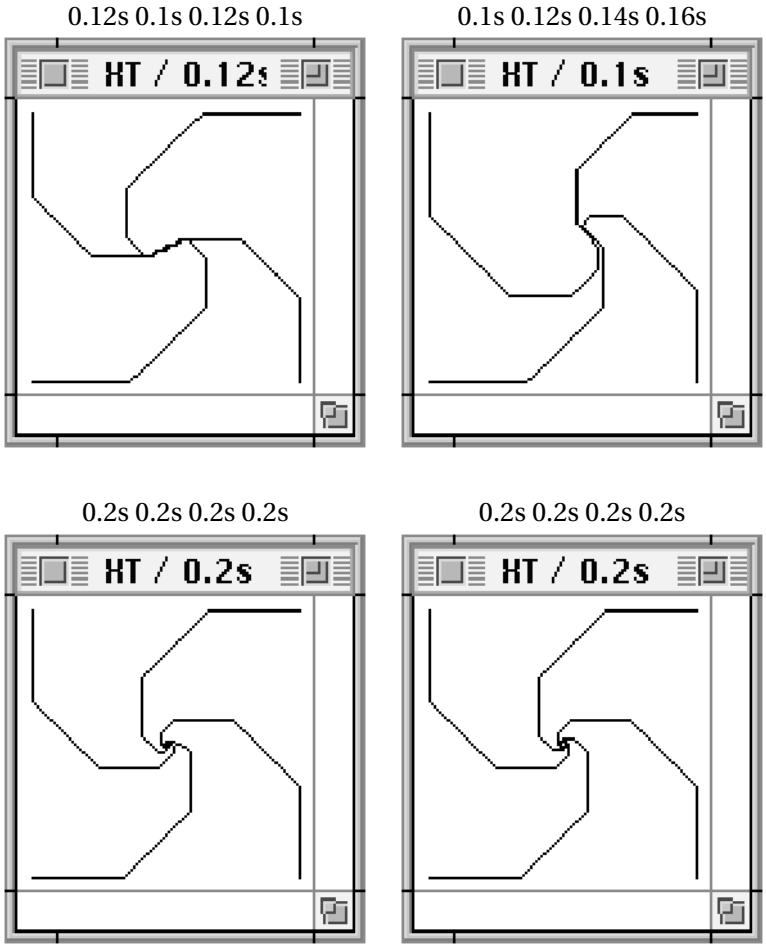

FIG. XI.4 – Trajectoire des tortues en communication asynchrone par valeur la plus récente

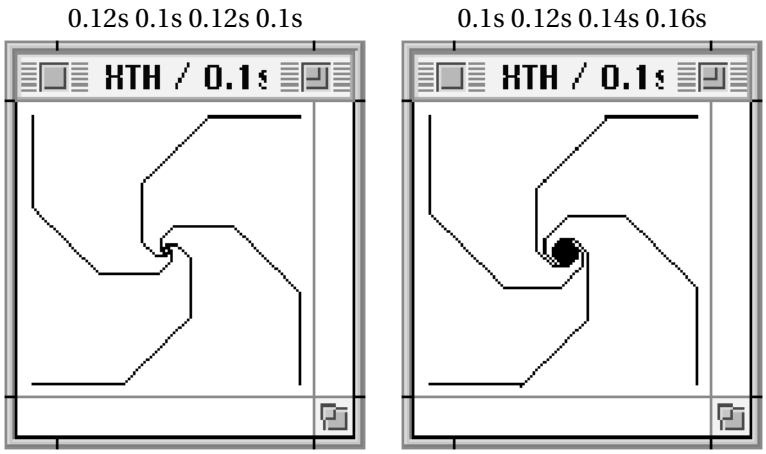

FIG. XI.5 – Trajectoire des tortues en communication asynchrone avec historique

XII Exemple d'instanciation dynamique

Comme nous l'avons vu, l'implémentation actuelle de l'ordonnanceur ne traite pas la dynamicité des objets synchrones conformement au modèle. Pour cet exemple, nous avons donc modifié son code afin que l'instanciation dynamique d'objets synchrones se fasse correctement. Cette version de l'ordonnanceur n'est pas définitive car elle n'autorise pas la destruction d'objets synchrones telle qu'elle est définie dans le modèle. Elle est toutefois suffisante pour illustrer l'utilisation d'objets synchrones instanciés dynamiquement en réaction à un événement.

Nous souhaitons calculer la factorielle d'un entier grâce au module synchrone suivant :

```
module facto:
  type pfacto;
  constant this:pfacto;
  procedure nouvFacto()(pfacto);

  input arg(integer);
  output res(integer);
  input subres(integer);
  output subarg(integer);

  var monArg:integer in
    await immediate arg;
    monArg := ?arg;
    if (monArg < 2) then
      emit res(1);
    else
      call nouvFacto()(this);
      await tick;
      emit subarg(monArg - 1);
      await immediate subres;
      emit res(monArg * ?res);
    end if
  end
end module
```

Ce module attend son argument, puis, s'il est égal à 0 ou 1, émet le résultat 1, sinon, il crée un nouveau module identique dont le signal arg est connecté à son signal subarg et le signal res à son signal subres à travers un retard pour couper la boucle de dépendance (procedure nouvFacto). Il attend alors l'instant suivant (pour que le nouveau module existe) pour envoyer son argument initial moins 1 à ce nouveau module. Il ne reste alors qu'à attendre le résultat pour pouvoir émettre le résultat du premier module.

On remarque la déclaration de la constante this qui correspond au pointeur sur l'objet. Cette constante est passée à la procédure nouvFacto afin qu'elle puisse connecter les signaux du nouvel objet à ceux du créateur. Le

code C++ utilisant ce module pour calculer une factorielle est donné ci-dessous. On remarquera qu'il fait appel à une instance du gabarit de classe constant pour les entiers. Les instances de constant<int> produisent un signal entier constant dont la valeur est fournie lors de l'instanciation. Le paramètre booléen du constructeur permet de préciser si le signal doit être émis en permanence où simplement lors de la première réaction.

```
#include "facto.H"
int main(int argc, char** argv) {
  int val;
  if (argc > 1) {
    istrstream buf(argv[1]);   // Récupérer l'argument sur la
    buf >> val;                //  ligne de commande
  } else {
      ...
  }
  theScheduler = new Scheduler("ordo");
  facto fact("appel facto");  // Racine du calcul
  constant<int> arg("arg", val, false);
                  // constante pour fournir son argument à fact
  OutputModule<int> pRes("printRes");
                  // pour afficher le résultat
  fact.arg() << arg.output();  // Connecter fact à son argument
  pRes.probe() << fact.res();  // et l'afficheur au résultat
  theScheduler->start();
  theScheduler->join();
  return 0;
}
```

L'appel de ce programme avec l'argument 5 donne :

```
>fact 5
120
>
```

et correspond à la création de 5 instances de facto comme le montre la figure XII.1.

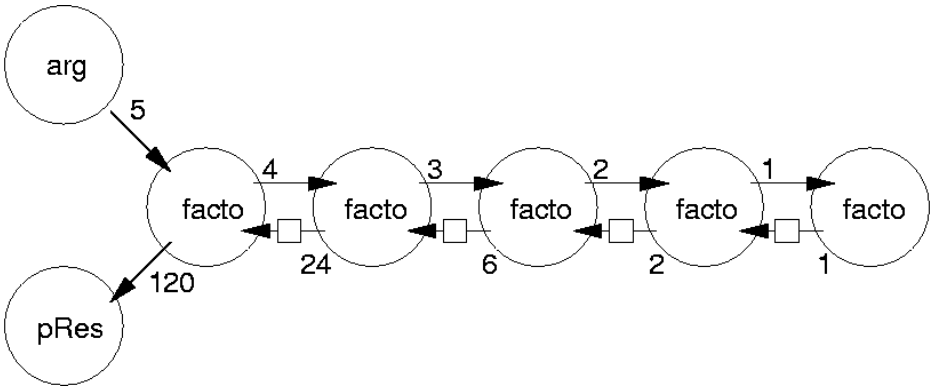

FIG. XII.1 – Instanciation dynamique d'objets synchrones

XIII Conclusion et perspectives

Le système présenté dans cette thèse 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 développement séparé des diverses parties de l'application que sur le plan de la maintenance et du prototypage.

XIII.1 Adaptabilité du système

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.

Les classes qui doivent être traitées suivant l'approche synchrone donnent lieu à l'écriture de modules dans un langage synchrone. L'implémentation actuelle de notre système ne traite, grâce à occ++, que le code oc produit par le compilateur Esterel, mais le code produit par d'autres compilateurs synchrones pourrait être interfacé de la même façon au prix d'un effort de codage minime. 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. Prenons pour exemple le système informatique de gestion d'une voiture : Nous définissons une classe voiture en y intégrant quatre instances de frein, une instance d'allumage_electronique, et pourquoi pas, pour le haut de gamme, une instance de systeme_anti_collision. Si l'on a pris soin de définir la classe frein_ABS comme sous-classe de frein, on pourra remplacer les instances de frein par des instances de frein_ABS sans avoir à modifier le code de voiture. La communication asynchrone entre les parties synchrones de voiture et ses parties asynchrones permettra à un objet depollueur qui analyse les gaz d'échappement, d'influer sur l'instance d'allumage_electronique afin d'optimiser la combustion. Lors d'un voyage en Angleterre, la dynamicité de la topologie de l'horloge permettra de remplacer sans s'arrêter l'instance de systeme_anti_collision par une version adaptée à la conduite à gauche.

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. La bibliothèque de classes synchrones ne fournit que les mécanismes de vérification dynamique des propriétés qui doivent être respectées pour que le modèle d'exécution reste valide.

Quand toutes les classes ont été développées, il ne reste qu'à procéder à l'édition de liens, sans oublier la bibliothèque synchrone, pour obtenir l'application.

Notre système se prête bien au développement modulaire, car toutes les parties de l'application étant des classes, et leur interface étant définie lors de la phase de conception, le codage et les tests d'une classe peuvent se faire grâce à une simulation des classes qu'elle utilise. Ainsi, pour reprendre notre exemple, il n'est pas nécessaire d'attendre que le code synchrone de la classe systeme_anti_collision soit développé pour tester la classe voiture : on peut utiliser une « coque vide » ayant la même interface et un comportement stéréotypé correspondant à la situation à tester. De même, 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.

XIII.2 Possibilités de prototypage et d'extension

Les mécanismes du système ne sont pas cachés dans le code d'un compilateur ou d'un pré-processeur ad hoc. Ils 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, d'expérimenter de nouvelles méthodes de communication entre le monde synchrone et le monde asynchrone, voire même d'envisager des systèmes de mise à jour de code par édition de lien dynamique pour des systèmes distants dont le fonctionnement ne peut être interrompu (satellites en orbite par exemple).

D'ailleurs, l'implémentation actuelle ne couvre pas tous les aspects du modèle, notamment ceux qui assurent la cohérence des systèmes dynamiques, et la poursuite de son développement se fait en créant de nouvelles classes et en les intégrant à la bibliothèque une fois l'expérimentation terminée. Ce qui est valable pour nous l'est aussi pour le développeur qui rencontre

un problème particulier et souhaite expérimenter différentes solutions pour les évaluer.

XIII.3 État actuel de l'implémentation

A l'heure actuelle, les outils occ++ et mdlc (qui représentent 6500 lignes de C) 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 (2000 lignes de C++) ne couvre pas tous les aspects du modèle actuellement. Elle a été compilée pour Ultrix 4.3 et pour DECelx 1.0 (version DIGITAL de VxWorks). Son portage sur d'autres plates-formes est conditionné par celui de la bibliothèque de threads (1300 lignes de C++) 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 templates. Il faut donc disposer d'un compilateur C++ au moins aussi performant que la version 3 de celui d'ATT pour la machine sur laquelle on veut utiliser l'application. Cette contrainte nous a d'ailleurs forcés à porter ce compilateur afin qu'il génère du code pour le 68030 sous DECelx de notre système temps-réel, ce qui nous a pris autant de temps que le développement des autres outils et bibliothèques.

Ceci nous a toutefois 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 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.

XIII.4 Perspectives

Il reste tout d'abord à terminer l'implémentation de la bibliothèque synchrone, avec passage au nouveau modèle d'exécution présenté au chapitre VII. 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 ? Ce dernier est en effet un objet synchrone, et a donc un certain nombre, fixe, de signaux. 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 aéronef afin de ne pas perdre d'échantillons. Mais comment 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. Avec $2^{k+1}-1$ de ces connecteurs, on peut multiplexer 2^{k+1} signaux sur un même signal. A charge au centre de contrôle de démultiplexer tout ce qui arrive sur son unique signal d'entrée. C'est en fait ce qui se passe sur un véritable aérodrome où une fréquence est attribuée à la tour pour toutes les communications, le contrôleur démultiplexant grâce à l'indicatif de l'appelant qui est répété dans chaque message.

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 ? Le multiplexage peut-il être considéré comme instantané ?

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 en avons donné, nous pensons que sous certaines contraintes imposées à l'histoire du système, il doit être possible de vérifier formellement des propriétés de ce système.

Références Bibliographiques

[AME-90] Pierre America
Designing an Object-Oriented Programming Language with Behavioural Subtyping
REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990 Proceedings.
in Foundations of Object-Oriented Languages, Lecture Notes in Computer Science
Springer-Verlag 1991

[AND-91] Charles André, Jean-Paul Marmorat et Jean-Pierre Paris
Execution Machines for 'Esterel'
Rapport de recherche I3S, CNRS - URA 1376 n° 91-32, Juillet 1991

[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

[BER-92] Gérard Berry, S. Ramesh, R.K. Shyamasundar
Communicating Reactive Processes

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

[COU-90] P. Couronné, J-P Paris, J-B Saint et J A Plaice
The Lustre-Esterel portable format

[GOL&ROB-83] Adele Goldberg and David Robson
Smalltalk-80, the Language and its Implementation
Addison-Wesley 1983

[GON-88] Georges Gonthier
Sémantiques et modèles d'exécution des langages réactifs synchrones ; application à Esterel
Thèse de doctorat, Orsay, Paris XI, 1988

[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

[HAL-93] Nicolas Halbwachs
Synchronous Programming of Reactive Systems
Kluwer Academic Publishers, 1993

[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

[HEW&BAK-77] Carl Hewitt et Henry Baker, Jr.
Actors and Continuous Functionals
Technical Report MIT/LCS/TR-194, Cambridge, 1977

[LAM-78] L. Lamport
Time, clocks and the ordering of events in a distributed system
Communications of the ACM, 21(7) :558-565, 1978

[LGU&GAU-90] Paul Le Guernic et Thierry Gautier
Data-flow to von Neumann : the SIGNAL approach
Publication interne n° 531, IRISA-INRIA, avril 1990

[MAF-91] Olivier Maffeïs, Bruno Chéron et Paul Le Guernic
Transformations du graphe des programmes SIGNAL
Publication interne n° 619, IRISA/INRIA, novembre 1991

[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

[STR-87] Bjarne Stroustrup
What is "Object-Oriented Programming" ?
ECOOP'87, BIGRE+GLOBULE n° 54, juin 1987

[STR-91] Bjarne Stroustrup
The C++ Programming Language, second edition
Addison Wesley, 1991

[UNI-91] UNIX System Laboratories
C++ Language System, Release 3.0.1

[WEG-87b] Peter Wegner
Dimensions of Object-Based Language Design
OOPSLA'87 Proceedings, October 4-8 1987

Bibliographie

[AND&PER-92] Charles André et Marie-Agnès Peraldi
Hard Real-Time System Implementation on a Microcontroller
WRTP'92, Bruges, juin 1992

[BER&GON] Gérard Berry, Georges Gonthier
The Esterel Synchronous Programming Language : Design, Semantics, Implementation

[BOO-88] Grady Booch
Ingénierie du Logiciel avec ADA
InterÉditions, 1988

[BOUL-93a] Frédéric Boulanger, Henri Delebecque, Guy Vidal-Naquet
Intégration de Modules Synchrones dans la Programmation par Objets
Rapport de Recherche n^o 864, septembre 1993
Laboratoire de Recherche en Informatique, Université de Paris Sud, Centre d'Orsay

[BOUL-93b] Frédéric Boulanger, Henri Delebecque, Guy Vidal-Naquet
Intégration de Modules Synchrones dans la Programmation par Objets
Accepté au Salon RTS'94, 11-14 janvier 1994, Palais des Congrès, Paris

[BOU-92] Frédéric Boussinot
Réseaux de Processus Réactifs
Rapport de Recherche INRIA n^o 1588, janvier 1992

[BOU&DOU-91] Frédéric Boussinot et Guillaume Doumenc
Reactive C Reference Manual
Décembre 1991

[BOU&DSI-91] Frédéric Boussinot, Robert de Simone
The ESTEREL Language
Proceedings of the IEEE, vol. 79, n^o 9, September 1991

[CHA&COU-91] J. Chailloux, P. Couronné
AGEL : Un atelier de développement de systèmes réactifs synchrones
Génie Logiciel et Systèmes Experts, n^o 25, décembre 1991

[CIS-88] ISI INGENIERIE
Esterel V3 Documentation
1988

[COL-92] Derek Coleman, Fiona Hayes, Stephen Bear
Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design
IEEE Transactions on Software Engineering, vol. 18, n^o 1, January 1992

[DON-91] Christophe Dony, Jan Purchase et Russel Winder
Exception Handling in Object-Oriented Systems
Report on ECOOP'91 Workshop W4, 1991

[DUT-91] Bruno Dutertre, Jean-Christophe Gilbon, Paul Le Guernic, Guy Vidal-Naquet
SOSIE, Spécification Objet Synchrone IntégréE
MEMO-DIN T/28/91, document interne Alcatel Alsthom Recherche et IRISA/INRIA
Janvier 1991

[DZI-90] Daniel Dzierzgowski
Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient.
Technique et Science Informatique, avril 1990

[GER&LEE-92] Richard Gerber et Insup Lee
A Layered Approach to Automating the Verification of Real-Time Systems
IEEE Transactions on Software Engineering, vol. 18, n^o 9, September 1992

[LGU-91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne et Claude Le Maire
Programming Real-Time Applications with SIGNAL
Proceedings of the IEEE, vol. 79, n^o 9, September 1991

[MEY-90] Bertrand Meyer
Conception et Programmation par Objets
InterÉditions, 1990

[MEY-92] Bertrand Meyer
Introduction à la théorie des langages de programmation
InterÉditions, 1992

[NIC-92] Xavier Nicollin, Joseph Sifakis et Sergio Yovine
Compiling Real-Time Specifications into Extended Automata
IEEE Transactions on Software Engineering, vol. 18, n^o 9, September 1992

[SHA-92] Alan C. Shaw
Communicating Real-Time State Machines
IEEE Transactions on Software Engineering, vol. 18, n^o 9, September 1992

[WEG-87a] Peter Wegner
The Object-Oriented Classification Paradigm
Research Directions in Object-Oriented Programming
MIT Press, 1987

Annexe A Manuel de Référence d'Occ++

Occ++ traduit le code oc d'un module synchrone en une classe synchrone C++. La version 3.2.8 reconnaît le code oc v3 généré par le compilateur Esterel.

La présence de références externes (types, constantes, fonctions et procédures) provoque l'inclusion automatique d'un fichier xxDef.H par le fichier d'interface généré. Le nom de ce fichier peut être changé par l'option -H <nomDeFichier>, et cette inclusion peut être inhibée par l'option -Hnone.

Le code C++ généré par occ++ utilise les opérateurs standards de C++, il est donc inutile de fournir le jeu de fonctions implémentant ces opérateurs comme cela est décrit dans le manuel d'interface avec C d'Esterel. Il faut toutefois que ces opérateurs soit définis pour la classe importée en Esterel. Ainsi la fonction _TYPE appelée par occ pour faire une affectation d'une variable de type TYPE est remplacée par l'opérateur = de C++. De même pour la fonction _eq_TYPE qui est remplacée par l'opérateur == , etc.

Occ++ ne crée qu'un seul fichier .C et le fichier .H associé. Si le source oc contient plusieurs modules, plusieurs classes seront définies dans ce fichier. Les autres options d'occ++ sont :

-v provoque l'affichage des modules et de leurs caractéristiques au fur et à mesure de leur traduction en C++

-version affiche la version d'occ++

-B <nom> impose <nom> comme nom de base pour les fichiers produits. Par défaut, occ++ prend le nom du premier module rencontré.

-D <dir> indique le répertoire dans lequel seront placés les fichiers produits.

-cut <n> limite la longueur des lignes dans les tables d'état à <n> actions par ligne.

-debug insère du code permettant de suivre la réaction de l' automate (affichage sur stderr). La compilation de ce code est soumise à la définition du symbole _SYNC_DEBUG_.

-tab crée un fichier par état contenant la table des actions pour cet état. Cette option est utile lorsque les tables d'état deviennent suffisamment grosses pour ne plus pouvoir être compilées toutes ensemble — gcc-cpp semble avoir des problèmes avec les fichiers volumineux.

-tabB <pref> permet de spécifier un préfixe pour les noms des fichiers de tables d'état. Ce préfixe est purement textuel et peut donc comporter un chemin d'accès à un répertoire. Cette option implique automatiquement l'option -tab.

-mdf génère un fichier de description de module pour chaque module traité par occ++. Le nom du fichier est <xx>.mdf, où <xx> est le nom du module correspondant.

-mdfB <pref> permet de spécifier un préfixe pour les noms de fichiers de description de module. Le principe est le même que pour -tabB. Cette option implique automatiquement l'option -mdf.

-param indique à occ++ que les constantes du module doivent être traduites en constantes des objets, la valeur de ces constantes étant fournie lors de l'instanciation sous forme d'une structure de type <nom_module>Consts. A chaque constante du module correspond un membre de même nom et de même type dans la structure. Lorsque l'option -param n'est pas donnée, occ++ implémente les constantes du module sous forme de constantes de la classe, qui sont donc communes à toutes les instances.

Annexe B Manuel de Référence de Mdlc

Mdlc permet de construire des modules synchrones par composition d'autres modules synchrones. Il permet aussi de définir un module comme héritant d'un autre module. Ces modules sont traduits en classe synchrone C++. La version décrite ici est la version 3.2.0. Mdlc ne construit qu'un module à la fois. Il ne doit donc y avoir qu'une seule spécification de module parmi les fichiers passés à mdlc. Les fichiers de description de module auront en général été créés automatiquement par occ++ ou mdlc. Mdlc supporte les options suivantes :

-v provoque l'affichage des modules traités par mdlc.

-version donne le numéro de version de mdlc.

-B <nom> permet de changer le nom des fichiers C++ produits. Par défaut, ce nom est <xx>.C et <xx>.H, où <xx> est le nom du module généré.

-debug insère du code permettant de suivre la réaction du module produit. La compilation de ce code est soumise à la définition du symbole _SYNC_DEBUG_.

-mdf génère un fichier de description du module produit

-mdfB <nom> permet de changer le nom du fichier de description de module. Par défaut, ce nom est <xx>.mdf, où <xx> est le nom du module décrit.

B.1 Structure d'un fichier mdl

Un fichier mdl permet de définir un nouveau module en fonction d'autres modules. Il doit être passé à mdlc en même temps que les fichiers de description (en .mdf) des modules auxquels il fait référence.

Dans un fichier mdl, les commentaires sont introduits par la séquence '//' et se terminent à la fin de la ligne. Trois pragmas permettent de garder une trace des origines d'un module :

`#source <fichier_source>` indique le fichier dans lequel le module est décrit. Ce fichier peut contenir de l'Esterel ou du mdl dans l'implémentation actuelle.

`#oc <fichier_oc>` indique, s'il existe, le fichier contenant le code oc du module.

`#c++ <fichier_interface>` indique le fichier contenant l'interface de la classe C++ correspondant au module. Ce fichier est automatiquement inclus par le fichier d'interface de la classe produite par mdlc s'il est fait référence à ce module dans la définition du nouveau module.

B.1.1 Description d'un module

Un module est décrit selon la grammaire suivante, les éléments non terminaux étant entre < et >, et les identificateurs entre ‘ et ’ :

```
module = 'nom' {
  <déclarations_de_signaux>
}

déclarations_de_signaux =
  <déclarations_de_signaux> <déclaration_signaux>
  |
  <déclaration_signaux>

déclaration_signaux =
  <déclaration_entrées>
  |
  <déclaration_sorties>

déclaration_entrées =
  input: <liste_signaux> ;

décalaration_sorties =
  output: <liste_signaux> ;

liste_signaux =
  <liste_signaux> , <signal>
  |
  <signal>

signal =
  <signal_typé>
  |
  <signal_pur>

signal_typé = 'nom_type' 'nom_signal'

signal_pur = 'nom_signal'
```

Par exemple, un module ‘horloge’ prenant un signal pur ‘tick’ en entrée et produisant les signaux entiers ‘heures’, ‘minutes’ et ‘secondes’ sera décrit comme suit :

```
horloge {
  input: tick;
  output: integer heures, integer minutes, integer secondes;
}
```

B.1.2 Définition d'un module composé

Un module composé est définit selon la grammaire suivante, les non terminaux qui ne sont pas redéfinis étant ceux de la grammaire précédente :

```
module_composé = 'nom' : {
  <déclarations_de_signaux>
  <déclarations_de_modules>
  <équivalences_signaux>
  <connexions_signaux>
}
```

```
déclarations_de_modules =
  <déclarations_de_modules> <déclaration_module>
  |
  <déclaration_module>

déclaration_module =
  'nom_type' 'nom_module' ;

équivalences_signaux =
  <équivalences_signaux> <équivalence_signaux>
  |
  <équivalence_signaux>

équivalence_signaux =
  <signal> = <signal> ;

connexions_signaux =
  <connexions_signaux> <connexion_signaux>
  |
  <connexion_signaux>

connexion_signaux =
  <signal> << <signal> ;

signal =
  <signal_sousmodule>
  |
  <signal_module>

signal_sousmodule =
  'nom_module'.'nom_signal'

signal_module =
  'nom_signal'
```

Par exemple, pour définir un frein ABS à partir d'un frein et d'un contrôleur ABS, on écrira :

```
frein_ABS : {
  input: integer pedale, glissement;
  output: integer pression;

  frein f;
  ABS_controleur c;

  f.pedale = pedale;
  c.glissement = glissement;
  pression = c.pression_filtree;

  c.pression_brute << f.pression;
}
```

Les descriptions de frein et de ABS_controleur étant les suivantes :

```
frein {
  input: integer pedale;
  output: integer pression;
}
```

```
ABS_controleur {
  input: glissement, integer pression_brute;
  output: integer pression_filtree;
}
```

B.1.3 Définition d'un module dérivé

La définition d'un module dérivant de son super module se fait selon la grammaire suivante, les non terminaux étant ceux des grammaires précédentes :

```
module_dérivé = 'nom' : 'nom_super' {
  <déclarations_de_signaux>
  <déclarations_de_modules>
  <équivalences_signaux>
  <connexions_signaux>
}
```

La différence avec un module composé est que le module dérivé possède automatiquement les signaux de son super module. Seuls les signaux additionnels doivent être déclarés. Pour définir frein_ABS comme héritant de frein, on écrira :

```
frein_ABS : frein {
  input: glissement;

  ABS_controleur c;

  c.glissement = glissement;
  pression = c.pression_regulée;

  c.pression_brute << pression;
}
```

Notons que le signal pression de « `pression = ...` » et celui de « `... << pression` » ne sont pas les mêmes. Dans le premier cas, il s'agit du signal pression du nouveau module — les équivalences ne peuvent se faire que sur des signaux d'interface — alors que dans le second cas, il s'agit du signal pression hérité du super module — on ne peut pas connecter un signal interne à un signal d'interface.

Résumé

Nous présentons un système permettant d'intégrer des modules synchrones dans un langage à objets de façon à ce qu'ils bénéficient automatiquement d'une machine d'exécution leur permettant de communiquer entre eux et avec le monde asynchrone.

Cette approche permet de bénéficier des avantages des langages synchrones pour le traitement des problèmes temps-réel, et de ceux des langages à objets pour le développement global d'une application.

Ce système se compose d'un traducteur d'OC (langage de description d'automates utilisé comme code intermédiaire par les compilateurs Lustre et Esterel) en C++, d'un compilateur de modules, et d'une bibliothèque de classes synchrones. Ces trois outils s'appuient sur un modèle qui définit le comportement et les interactions entre objets synchrones et entre un groupe d'objets synchrones et le monde asynchrone. Ce modèle couvre les aspects temporels (communication synchrone et asynchrone, notions d'instant et d'horloge) et les aspects objet (instanciation, destruction), ainsi que leurs conséquences dans le modèle temporel.

Nous avons délibérément ignoré le problème de la compilation des langages synchrones et avons pris comme point de départ le code intermédiaire OC. Ce choix nous donne une relative indépendance vis-à-vis du langage synchrone choisi, puisque Esterel et Lustre produisent directement du code oc, et que Signal est capable de le faire, même si cette représentation n'est pas la plus naturelle pour les modules écrits dans ce langage.

Le traducteur occ++ produit une classe C++ à partir du code oc d'un module synchrone. Chaque instance de cette classe a le comportement du module et communique avec d'autres objets, synchrones ou non, à travers une interface qui dissimule et protège son état. On bénéficie ainsi des propriétés de l'encapsulation pour les modules synchrones.

Le compilateur de modules, mdlc, traduit une description de module en une classe C++ analogue à celles produites par occ++. Le langage de description de module, mdl, permet de définir de nouveaux modules à partir de modules préexistants, ce qui autorise une forme de compilation séparée lorsqu'il n'y a pas de cycle dans le graphe de dépendance des modules composants. Mdl permet aussi de construire des sous-classes de modules synchrones, ce qui permet de hiérarchiser l'ensemble des classes synchrones grâce à l'héritage.

Enfin, la bibliothèque de classes synchrones fournit la machine d'exécution des objets synchrones. Elle contient tout ce qui n'est pas spécifique à un module particulier, notamment le moteur d'exécution des automates, les vecteurs de communication entre modules et des classes utilitaires pour les entrées-sorties, l'accès au temps système etc.

Points essentiels

- Encapsulation des modules synchrones

  Notre approche permet d'intégrer facilement du code synchrone dans une application développée suivant une approche par objets. La définition de classes d'objets synchrones permet de bénéficier de l'encapsulation, qui oblige le programmeur à manipuler le module synchrone à travers son interface, ce qui évite les altérations accidentelles de l'état interne du module.

- Communication synchrone

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

- Dynamicité des objets synchrones

  Il est possible de créer et de détruire des objets synchrones, ainsi que 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.

- Composition de modules et héritage

  La possibilité de l'instanciation dynamique et la reconfiguration des communications entre modules à l'exécution empêche de connaître a priori les dépendances entre modules. L'ordonnancement doit donc être fait à l'exécution, ce qui est moins performant qu'un ordonnancement statique. Dans les cas ou le nombre de modules est fixe et où les connexions entre modules ne varient pas, on aura donc intérêt à construire un module composite grâce à mdl. Ce langage de description de modules permet en effet de construire un nouveau module synchrone en interconnectant des modules pré-existants. Toutes les vérifications (type des signaux, modules n'intervenant pas dans la réaction du nouveau module, présence de boucles dans le graphe de dépendance) sont faites à la compilation, ainsi que l'ordonnancement. On peut ainsi éviter le coût de la dynamicité lorsqu'elle n'est pas nécessaire. Mdl permet aussi de décrire des modules comme dérivant (au sens des langages à objets) d'un autre module. La classe C++ générée par mdlc sera alors sous-classe de la classe du module de base. Ceci

est particulièrement utile du point de vue du langage à objets : on peut par exemple spécifier qu'une voiture à quatre freins, chaque frein étant une instance de la classe Frein. Si l'on définit la classe FreinABS comme sous-classe de Frein, une voiture pourra avoir quatre freins ABS sans qu'il soit nécessaire de modifier sa définition.

– Communication asynchrone

  La communication asynchrone entre les objets synchrones et les autres objets du programme est assurée par des objets d'interface. 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.

Mots-clef

Synchrone, Objets, Temps-Réel, Systèmes Réactifs,
Outils de Développement