Objets et Programmation Synchrone

Charles ANDRÉ[1] Frédéric BOULANGER[2] Marie-Agnès PÉRALDI[1,3]
Jean-Paul RIGAULT[4] Guy VIDAL-NAQUET[2]

[1] Laboratoire Informatique, Signaux, Systèmes (I3S) — Université de Nice-Sophia Antipolis / CNRS
41, bd Napoléon III — 06041 NICE Cedex

[2] Laboratoire de Recherche en Informatique — Université Paris-Sud (Orsay) et Supélec
SUPÉLEC — Plateau de Moulon — 91192 GIF SUR YVETTE Cedex

[3] Laboratoire d'Informatique Technique — École Polytechnique Fédérale de Lausanne
IN-Ecublens — CH-1015 LAUSANNE — Suisse

[4] Ecole Supérieure en Sciences Informatiques (ESSI) — Université de Nice-Sophia Antipolis
BP 145 — 06903 SOPHIA ANTIPOLIS Cedex

Résumé

Facilités de structuration, d'abstraction, de réutilisation et d'évolution sont quelques unes des caractéristiques de l'approche par objets. Description formellement rigoureuse de la partie réactive, permettant des preuves de correction logique, telle est l'essence de la programmation synchrone.

Cet article propose une combinaison de ces deux approches. Il introduit la notion d'*objet synchrone* au travers d'un exemple. Divers problèmes relatifs aux objets et au synchronisme sont mis en évidence. Un environnement complet de développement, en cours de réalisation, est présenté. Il dispose d'éditeurs, de compilateurs, de simulateurs et d'interfaces vers des systèmes de preuves formelles. Cette plate-forme devrait contribuer à l'amélioration de la qualité des logiciels, en particulier dans le domaine du temps réel.

Abstract

Clear structure, support for abstraction, reuse, and evolution, these are the striking features of the object-oriented approach. Formal description of the reactive behavior, making it possible to prove logical correctness, this is the essence of the synchronous paradigm.

This paper proposes to combine these two approaches. An introductory example presents the notion of a *synchronous object*. Then, various issues related to objects and synchrony are addressed. Finally, we report on our progress in building a complete design and programming environment. Editors, compilers, simulators, interfaces towards model checkers are integrated within this environment, which should contribute to better software quality in the field of real-time.

1 Introduction

La *notion d'objet* est un bon paradigme pour la programmation des *systèmes complexes*. L'un des principaux bienfaits de l'approche par objets est l'amélioration de l'architecture des programmes. Celle-ci est obtenue d'abord par une décomposition modulaire rigoureuse reposant sur les entités réelles manipulées par l'application (les objets) ; puis, pour chacune de ces entités, par la séparation entre leur interface (abstraite) d'*utilisation* et les détails de leur *implémentation*. Des mécanismes puissants comme la composition, l'héritage, le polymorphisme permettent de prendre avantage de cette séparation et de doter le système de propriétés d'extensibilité et de réutilisabilité, favorisant son évolution et sa maintenance.

Jusqu'à présent, la programmation par objets a été surtout appliquée aux *systèmes transformationnels*, c'est à dire à des systèmes qui effectuent leurs traitements à un rythme choisi par eux-mêmes. À l'opposé, il existe des systèmes dits *réactifs* [HP85] qui sont en relation permanente avec leur environnement et dont le rythme d'évolution est imposé par l'extérieur. De nombreux systèmes réactifs sont également des systèmes *temps-réel* : leurs réactions sont contraintes temporellement.

La *programmation synchrone* [BB91] a été introduite pour répondre aux exigences de la programmation réactive. Les langages synchrones ont des sémantiques mathématiques simples et rigoureuses. Il a donc été possible de développer des compilateurs sûrs et efficaces. De plus la *correction logique* des programmes peut être établie formellement. Toutefois, les langages synchrones sont essentiellement dédiés au contrôle. Ils s'appuient sur des langages classiques et, pour effectivement exécuter le code produit par un compilateur de langage synchrone, l'utilisateur doit fournir une *machine d'exécution* [AP93].

Les recherches présentées ont pour but de *combiner approches synchrones et approches par objets*, afin de pouvoir programmer des *systèmes réactifs complexes*. La coopération peut se faire à plusieurs niveaux : analyse, conception globale ou détaillée, implémentation.

L'article est organisée de la façon suivante :

– Nous débutons par un rappel des caractéristiques des approches par objets.
– Nous présentons ensuite un exemple portant sur la modélisation des modes de fonctionnement d'un magnétophone. Cet exemple comprend des objets classiques (cassette) et beaucoup de contrôle, que nous modélisons par des objets synchrones.
– Dans une troisième partie, nous expliquons ce que sont les *objets synchrones*.
– La quatrième partie traite des *communications* entre objets.
– Finalement, nous introduisons brièvement un environnement de développement dédié à l'approche préconisée.

2 Caractéristiques des approches par objets

L'approche par objets permet de considérer le système à développer comme une collection d'objets discrets qui coopèrent. Ces objets sont des abstractions de sous-systèmes qui intègrent états (données) et opérations sur ces données. Les données sont destinées à être *encapsulées*, c'est-à-dire qu'elles ne peuvent être accédées directement, mais seulement par l'intermédiaire des opérations — on dit aussi *méthodes* — de l'objet. Ainsi, chaque objet est-il responsable de ses propres données, et par là même de son propre état et de son propre comportement. La liste de méthodes fournit l'interface d'*utilisation* de l'objet, seule information que les autres objets ont besoin de connaître.

Dans les méthodologies d'analyse et de conception par objets (*e.g.* OMT [RBP$^+$91]) apparaissent trois modèles différents :

1. le *modèle objet* proprement dit, souvent appelé également *modèle de classes*[1], qui décrit les objets du système, leur interface d'utilisation, et les relations (statiques) qui existent entre les objets ;
2. un *modèle dynamique* qui précise d'une part les interactions entre les objets, d'autre part le comportement (réactif) de chaque objet individuel ;
3. un *modèle fonctionnel* qui traite des transformations de données.

Un des attraits majeurs de l'approche objet est son pouvoir d'abstraction : elle permet d'exprimer *ce qui* change dans un système sans nécessairement préciser *comment* s'effectuent ces changements. L'ajout ou le retrait d'objets est relativement facile à prendre en compte. De même la modification locale d'un objet est aisée si cette modification ne met en cause ni l'interface publique de l'objet, ni ses relations avec les autres objets. Ainsi l'incrémentalité du développement (ajout d'objets), la réutilisabilité (des objets) et leur maintenabilité (modifications locales) peuvent elles être renforcées.

3 Présentation d'un exemple

Nous voulons modéliser et simuler le fonctionnement d'un magnétophone[2]. Celui-ci comprend deux platines : (`Deck1` lecture uniquement et `Deck2` lecture-enregistrement). Les fonctions de lecture et enregistrement présentent des options sophistiquées d'enchaînement d'opérations (copie en auto-reverse, saut en lecture des zones non enregistrées...). Nous décrivons ce système comme un ensemble d'*objets réactifs coopérants*. Parmi ces objets, certains sont quasiment

[1] En effet la plupart des méthodes d'analyse et de conception par objets utilisent des classes pour représenter les propriétés communes à un ensemble d'objets (*i.e.* le *type*). Les modèles utilisent alors les classes, plutôt que des objets individuels.

[2] Cette illustration reprend une expérimentation, non publiée, faite par Frédéric Boussinot avec les Objets Réactifs[BDS95].

imposés par l'application, ce sont les capteurs (boutons poussoir, palpeurs…) et les actionneurs (moteurs, têtes magnétiques, voyants). Les cassettes magnétiques se prètent bien à une modélisation par objet qu'on peut qualifier de classique puisqu'on rassemble dans le même modèle les attributs de la bande magnétique et les méthodes d'accès associées. D'autres objets sont introduits par notre structuration. Ce sont les contrôleurs. Alors qu'une approche comme SA/RT fait jouer un rôle particulier aux contrôleurs, ils ne sont pour nous que des objets ayant un caractère réactif prépondérant.

En illustrant nos propos par cette application, nous allons montrer les aspects modèle de classe (son intérêt et ses insuffisances), puis les aspects dynamiques (en introduisant un nouveau modèle). Dans cet exemple, les aspects fonctionnels sont réduits ; ils seront omis.

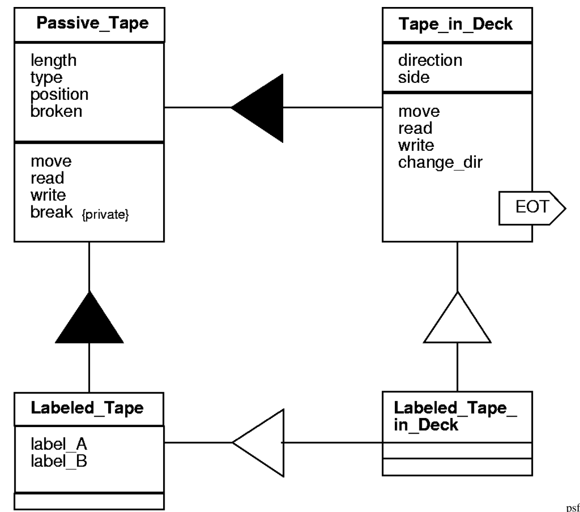

FIG. 1 – Modèle objet d'une bande magnétique

3.1 Le modèle objet

La figure 1 présente un modèle de classes OMT (voir [RBP+91]) pour l'un des objets de l'application : la bande magnétique. En fait, il y a deux sortes de bandes magnétiques : la bande proprement dite (`Passive_Tape`) et la bande une fois chargée dans le magnétophone (`Tape_in_Deck`). Le premier objet est relativement passif (sa seule propriété dynamique est de pouvoir se briser). Le second enrichit à la fois la structure de données et le comportement du premier par le choix d'une face et d'une direction de lecture d'une part, et la possibilité de générer un événement de fin de bande (`eot`) d'autre part[3]. L'héritage (dénoté par un triangle en OMT) convient bien pour représenter ce type d'enrichissement. On peut d'ailleurs dériver d'autres type de bandes, comme celles qui disposent d'une étiquette (`Labeled_Tape`), ou encore celles qui auraient un comportement spontané d'auto-destruction (les bandes de « Mission impossible »). L'héritage multiple permet alors (comme indiqué sur la figure 1) de représenter ces nouvelles bandes une fois insérées dans la platine[4].

Quand il s'agit de représenter la platine elle-même, la situation est un peu plus complexe. On souhaite bien sûr représenter une platine lecteur-enregistreur comme une spécialisation (c'est-à-dire une héritière) d'un lecteur seul. On peut représenter les composants d'une platine (le contrôleur, le(s) moteur(s), la ou les tête(s)) par agrégation comme dans le diagramme d'objets de la figure 2. Noter que l'association entre la platine est la bande n'est pas représentée par une agrégation d'OMT car la platine existe parfaitement sans qu'une bande y soit chargée. La seule difficulté dans ce modèle est que si une platine de lecture seule se contente d'un contrôleur simple (`Read_Controller`), la platine complète doit contenir un contrôleur complet (`Full_Controller`)[5].

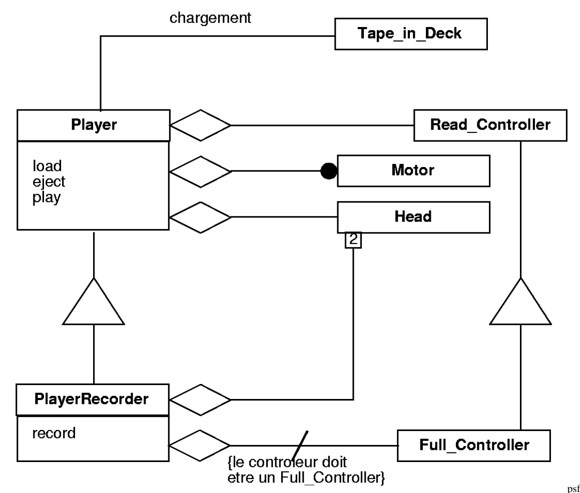

FIG. 2 – Modèle objet d'une platine (agrégation)

Une autre solution est de dériver (faire hériter) la platine du contrôleur (figure 3). Comme logiquement une platine *n'est pas* un contrôleur, nous avons recours à l'héritage privé que nous notons par une petite barre. Le schéma est alors élégamment symétrique, l'héritage multiple permettant de représenter correctement la structure du lecteur-enregistreur, par rapport à celle du lecteur seul.

On retrouve ici une discussion classique dans les approches par objets : l'héritage peut y être utilisé soit avec la sémantique du *sous-typage*, soit comme un simple moyen de partager l'implémentation (le code).

Bien entendu, le magnétophone lui-même est une agrégation de deux platines : `Deck1` est une instance de la classe `Player` (lecteur), alors que `Deck2` est une instance de la classe `PlayerRecorder`.

Ce que ne montrent pas les différents modèles d'OMT, ce sont les échanges de messages (ou d'évé-

[3] Cette émission « spontanée » d'événement n'est pas représentable directement dans un modèle de classes OMT ; nous avons donc enrichi la notation.

[4] Noter l'utilisation des triangles noirs d'OMT qui dénotent l'héritage multiple *virtuel*, c'est-à-dire avec une seule copie de la classe de base (ici `Passive_Tape`) dans les classes dérivées.

[5] Une fois de plus cette situation n'est pas directement exprimable en OMT ; nous l'avons représentée comme une *contrainte* sur une association d'agrégation *dérivée*.

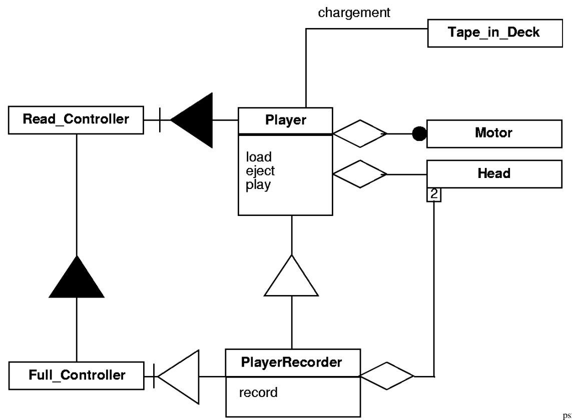

FIG. 3 – Modèle objet d'une platine (héritage)

nements) entre les différents objets d'une application. Nous avons donc introduit un nouveau type de diagrammes pour expliciter ces liens dynamiques (figure 5). Sur ces diagrammes, les boîtes sont des objets, les méthodes de ces objets sont représentées par des « ports » entrants « » ou sortants « ▭ ». Les liaisons sont orientées d'un port de sortie vers un port d'entrée ; elles représentent des communications point à point. Ces aspects communication seront repris dans la section 5. Notons que pour alléger le schéma, plusieurs signaux peuvent être regroupés en bus (c'est le cas de signaux de sortie de l'objet `keyboard` : `play, ..., rec`) qui se retrouvent en entrée de l'objet `Controller`.

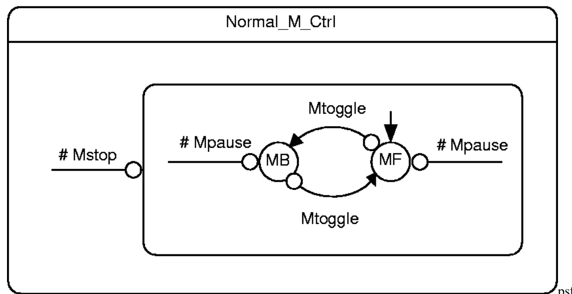

FIG. 4 – Contrôle du moteur

3.2 Le modèle dynamique

Pour le *modèle dynamique* proprement dit, qui exprime le comportement de chaque objet individuellement, nous avons adopté une représentation graphique « SYNCCHARTS » [And95] (Synchronous Charts) qui s'inspire des STATECHARTS : on y retrouve les notions de macro-état, hiérarchie, orthogonalité, diffusion d'événements. Les améliorations sont apportées par la distinction explicite de différents types de préemptions [Ber92] (suspensions ou avortements, faibles ou forts). Un SYNCCHART peut être traduit de façon systématique en programme ESTEREL. Une présentation complète du modèle sort du cadre de cet article. Nous allons seulement commenter deux exemples. Le comportement d'un contrôleur de lecture-enregistrement de base est donné figure 6. Deux macro-états ont des évolutions concurrentes (`Normal_M_Ctrl` et `Basic_Ctrl`). Le comportement de `Normal_M_Ctrl` est lui-même précisé dans la figure 4.

```
module Normal_M_Ctrl :
  input Mstop, Mpause,
Mtoggle;
  output MF, MB;
  suspend
    loop
      do
        suspend
          sustain MF
        when immediate Mpause
      watching Mtoggle;
      do
        suspend
          sustain MB
        when immediate Mpause
      watching Mtoggle
    end loop
  when immediate Mstop
end module
```

TAB. 1 – Exemple de traduction ESTEREL

Les arcs indiquent des possibilités de préemption sur l'occurrence d'événements associés à l'arc. Le petit rond placé à l'origine d'un arc exprime la *préemption forte*, c'est à dire que lors de l'occurrence d'un événement déclencheur, le comportement interne du macro-état à préempter est ignoré (une préemption faible, exprimée par l'absence du petit rond, aurait autorisé les « dernières volontés » de l'état préempté). Le modèle est complètement *déterministe*. Les entiers placés près de l'origine d'un arc expriment la priorité dans les choix. Dans `Basic_Ctrl_State` (figure 6) la préemption par l'événement `rec` est la plus prioritaire ; celle par `rewind` est la moins prioritaire.

Dans un modèle synchrone, la notion d'instant joue un rôle central. Il faut distinguer la possibilité de prendre en compte les occurrences strictement futures de celles présentes ou futures. Le caractère « # » précise que l'éventuelle présence dans l'instant courant doit être prise en compte (préemption immédiate). L'absence de ce symbole indique une préemption sur une occurrence strictement future. Un identificateur, placé dans une ellipse, dénote un signal qui est émis à chaque instant pour lequel le macro-état englobant est actif. On peut utiliser des signaux locaux à un macro-état (par exemple `Mpause,Mtoggle,Mstop` pour l'état `PlayRec_Ctrl`). Ces signaux sont essentiellement utilisés pour des communications internes. Les signaux locaux précités permettent de synchroniser `Basic_Ctrl` et `Normal_M_Ctrl` au sein de `PlayRec_Ctrl`.

Dans la figure 4 apparaît un nouveau type d'arc : celui de la *suspension*. La suspension peut être immédiate (par exemple pour `Mstop`), ou pour une occurrence strictement future (cas de `Mtoggle`). Lorsqu'un événement de suspension est présent, le macro-état correspondant est « inhibé ». Tout se passe pour lui comme si le temps ne passait plus.

En résumé, les SYNCCHARTS permettent d'exprimer des comportements réactifs sophistiqués. Ils peuvent être traduits de façon systématique en pro-

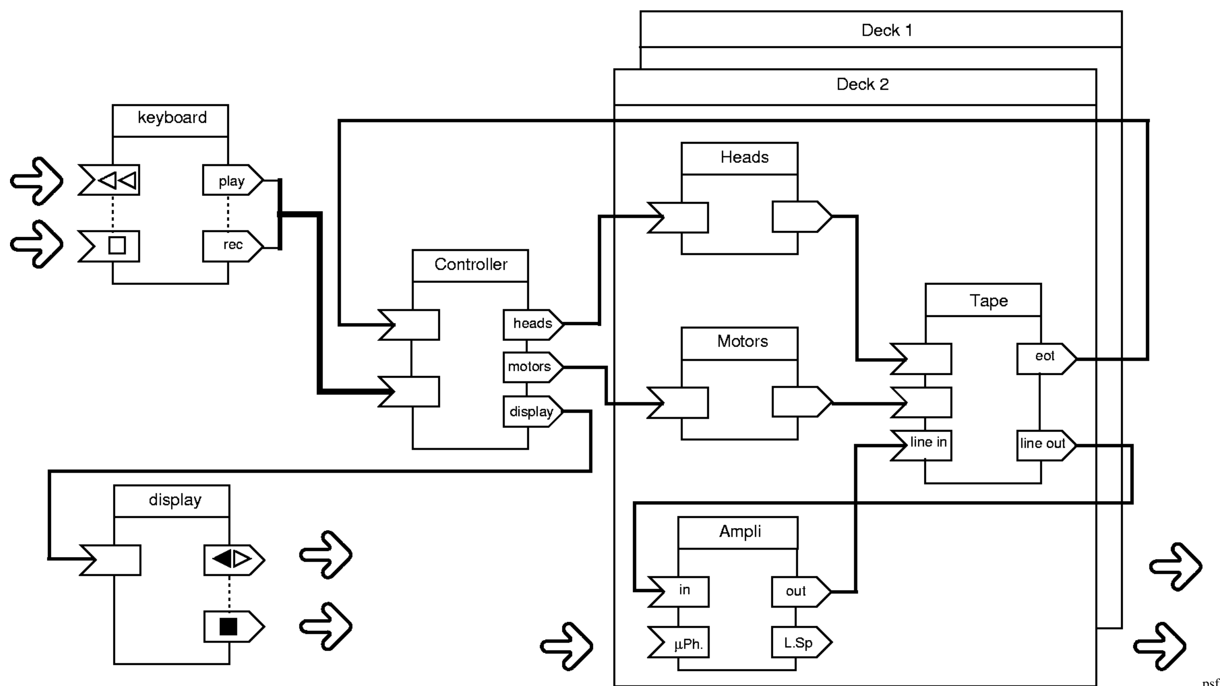

FIG. 5 – Connexions entre objets de l'application

grammes ESTEREL. À titre d'exemple, la table 1 contient le code ESTEREL associé au macro-état `Normal_M_Ctrl`. On peut voir les SYNCCHARTS comme une variante graphique de ce langage.

4 Les objets synchrones

Comme nous l'avons indiqué dans les généralités (section 2), l'approche objet a un grand pouvoir d'abstraction. Elle permet d'exprimer *ce qui* change dans un système, sans nécessairement préciser *comment* s'effectuent ces changements. L'approche synchrone des systèmes réactifs apporte quelque chose d'analogue au niveau des comportements (le *quand* des interactions).

Les interactions complexes qui existent entre sous-systèmes parallèles communicants sont représentées par des diffusions instantanées de signaux (abstraction des communications). La composition synchrone peut être définie de façon rigoureuse et possède des propriétés remarquables comme le caractère déterministe de la composition parallèle.

Comme le montre l'exemple du magnétophone, une application peut faire appel à des éléments transformationnels et à des éléments réactifs synchrones. L'approche objets permet d'intégrer les éléments de ces deux domaines *en encapsulant le code synchrone dans des objets*. Les modules synchrones se prêtent bien à cette encapsulation car ils ne communiquent avec l'extérieur que par leurs signaux et se présentent donc comme des boîtes noires.

Classes synchrones

La première étape est donc de transformer un module synchrone en une classe, que nous dirons synchrone. Les instances de cette classe ont le même comportement réactif que le module synchrone d'origine. Toutes les classes synchrones dérivent d'une classe abstraite[6] nommée `Synchronous`, qui définit le protocole minimal de tout objet synchrone. Ce protocole commun permet de définir une machine d'exécution synchrone qui *conserve la sémantique synchrone des modules* dans le langage à objets.

Les objets synchrones peuvent ainsi être interconnectés et communiquer de manière synchrone entre eux. Le comportement d'un tel réseau d'objets est identique à celui d'un programme synchrone composé des modules correspondants. Ceci permet d'utiliser des bibliothèques de classes synchrones pour construire des programmes sans avoir à utiliser un compilateur de langage synchrone, donc sans avoir besoin du code source des modules utilisés. Cette compilation séparée a bien sûr des limites : le graphe orienté d'interconnection des objets doit être acyclique (sauf déclaration explicite de non-dépendance instantanée, comme dans le cas d'un retard), car la résolution des problèmes de causalité nécessite la connaissance de la sémantique des modules, il faut « ouvrir la boîte noire ».

5 Communications entre objets

Les objets classiques communiquent simplement par messages. Les objets synchrones peuvent communiquer par signaux. Il convient de définir des proto-

[6]Une classe abstraite est une classe dont certaines méthodes ne sont pas définies. Une telle classe ne peut donc pas être instanciée. Elle définit les propriétés communes à toutes les classes qui dérivent d'elle.

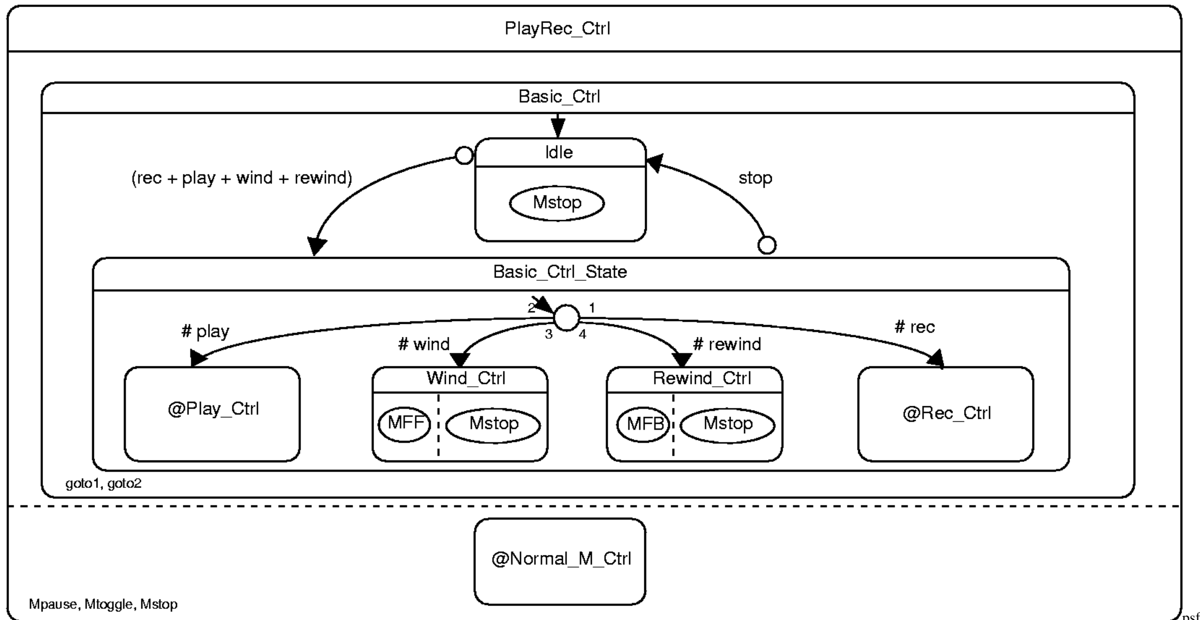

FIG. 6 – Contrôle d'un lecteur-enregistreur

coles nouveaux pour les communications non seulement entre objets synchrones, mais également entre objets synchrones et autres objets.

5.1 Communication synchrone et horloges

La communication synchrone entre objets implique que *ces objets partagent une même notion d'instant*. La machine d'exécution dispose pour cela d'une classe `Clock`. Il est possible d'utiliser plusieurs instances de cette classe (horloges) dans un même programme pour traiter des événements survenant à des échelles de temps différentes. Chaque horloge calcule un ordonnancement des objets qu'elle gère en fonction de leurs interconnexions et les fait réagir à chaque instant dans un ordre qui satisfait leurs dépendances.

Contrairement à ESTEREL où les communications synchrones s'effectuent par « partage de nom » (tous les signaux qui ont le même nom ont la même valeur), les communications synchrones entre objets synchrones se font point à point, plusieurs signaux d'entrée pouvant être connectés à un même signal de sortie. En effet, les signaux de plusieurs instances d'une même classe synchrone portent le même nom, or on ne souhaite généralement pas qu'ils soient tous connectés ensemble, la méthode du partage de nom n'est donc pas exploitable. Les signaux d'entrée sont donc dotés d'une *méthode de connexion* qui prend pour argument un signal de sortie, ce qui permet de déclarer explicitement les connexions entre objets synchrones.

Cette méthode de connexion point-à-point permet d'utiliser les mécanismes de vérification de types de C++ [Str86] pour assurer qu'un signal d'entrée ne peut être connecté qu'à un signal de sortie de même type.

5.2 Communications asynchrones

Pour que les objets synchrones puissent communiquer avec des objets classiques, ou avec des objets synchrones situés sur une autre horloge, un mécanisme de communication asynchrone est indispensable. Les communications vers l'extérieur de l'horloge ne posent pas de problème puisqu'il est possible de consulter la valeur des signaux de sortie d'un objet synchrone sans le perturber. Par contre, les communications de l'extérieur vers un objet de l'horloge nécessitent *une phase de synchronisation* : il faut construire un événement synchrone à partir d'un ou de plusieurs événements asynchrones.

La transformation d'événements asynchrones en événements synchrones est une tâche non triviale qui est confiée à des *objets d'interface* [Bou93]. Ces objets synchrones sont sensibles à des événements asynchrones grâce à certaines de leurs méthodes, et produisent des événements synchrones pour les autres objets de l'horloge. Ils peuvent être écrits en C++ ou dans un langage synchrone. Dans ce dernier cas, aucune hypothèse ne doit être faite sur la simultanéité des entrées puisque cette notion n'a pas de sens pour des événements asynchrones.

5.3 Dynamicité

Nous avons vu que les objets synchrones peuvent être connectés entre eux et communiquer de manière synchrone sous le contrôle d'une horloge. Pour un tel groupe d'objets, il est toujours possible d'écrire un programme équivalent dans un langage synchrone et d'utiliser des outils de preuve pour vérifier certaines propriétés. L'approche objet apporte en outre la réutilisabilité et la souplesse d'intégration. Une fois qu'un module est traduit en une classe C++, rien n'interdit de créer ou de détruire dynamiquement des instances de cette classe au cours de l'exécution du pro-

gramme. Ceci impose de pouvoir changer dynamiquement les connexions entre objets synchrones de façon à pouvoir communiquer avec les nouveaux objets, ou de rediriger les communications avec des objets qui n'existent plus.

On passe alors à l'ordre supérieur puisqu'un événement ne déclenche plus uniquement la réaction d'un objet, mais peut aussi déclencher une « réaction d'horloge », c'est-à-dire une modification du programme synchrone équivalent au groupe d'objets.

La dynamicité a l'inconvénient de supprimer l'existence d'un programme synchrone équivalent à une horloge. Il n'est donc plus possible d'utiliser les outils de preuve classiques pour vérifier des propriétés. Mais elle permet d'exprimer des comportements qui ne peuvent pas être écrits dans un langage synchrone tout en bénéficiant des avantages de l'approche synchrone pour décrire les constituants et le contrôle de la dynamicité.

La dynamicité peut être utilisée dans des systèmes où le nombre d'objets n'est pas connu a priori. Des exemples d'utilisation de la dynamicité sont présentés dans la thèse de F. Boulanger [Bou93].

6 Environnement de développement

Pour mettre en œuvre la programmation synchrone par objets, nous développons un environnement de programmation. Cet environnement intègre différents logiciels dédiés aux approches synchrones : éditeurs, compilateurs, simulateurs, générateurs de code et outils de preuves.

6.1 Éditeurs

Sans négliger les éditeurs de texte classiques comme EMACS, nous avons été amenés à développer nos propres éditeurs de machines à états et de GRAFCET vu comme un modèle synchrone graphique ([AG94]). L'éditeur de SYNCCHARTS est encore à créer.

6.2 Compilateurs

Les comportements réactifs peuvent être indifféremment exprimés par des langages synchrones (LUSTRE ou ESTEREL) ou des formalismes graphiques (GRAFCET). Ils sont compilés en code intermédiaire commun OC par les compilateurs ESTEREL, LUSTRE et notre propre compilateur GRAFCET [AG94].

La transformation d'un module synchrone en classe C++ est réalisée automatiquement à partir de son code oc grâce à l'outil OCC++ [Bou93], qui peut aussi générer un fichier de description du module (interface, fichiers source...) en MDL. MDL est un langage de description et de définition de modules synchrones qui permet de créer de nouveaux modules par assemblage de modules existants. L'outil associé, MDLC (MDL Compiler), produit une classe C++ synchrone à partir de sa description en MDL.

MDL permet aussi de définir de nouveaux modules par héritage. La classe C++ produite est alors une classe dérivée de la classe correspond au module de base.

La machine d'exécution synchrone se présente sous la forme d'une bibliothèque de classes, libSync, qui fournit tout l'environnement nécessaire aux classes synchrones produites par OCC++ ou MDLC.

6.3 Simulateurs

A tous les niveaux du développement, les comportements réactifs peuvent être aisément testés à l'aide de simulateurs à interface graphique basés sur TCL-TK [Ous94] et construits automatiquement à partir des objets réactifs.

Le déverminage des programmes est également facilité par les exécutions interactives de modules avec *remontées dans le source* disponibles pour ESTEREL (simulateur XES) et GRAFCET [AG96].

6.4 Outils de Preuves

Un programme synchrone LUSTRE, ESTEREL, GRAFCET peut être compilé sous forme d'automates ou de systèmes d'équations. La première forme permet d'utiliser les logiciels AUTO [dV89], MEC [ABC94] et TEMPEST [JPO95]. La deuxième forme est exploitée avec des programmes écrits en LUSTRE ou GRAFCET qui peuvent être compilés en format d'entrée pour le vérificateur de modèles BAC [Hal94, AG96]. On teste alors des propriétés de sûreté.

7 Conclusion

La combinaison des approches par objets et des modèles de systèmes réactifs est une voie qui semble prometteuse et qui a déjà donné lieu à quelques travaux (méthode ROOM [SGW94], modèles O-Charts [HaG94] et Objets Réactifs [BDS95]). Nous nous inscrivons résolument dans cette perspective.

Le premier avantage de cette approche est certainement le « confort » de modélisation des systèmes complexes. La vue par objets s'appuie sur des méthodes largement utilisées du type OMT et la description du comportement bénéficie de la sémantique rigoureuse du modèle synchrone.

Pour profiter pleinement de ce confort, il est indispensable de disposer d'une chaîne complète de traitement. Nous l'avons pratiquement réalisée. En particulier les outils de production de code sont disponibles : le langage cible est C++, devenu une norme de fait dans l'industrie.

Cet environnement ne serait pas complet s'il n'intégrait pas des simulateurs et des passerelles vers des outils de preuves formelles. Dans ce but, nous avons assuré l'interface avec des vérificateurs de modèles de diverses origines.

Nous sommes convaincus que l'approche que nous esquissons ici contribuera à l'amélioration de la *qualité* des logiciels indispensable pour les systèmes temps réel critiques. C'est dans ce domaine que nous irons chercher les applications futures.

Références

[ABC94] A. Arnold, D. Begay, and P. Crubillé. *Construction and Analysis of Transition Systems with MEC*, volume 3 of *AMAST Series in Computing*. World Scientific, Singapore, 1994.

[AG94] C. André and D. Gaffé. Coopération GRAFCET/ESTÉREL. In *Colloque AGI'94*, pages 221–224, Poitiers, Juin 1994. Association AGI.

[AG96] C. André and D. Gaffé. GRAFCET et environnements synchrones. In *Modélisation des Systèmes Réactifs*, Brest, Mars 1996. Afcet.

[And95] Charles André. SYNCCHARTS : a Visual Representation of Reactive Behaviors. Technical Report RR 95–52, I3S, Sophia-Antipolis, France, October 1995.

[AP93] C. André and M-A. Péraldi. Effective implementation of ESTEREL programs. In 5th *Euromicro Workshop on Real-Time Systems*, pages 262–267, Oulu (Finland), June 1993. IEEE.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9) :1270–1282, September 1991.

[BDS95] F. Boussinot, G. Doumenc and J.B. Stefani. Reactive Objects. Technical Report (to appear), INRIA, Sophia-Antipolis, France, October 1995. (http ://cma.cma.fr.RC/rc-project.html)

[Ber92] G. Berry. Preemption in concurrent systems. *Proc FSTTCS, Lecture notes in Computer Science*, 761 :72–93, 1992.

[Bou93] F. Boulanger. Intégration de Modules Synchrones dans la Programmation par Objets. Thèse de Doctorat, Supélec / Université de Paris-sud, décembre 1993.

[dV89] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, October 1989.

[Hal94] N. Halbwachs. BAC : A boolean automaton checker. Technical report, VERIMAG, Montbonnot (France), February 1994.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems in logic and models of concurrent systems. *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, 13 :477–498, 1985.

[HaG94] D. Harel and E. Gery. Executable Objects Modeling with Statecharts. Technical Report CS94–20, Weizmann Institute of Science, September 1994 (rev. August 1995).

[JPO95] Lalita J. Jagadeesan, Carlos Puchol, and James Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunication software. Liege, Belgium, July 1995. Conference on Computer Aided Verification (CAV'95).

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, 1991.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward *Real-Time Object-Oriented Modeling*. J. Wiley Publ., 1994.

[Str86] B. Stroustrup. *The* C++ *Programming Language*. Addison Wesley Publishing Company, 1986.