Objets réactifs pour le développement modulaire du contrôle et des traitements

Frédéric Boulanger[*] — **Guy Vidal-Naquet**[*,**]

[*] *Supélec – Service Informatique*
Plateau de Moulon
F-91192 Gif-sur-Yvette cedex

[**] *Laboratoire de Recherche en Informatique*
CNRS – Université Paris Sud
F-91405 Orsay cedex

Email : {Frederic.Boulanger, Guy.Vidal-Naquet}@supelec.fr

RÉSUM'E. *Nous montrons ici comment une séparation claire du contrôle et des traitements dans une application permet d'augmenter la portée des outils de vérification du modèle réactif synchrone et des mécanismes de réutilisation de l'approche objet. Nous présentons une approche et des outils qui autorisent cette séparation en permettant l'intégration du code généré pour le contrôle et pour les traitements dans le cadre de la plate-forme Ptolemy.*

ABSTRACT. *In this paper, we show how a clear separation between the control and the processing parts of an application improves the efficiency of both the proof tools of the synchronous reactive model and the reuse mechanisms of the object-oriented approach. We present an approach and some tools that enable this separation by allowing the integration of the code produced for the control and for data processing in the Ptolemy framework.*

MOTS-CLÉS : *Systèmes réactifs synchrones, objets, programmation multi-paradigmes, Ptolemy, contrôle-commande*

KEY WORDS : *Synchronous reactive systems, objects, multi-paradigm programming, Ptolemy, control-command*

1. Introduction

En dehors de quelques cas particuliers, une application comporte des traitements (transformations de données) et du contrôle (choix et paramètrage des traitements). Ces deux composantes sont souvent intimement mêlées : considérons par exemple la simple expression `if (v > seuil) then T1 else T2`. Le `if` correspond à du contrôle puisqu'il permet de choisir entre les traitements `T1` et `T2`, par contre, la comparaison `v > seuil` est un traitement dont le résultat est utilisé par le contrôle.

Si l'on souhaite prouver que ce code réagit de façon adéquate lorsque la vitesse `v`

dépasse seuil, le contrôle ne doit utiliser que des booléens, et le traitement que constitue cette comparaison doit être effectué en dehors du module de contrôle, de façon à ne faire apparaître qu'un booléen en argument du if. De plus, si la vitesse doit ensuite rester dans une certaine plage au lieu d'être inférieure à seuil, il suffira de changer le module évaluant la condition : le module de contrôle — qui aura peut-être été validé entre temps — n'aura pas à être modifié. De même, si l'évaluation du critère de contrôle nécessite un traitement complexe, on pourra bénéficier de bibliothèques standards efficaces pour l'implémenter, et l'on n'aura pas à se soucier de ce qu'offre le langage utilisé pour le contrôle lors du choix de l'algorithme de traitement.

Cet exemple met en évidence deux inconvénients majeurs du mélange entre le contrôle et les traitements :

1. la « pollution » du contrôle par les traitements, qui limite la portée des outils de vérification dont disposent certaines approches comme le modèle réactif synchrone. Ceci entraîne de plus une dépendance entre le code de contrôle et le code de traitement, qui oblige à modifier le code de contrôle lorsqu'un algorithme de traitement change ;
2. la « pollution » des traitements par du contrôle limite les possibilités de réutilisation et la constitution de bibliothèques de composants fiables et optimisés. De plus, le contrôle évoluant en général plus rapidement que les traitements, cette dépendance entraîne des interventions inutiles sur le code des traitements.

Nous proposons ici une approche et des outils pour le développement modulaire du contrôle et des traitements. Le but est de rendre explicite le contrôle qui est trop souvent masqué dans les traitements. Par exemple, dans un filtre adaptatif, on distinguera le filtrage proprement dit de la logique d'adaptation des paramètres du filtre.

Un des avantages de cette approche est qu'elle permet de choisir le formalisme le plus adapté à chacune des composantes. Nous utiliserons ici l'approche réactive synchrone [Benveniste & Berry 91] pour le contrôle, et une approche à flots de données synchrones (au sens du traitement du signal[1]) pour les traitements.

Une difficulté majeure lors du développement modulaire du contrôle et des traitements est la production de l'application une fois que le code de chaque partie a été généré. C'est à ce niveau que se situent nos outils, et non au niveau de la séparation automatique du contrôle et des traitements.

Pour les outils d'intégration, une application n'est qu'un réseau de boîtes noires interconnectées, et la sémantique des connexions est la seule information utilisable pour réaliser l'intégration. Il est donc indispensable que chaque module de l'application soit développé selon un formalisme connu des outils d'intégration et pour lequel existe une machine d'exécution.

Si les machines à états finis et les langages synchrones fournissent un formalisme adapté au développement du contrôle, le choix d'un formalisme pour les traitements est plus délicat. Nous avons utilisé l'approche à flots de données synchrones de la plateforme PTOLEMY [Buck et al 91] car il s'agit d'une approche « boîtes noires »

[1] les taux de consommation et de production de données des opérateurs sont dans des rapports constants

assez répandue, notamment dans le domaine du traitement du signal.

Le système PTOLEMY[2] de l'Université de Californie à Berkeley, utilise l'approche objet pour définir un cadre dans lequel viennent s'intégrer différents modèles de calcul appelés « domaines ». L'intérêt de PTOLEMY est qu'il est possible d'utiliser conjointement plusieurs domaines dans la même application, et qu'un domaine de simulation peut avoir un domaine dual de génération de code. Il est ainsi possible de simuler un système, puis de générer le code correspondant en passant au domaine dual.

2. Contrôle et traitements

La distinction entre contrôle et traitements n'est pas aussi nette que le discours précédent peut le faire croire. Ainsi, le « traitement » `s = i ;` peut aussi s'écrire :

```
if (i) s = true ; else s = false ;
```

De même, des structures de contrôle peuvent faire partie d'un traitement, par exemple pour le parcours des lignes et des colonnes d'une matrice.

Cette distinction ne peut donc se faire selon des critères purement syntaxiques. Il faut décider, lors de la conception de l'application, ce qui sera considéré comme du contrôle et ce qui sera considéré comme un traitement. Lorsque l'on souhaite utiliser des outils de vérification pour prouver une propriété d'un module de contrôle, c'est qu'il a déjà été identifié implicitement comme participant au contrôle de l'application, et il ne reste qu'à en extraire tout ce qui n'est pas booléen.

La principale difficulté est d'identifier le contrôle qui est enfoui dans un algorithme de traitement. Considérons par exemple un système de mise au point automatique d'une optique. Un signal d'erreur indique l'écart par rapport à la mise au point idéale, et des actionneurs permettent de déplacer des lentilles pour former l'image dans le plan des capteurs. On peut exprimer la loi de commande des actionneurs en fonction du signal d'erreur sous forme d'un filtre et l'implémenter comme un traitement. Mais rapidement, on devra enrichir ce traitement pour tenir compte de différentes conditions d'utilisation, et créer ainsi de nombreux modes de fonctionnement. De ce fait, on aura construit un traitement complexe mais peu structuré, avec des boucles de rétroaction du type *si la dérivée de l'erreur dépasse X, le 6^e coefficient du filtre Z passe à Y*, éparpillées dans tout le code.

Un critère d'identification du contrôle est qu'il modifie la manière dont des données sont traitées. Ainsi, dans notre exemple, le module de contrôle sera en charge du choix de la loi de commande (et de ses paramètres), du choix du capteur et du mode de mise au point. Pour prendre des décisions, le module recevra les informations nécessaires, mais sans savoir d'où elle viennent. Il sera alors aussi facile de « forcer » un mode sélectionné par l'utilisateur que de choisir le meilleur mode en fonction du résultat des traitements (mode automatique). D'autre part, les traitements, une fois débarrassés du contrôle propre à l'application dans laquelle ils sont utilisés, deviennent standards, ce qui permet de choisir leur implémentation dans des bibliothèques.

---

[2] http ://ptolemy.eecs.berkeley.edu

La stabilité et la portée générale des algorithmes de traitement (tris, calcul matriciel, traitement du signal) fait que l'on trouve principalement des bibliothèques de traitements. Mais le contrôle se prête aussi à la réutilisation de certaines structures simples comme le montre la bibliothèque standard C++ [Stroustrup 97].

Notre approche consiste donc à insérer dans le développement de l'application une phase explicite d'identification des modules de contrôle dont seront ensuite extraits tous les traitements. Après cette phase, les modules de contrôle ne reçoivent et ne produisent que des booléens (ou des signaux purs, si on utilise ESTEREL). Il devient ainsi possible de vérifier formellement des propriétés de ces modules grâce aux outils de l'approche réactive synchrone. Ceci aurait été impossible si le module de contrôle avait pris des décisions fondées sur des traitements de types de données aussi simples que les entiers.

3. Modularité et séparation contrôle-traitements

De même que la modularité nous amène à distinguer interface et implémentation, le développement séparé du contrôle et des traitements nous amène à distinguer interface de contrôle et interface de traitement, ainsi qu'implémentation du contrôle et implémentation des traitements.

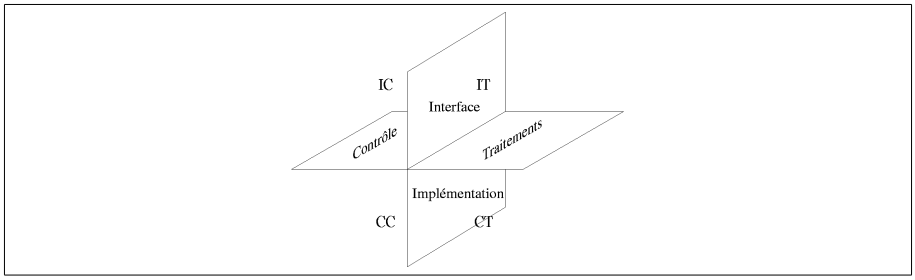

Figure 1. *Modularité, contrôle et traitements*

La figure 1 montre les quatre types d'entités que nous devons manipuler pour passer de la spécification d'une application à son code.

L'interface de contrôle et l'interface de traitement doivent définir les interactions possibles entre le contrôle et les traitements. Les implémentations correspondantes fournissent les machines d'exécution autorisant les interactions entre les modules de même nature.

Le passage de la zone interface de contrôle (IC) à la zone code (implémentation) du contrôle (CC) correspond à l'implémentation de la sémantique utilisée pour le contrôle (réactif synchrone par exemple). Il en est de même pour le passage de la zone interface de traitement (IT) à la zone code des traitements (CT).

Il faut ensuite intégrer les deux codes pour obtenir l'application, ce qui revient à implémenter la sémantique des interactions entre contrôle et traitements. Cette phase

peut intervenir lors du passage de l'interface à l'implémentation, ce qui suppose que les machines d'exécution du contrôle et des traitements aient été conçues pour fonctionner ensemble — c'est ce que nous appellerons l'intégration directe. L'autre solution consiste à ajouter un adapteur entre les deux machines d'exécution après la génération de code, ce que nous appellerons la co-intégration. Dans Ptolemy, un tel adapteur est appelé « trou de vers » par analogie avec les structures cosmiques qui permettraient à deux univers de communiquer.

4. Intégration directe du contrôle et des traitements

À l'heure actuelle, nous savons implémenter la sémantique des interactions entre contrôle et traitements lors du passage de l'interface à l'implémentation, qui se fait directement de la zone IC à la zone CT pour le contrôle. Autrement dit, chaque module de contrôle écrit en ESTEREL ou en LUSTRE est transformé en module de traitement intégrant sa propre machine d'exécution réactive synchrone.

Un inconvénient de cette approche est que si deux modules de contrôle doivent communiquer, il le feront à travers la machine d'exécution des traitements, c'est-à-dire avec la sémantique des communications pour les traitements. En l'occurrence, si les traitements sont décrits dans le domaine SDF de PTOLEMY, qui a une sémantique à flots de données synchrones adaptée au traitement du signal, les boucles instantanées entre modules de contrôle seront interdites.

Lorsque la modularité du contrôle est traitée ailleurs, par exemple en compilant plusieurs nœuds LUSTRE ou plusieurs modules ESTEREL, l'outil d'intégration ne traite qu'un seul module de contrôle et ce problème est évacué. C'est aussi la seule méthode utilisable si l'on souhaite prouver formellement des propriétés du contrôle, puisque les outils de preuve doivent disposer de l'intégralité du code de contrôle pour effectuer une vérification.

Un autre inconvénient de l'intégration directe est que le contrôle ne peut pas agir sur la machine d'exécution du domaine hôte. Si plusieurs traitements doivent être activés à tour de rôle selon une décision du contrôle, ils devront fonctionner en permanence et le contrôle ne fera que sélectionner le résultat de l'un ou de l'autre. Ceci entraîne l'exécution de traitements inutiles qui ne peut être évitée que s'il est possible d'indiquer au domaine hôte que certains modules de traitement doivent être ignorés par la machine d'exécution en fonction des signaux émis par le contrôle.

L'intégration directe est effectuée en traduisant le code OC produit par LUSTRE ou ESTEREL en « étoile » (entité opératoire de base pour PTOLEMY) du domaine utilisé pour les traitements. Le traducteur OCPL[3] que nous avons développé traduit ainsi le code OC d'un module de contrôle en un module qui, bien qu'implémentant du code synchrone, peut fonctionner sur la machine d'exécution des traitements. Pour parvenir à ce résultat, OCPL intègre au code généré une machine d'exécution qui effectue la conversion entre les signaux synchrones et les signaux de traitement, et assure l'activation du code de contrôle en fonction des informations reçues de la machine

[3] disponible par ftp anonyme sur ftp ://ftp.supelec.fr/pub/cs/distrib/

d'exécution des traitements.

Dans le cas des domaines de génération de code, la machine d'exécution intégrée par OCPL ne doit pas fournir des services au module de contrôle, mais générer du code qui lui rendra ces services lors de l'exécution du programme.

4.1. *Exemple : un régulateur de vitesse*

Nous illustrons l'intégration directe avec l'exemple d'un régulateur de vitesse pour automobile. Ce régulateur a un mode initial transparent dans lequel la pédale d'accélérateur pilote les gaz, un mode de régulation de vitesse et un mode d'accélération constante qui ne peuvent être actifs que si la vitesse est comprise entre 50 et 130 km/h, et un mode transparent temporaire qui permet d'accélérer volontairement dans un des modes de régulation en retournant à ce mode dès que la pédale d'accélérateur est relâchée. Pour des raisons de sécurité, on souhaite vérifier formellement que l'on est en mode transparent dès que l'on appuie sur la pédale de frein.

Le contrôle doit activer les différents modes selon les commandes du conducteur, la position des pédales et la vitesse. La séparation du contrôle et des traitements nous amène à considérer la comparaison de la vitesse à 50 et à 130, ainsi que la comparaison de la position des pédales à zéro comme des traitements. Le contrôle travaille ainsi sur des signaux booléens du type appui sur la pédale de frein ou vitesse minimale atteinte.

Le code de contrôle est écrit en ESTEREL, et le code OC correspondant est transformé en étoile SDF ou CGC par OCPL. Les traitements sont conçus dans le domaine SDF/CGC de PTOLEMY et sont pilotés par l'étoile générée pour le contrôle.

Bien que d'autres outils de vérification formelle existent, l'outil CHECKBLIF fourni avec l'environnement de validation standard d'ESTEREL XEVE suffit pour vérifier que l'appui sur la pédale de frein place systématiquement le système en mode transparent. Le principe de la vérification consiste à écrire un module ESTEREL qui émet un signal ERREUR dès que la propriété n'est pas vraie. CHECKBLIF calcule les états atteignables de l'automate correspondant et annonce que le signal ERREUR n'est jamais émis, quelque soient les entrées du module.

Si nous avions fait parvenir au contrôle la position de la pédale de frein au lieu d'un booléen indiquant qu'elle est enfoncée, CHECKBLIF n'aurait pas pu conclure car il ignore la sémantique de la comparaison entre entiers et ne peut explorer toutes les combinaisons possibles de valeurs entières pour les entrées du module de contrôle.

5. Co-intégration

Contrairement à l'intégration directe, la co-intégration permet de traiter la modularité du contrôle au niveau de l'intégrateur. Elle permet aussi aux machines d'exécution d'interagir d'égale à égale, alors qu'avec l'intégration directe, la machine d'exécution du contrôle n'interagissait avec celle des traitements qu'en tant que module de traite-

ment.

Il existe deux domaines de PTOLEMY adaptés à la conception du contrôle : Finite State Machines (FSM) et Synchronous Reactive (SR). Le domaine FSM permet de définir des automates dont les états peuvent contenir un autre automate ou un système SDF. Il est donc possible de construire des automates hiérarchiques et d'activer du code de traitement SDF uniquement lorsqu'un automate est dans l'état contenant ce code.

Le domaine SR [Edwards 97] permet de concevoir graphiquement des systèmes réactifs synchrones par assemblage de composants. Dans ce domaine, un signal peut être non déterminé, absent ou présent. On ne peut accéder à la valeur d'un signal que s'il est présent. Chaque étoile de SR implémente une fonction strictement croissante au sens où, si on l'appelle avec des entrées plus déterminées, elle détermine un plus grand nombre de ses sorties. Le principe de SR est donc d'évaluer les étoiles d'un système jusqu'à ce que tous les signaux soient déterminés. On distingue deux types d'étoiles : les étoiles strictes, qui ne peuvent déterminer leurs sorties que lorsque toutes leurs entrées sont déterminées, et les étoiles non-strictes, qui peuvent déterminer certaines de leurs sorties même lorsque toutes leurs entrées ne sont pas déterminées.

Les étoiles non-strictes permettent d'utiliser des boucles instantanées dans un système SR et évitent d'être limité à la mise en cascade de modules synchrones ou aux bouclages avec retard. SR semble donc être un bon candidat pour la spécification du contrôle dans PTOLEMY. En collaboration avec Thomson-CSF Optronique, nous avons développé le domaine dual SRCGC de génération de code C, afin de pouvoir obtenir le code complet d'une application dont le contrôle serait développé en SR et les traitements en SDF.

5.1. *D'*ESTEREL *à SR : le traducteur SSCPL*

Si SR permet d'assembler facilement des composants de base de manière synchrone, il faut toutefois construire ces composants de base, et C++ n'est pas le meilleur langage pour cela. Nous avons donc cherché à traduire des modules ESTEREL en étoiles SR ou SRCGC. La première étape a été d'ajouter ces deux domaines comme cibles pour OCPL. Malheureusement, le format OC ne permet de créer que des étoiles strictes qui ne peuvent être utilisées dans des boucles instantanées.

Nous nous sommes donc tournés vers le format SSC produit par le compilateur ESTEREL. Ce format distingue la partie combinatoire et les registres qui codent l'état du système. Il est alors possible d'évaluer partiellement la partie combinatoire en fonction des entrées connues et de déterminer ainsi certaines sorties. Le nouvel état du système n'est évalué qu'à la fin de l'instant, lorsque toutes les sorties de la partie combinatoire sont déterminées. Le traducteur SSCPL[4] construit une étoile SR ou SRCGC capable d'évaluer partiellement la combinatoire d'un circuit SSC correspondant à un module ESTEREL.

[4]SSCPL est en cours de réécriture et sera disponible prochainement au même endroit qu'OCPL

6. Conclusion

À l'heure actuelle, nous savons intégrer des modules de contrôle écrits en LUSTRE ou en ESTEREL à des traitements écrits en SDF, et ceci aussi bien en simulation (SDF) qu'en génération de code C (CGC).

Nous savons aussi construire des étoiles SR à partir de code ESTEREL et simuler (SR) ou générer du code C (SRCGC) pour des applications comportant peu ou pas de traitements (ces traitements sont alors effectués dans des étoiles SR ou SRCGC).

Notre but est de pouvoir exploiter pleinement les possibilités de SR pour la spécification du contrôle et celles de SDF pour les traitements. Malheureusement, l'architecture de génération de code de PTOLEMY ne permet actuellement pas d'utiliser conjointement le code généré par CGC et celui généré par SRCGC.

En collaboration avec Xavier Warzee, de Thomson CSF Optronique, nous développons un domaine de génération de code dual de FSM. La sémantique de FSM est légèrement modifiée afin qu'un automate puisse être utilisé comme une étoile non-stricte dans SRCGC. La co-intégration du contrôle et des traitements est réalisée en associant des traitements exprimés en SDF/CGC aux états d'automates FSM. Ces automates sont eux-mêmes utilisés dans le domaine SRCGC conjointement à d'autres modules de contrôle écrits en ESTEREL.

L'absence de traitements dans la partie contrôle permettrait de générer une représentation du contrôle dans un format acceptable par les outils de vérification. SR permet en effet de construire des systèmes réactifs synchrones mais ne fournit aucun moyen de vérifier formellement que le système obtenu est causal ou qu'il vérifie certaines propriétés. L'utilisation d'outils externes semble donc la meilleure solution pour pallier ce manque.

L'aboutissement de ces travaux permettra, grâce à une meilleure séparation du contrôle et des traitements, de limiter les dépendances entre ces deux composantes d'une application, d'augmenter la portée des outils de vérification formelle, de permettre une plus grande réutilisation de composants standards pour les traitements, et de constituer des bibliothèques de modules de contrôle.

Bibliographie

[Benveniste & Berry 91] BENVENISTE A. et BERRY B., *The Synchronous Approach to Reactive and Real-Time Systems.* Proceedings of the IEEE, vol. 79, n° 9, September 1991.

[Buck et al 91] BUCK J.T. et HA S. et LEE E.A., *Ptolemy : A Mixed-Paradigm Simulation/Prototyping Platform in C++.* C++ At Work Conference, Santa Clara CA, November 1991.

[Edwards 97] EDWARDS S.A., *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis UCB/ERL M97/31, University of California, Berkeley, 1997.

[Stroustrup 97] STROUSTRUP B., *C++Programming Language, 3rd Edition*. Addison Wesley, 1997.