Objets et Programmation Synchrone

Frédéric Boulanger

Guy Vidal-Naquet

L'approche objet apporte la facilité de structuration, d'abstraction, de réutilisation et d'évolution. L'approche réactive synchrone permet la description formellement rigoureuse des systèmes réactifs (e.g. systèmes de contrôle-commande) , ce qui autorise des preuves de correction logique.
*Cet article propose une combinaison de ces deux approches et introduit la notion d'*objet synchrone*. 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 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.*

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* 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* 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 pour la manipulation des données, et nécessitent une *machine d'exécution* pour exécuter le code produit par leur compilateur.

Les recherches présentées ont pour but de *combiner approches synchrones et approches par objets*, afin de pouvoir programmer des *systèmes complexes* où interviennent des aspects réactifs et transformationnels (par exemple, la production et la distribution d'énergie).

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 *encapsulées*, c'est-à-dire qu'on ne peut y accéder 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 des 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) apparaissent trois modèles différents :

1. le *modèle objet* proprement dit, souvent appelé également *modèle de classes*, qui décrit les objets du système, leur interface d'utilisation, et les relations (statiques) qui existent entre eux;

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.

Les objets synchrones

L'approche synchrone des systèmes réactifs est similaire à l'approche objet pour les systèmes transformationnels en ce qu'elle permet de décrire des systèmes complexes en combinant une multitude de comportement temporels simples. 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.
Une application peut faire appel à des éléments transformationnels et à des éléments réactifs synchrones (e.g. régulation et supervision de cette régulation). 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. Il est alors possible de combiner approche objet et approche synchrone.

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[1] 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 de constituer des bibliothèques de classes synchrones et d'obtenir une certaine forme de réutilisabilité du code synchrone. Cette compilation séparée est toutefois limitée par la résolution des problème de causalité qui nécessite la connaissance de la sémantique des modules et oblige donc à « ouvrir les boîtes noires ».

1. 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.

Communications entre objets

Les objets classiques communiquent simplement par messages (ou activation de méthodes). Les objets synchrones peuvent communiquer par signaux. Il convient de définir des protocoles nouveaux pour les communications non seulement entre objets synchrones, mais également entre objets synchrones et autres objets.

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.
Les connexions entre objets se font point à point grâce à la méthode de connexion dont sont dotés les signaux d'entrées. Ceci permet d'utiliser les mécanismes de vérification de types de C++ pour assurer qu'un signal d'entrée ne peut être connecté qu'à un signal de sortie de même type.

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.

Imaginons par exemple que pour déclencher l'enregistrement sur un magnétophone, il faille appuyer simultanément sur les touches $\boxed{\text{Record}}$ et $\boxed{\text{Play}}$. Le contrôleur de la platine est donc sensible aux événements {$\boxed{\text{Play}}$} et {$\boxed{\text{Record}}$, $\boxed{\text{Play}}$}, mais pas à l'événement {$\boxed{\text{Record}}$} qui ne doit jamais se produire. Dans notre application, l'enfoncement d'un bouton de la platine est un événement asynchrone puisqu'il est déclenché dans le monde physique. Il faut transformer ces événements asynchrones en événements synchrones pour le contrôleur. Ainsi, l'arrivée de l'événement asynchrone $\boxed{\text{Record}}$ ne générera pas d'événement synchrone. L'arrivée successive de $\boxed{\text{Record}}$ et de $\boxed{\text{Play}}$ générera l'événement synchrone {$\boxed{\text{Record}}$, $\boxed{\text{Play}}$}. L'événement asynchrone $\boxed{\text{Play}}$ peut poser un problème : s'agit-il d'une demande de lecture ou d'une demande d'enregistrement dont l'événement $\boxed{\text{Record}}$ n'est pas encore arrivé? Une solution est d'attendre un certain temps avant de générer l'événement synchrone. Si l'événement $\boxed{\text{Record}}$ survient pendant ce laps de temps, on générera {$\boxed{\text{Record}}$, $\boxed{\text{Play}}$}, sinon, on générera {$\boxed{\text{Play}}$}. Ce problème est semblable à celui de la détection du double-clic dans les interfaces graphiques.

La transformation d'événements asynchrones en événements synchrones est une tâche non triviale qui est confiée à des *objets d'interface*. 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.

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 programme. 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.

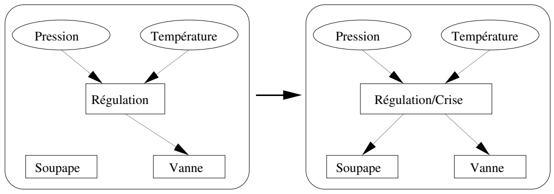

Fig. 1 – *Reconfiguration : dynamicité des objets*

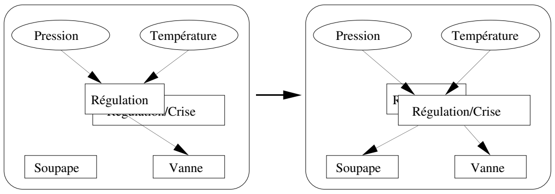

Fig. 2 – *Reconfiguration : dynamicité des communications*

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. En effet, la solution classique qui consiste à déclarer autant d'objets qu'il peut y en avoir au maximum consomme inutilement des ressources, ce qui peut amener le système à ne pas pouvoir traiter une situation alors qu'une meilleure utilisation du CPU ou de la mémoire permettrait de la traiter.

Une autre utilisation de la dynamicité est la reconfiguration du système pour assurer une qualité de service minimale. Lorsque certaines conditions sont détectées, le système change de configuration soit en modifiant uniquement les connexions entre objets, le temps de reconfiguration est alors très court, soit en remplaçant certains objets, ce qui est plus coûteux en temps, mais requiert moins de mémoire.

Par exemple, sur la figure 1, un système de contrôle se reconfigure en remplaçant un module de régulation par un module de régulation en conditions critiques lorsque certaines conditions de température et de pression sont détectées. Sur la figure 2, les deux modules de régulation

existent en permanence, et la reconfiguration se fait en modifiant uniquement les connexions.
La condition déclenchante d'une reconfiguration peut être par exemple la détection d'un dysfonctionnement (application à la sûreté de fonctionnement), ou la violation de l'hypothèse synchrone (le temps de réaction des objets n'est plus négligeable devant la période des événements), auquel cas, on remplacera les objets les plus lents par des objets assurant un service dégradé similaire plus rapide.

Transformations de graphes

Une application purement synchrone et monohorloge peut être décrite par un graphe orienté dont les sommets sont les objets synchrones et dont les arêtes représentent les connexions entre les signaux des objets. On peut modifier formellement ce graphe pour obtenir une nouvelle application synchrone. De telles transformations ont été étudiées pour dissocier le développement d'une application et la prise en compte de contraintes de sûreté de fonctionnement : à partir du graphe de l'application initiale, on applique des transformations (n-plication d'un objet, ajout de voteurs, de détecteurs d'incohérences etc.) pour obtenir le graphe de l'application intégrant les contraintes de sûreté de fonctionnement. Un langage de configuration permet d'exprimer les transformations du graphe, et il est possible de constituer des bibliothèques de transformations usuelles (par exemple, le remplacement d'un objet par trois exemplaires de cet objet avec des voteurs sur les sorties). Une application peut ainsi être adaptée à différents environnements en lui appliquant les fichiers de configuration correspondants. De telles transformations ont été étudiées dans le cadre d'un contrat avec la DGA.

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 : compilateurs, simulateurs, générateurs de code et outils de preuves.

Les comportements réactifs peuvent être indifféremment exprimés par des langages synchrones (LUSTRE ou ESTEREL) ou des formalismes graphiques (ARGOS). Ils sont compilés en code intermédiaire commun OC par leurs compilateurs respectifs.
La transformation d'un module synchrone en classe C++ est réalisée automatiquement à partir de son code oc grâce à notre outil OCC++, 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 correspondant au module de base.
Le langage de configuration du § est une extension de MDL.
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.

Les comportements réactifs peuvent être testés à l'aide de simulateurs à interface graphique basés sur TCL/TK et construits automatiquement par MDLC à partir des fichiers de description.

Nos outils s'appuyant sur le format commun OC, ils s'interfacent aisément avec les outils de preuves réalisés pour les langages synchrones.

Conclusion

La combinaison des approches par objets et des modèles de systèmes réactifs est une voie qui semble prometteuse.
Son premier avantage 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.
La chaîne de développement quasiment complète dont nous disposons pour cette approche permet de bénéficier de ce confort pour l'implémentation, le langage cible étant C++.
La plateforme de conception et de simulation PTOLEMY, développée à l'Université de Californie à Berkeley, permet d'utiliser conjointement plusieurs formalismes. Nous travaillons actuellement à son utilisation pour développer notre approche.