HAL
open science

Intégration de la synthèse de contrôleurs discrets dans un langage de programmation

Gwenaël Delaval, Eric Rutten, Hervé Marchand

To cite this version:

Gwenaël Delaval, Eric Rutten, Hervé Marchand. Intégration de la synthèse de contrôleurs discrets dans un langage de programmation. Modélisation des Systèmes Réactifs (MSR'11), Nov 2011, Lille, France. inria-00629104

HAL Id: inria-00629104

https://inria.hal.science/inria-00629104

Submitted on 5 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Intégration de la synthèse de contrôleurs discrets dans un langage de programmation

Gwenaël Delaval* — **Eric Rutten*** — **Hervé Marchand****

* *LIG (UJF, INRIA) Grenoble*
655 av. de l'Europe, MONTBONNOT, 38334 ST ISMIER
gwenael.delaval@inria.fr, eric.rutten@inria.fr

** *INRIA Rennes*
263 avenue du Général Leclerc, 35042 RENNES
herve.marchand@inria.fr

RÉSUMÉ. *Nous définissons un langage de programmation mixte impératif / déclaratif : des contrats déclaratifs sont imposés sur des comportements décrits impérativement. La définition du langage fait appel à la synthèse de contrôleurs discrets (SCD), une technique formelle issue de l'automatique des systèmes à événements discrets. Nous visons le domaine d'application des systèmes de calcul adaptatifs et reconfigurables: notre langage permet de programmer des contrôleurs d'adaptation en boucle fermée, pour une exécution flexible des fonctionnalités, en réponse à des changements dans l'environnement ou les ressources de calcul. Nous faisons une présentation synthétique du langage, de sa sémantique et de sa compilation, et nous illustrons son utilisation par un exemple de système robotique.*

ABSTRACT. *We define a mixed imperative/declarative programming language: declarative contracts are enforced upon imperatively described behaviors. We rely on the notion of Discrete Controller Synthesis (DCS), a formal technique stemming from control theory and the supervisory control of discrete event systems. We target the application domain of adaptive and reconfigurable computing systems: our language can serve programming closed-loop adaptation controllers, enabling flexible execution of functionalities w.r.t. changing resource and environment conditions. We give a synthetic presentation of the language, its semantics and compilation, and we illustrate its use with the example of a robot system.*

MOTS-CLÉS : *programmation synchrone, synthèse de contrôleurs discrets, compilation, contrats comportementaux, systèmes adaptatifs / reconfigurables.*

KEYWORDS: *synchronous programming, discrete control synthesis, compilation, behavioral contracts, adaptive / reconfigurable systems.*

1. Introduction

1.1. *Motivations*

Le moindre système embarqué, de nos jours, inclut de nombreuses fonctionnalités, qui sont à la fois indépendantes du point de vue de leur fonctionnalité, et très intriquées, ne serait-ce que du point de vue de l'usage des ressources du système. Dans ce contexte de séparation des préoccupations, il n'est souvent plus possible de concevoir le système dans son intégralité de manière impérative ou fonctionnelle sans compromettre la modularité de cette conception. Il est nécessaire de pouvoir décrire, de manière indépendante, la fonctionnalité principale du système (ou des sous-systèmes qui le compose), et les propriétés additionnelles que ce système doit vérifier ; et ce sans devoir se préoccuper systématiquement de la «programmation» de ces propriétés.

Dans le domaine de l'automatique, discrète ou continue, la *synthèse de contrôleurs* est une méthode permettant, à partir d'une part d'un modèle du système et de son environnement, et d'autre part de propriétés à assurer sur ce système, de construire plus ou moins automatiquement un *contrôleur*, tel que la composition de ce contrôleur avec le système vérifie les propriétés. Ce contrôleur va donc *imposer* des propriétés qui n'étaient pas *a priori* assurées par le système.

Il existe des outils automatiques de synthèse de contrôleurs dans un cadre discret (Marchand *et al.*, 2000). Nous proposons donc d'intégrer la Synthèse de Contrôleurs Discrets (SCD) dans le processus de compilation d'un langage réactif synchrone : BZR [1]. Notre contribution concerne donc la conception de langages de programmation, l'application de la SCD, et le contrôle de systèmes de calcul adaptatifs et reconfigurables.

1.2. *Contribution*

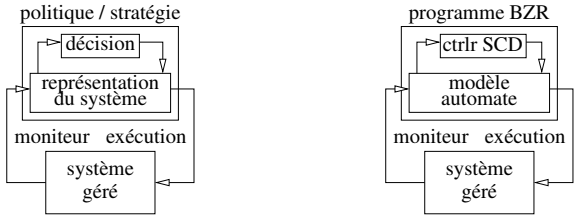
Du point de vue des langages de programmation, nous proposons un processus de compilation qui exploite concrètement une représentation du comportement dynamique du programme. Classiquement, les compilateurs considèrent des propriétés à satisfaire indépendamment des transitions possibles, c'est-à-dire statiques ; nous proposons de considérer les états et les aspects dépendant des traces (c'est-à-dire dynamiques). La notion de compilation comportant une exploration de l'espace d'état est à notre connaissance très peu courante (Wang *et al.*, 2009). La SCD est une opération constructive, au sens où elle calcule non pas un diagnostic de correction, mais une solution correcte. Quelques travaux existent sur son intégration dans le cadre d'un langage de programmation (Altisen *et al.*, 2003). Nous l'associons à un mécanisme de contrats, ce qui la rend plus facile à utiliser par des programmeurs, et favorise le passage à l'échelle par la modularité. De façon symétrique, nous proposons un point de

1. `http://bzr.inria.fr`

vue nouveau sur le principe des contrats : notre langage permet d'imposer le contrats sur des programmes non-déterministes, plutôt que de les vérifier ou prouver corrects.

Concernant la Synthèse de Contrôleurs Discrets, l'application modulaire de la SCD, que nous traitons dans cet article, est fondée sur l'imposition de contrats et l'abstraction de composants ; elle a pour but d'améliorer le passage à l'échelle des techniques de SCD. De plus, l'intégration de la SCD dans un langage de programmation de haut niveau la rend plus largement utilisable dans les systèmes informatiques, notamment via des générateurs de code. Elle permet aussi l'étude de la mise en œuvre des contrôleurs à un niveau plus haut que les automates programmables industriels utilisés dans les automatismes, domaine d'application usuel de la SCD.

Nous nous intéressons au contrôle de systèmes adaptatifs et reconfigurables. Les systèmes embarqués doivent être prédictibles, pour des raisons de sécurité critique. Ils doivent aussi être capables de s'adapter dynamiquement et de se reconfigurer, en réaction à des changements dans leur environnement, ou dans leurs ressources de calcul, ou liés à la tolérance aux fautes. Ceci requiert des capacités d'observation de l'état du système, de décision sur des actions de reconfiguration, sur la base d'une représentation du système, et de capacités d'exécution de ces dernières. Ces fonctionnalités sont assemblées dans une boucle d'adaptation illustrée en Figure 1(a).



(a) Système adaptatif. (b) Programmation BZR.

Figure 1. *Programmation en BZR du contrôle de l'adaptation*

Dans notre approche, nous voulons combiner les deux contraintes de *prédictibilité et adaptativité*. Notre langage de programmation BZR est conçu pour faciliter la conception de boucle de contrôle sûres pour les systèmes adaptatifs, en se fondant sur des techniques de l'automatique discrète. Il permet de séparer les problèmes en spécifiant séparément, comme on le voit en Figure 1(b), d'une part, les comportements possibles des composants à coordonner, sous forme de moèle en automates, et d'autre part, la politique d'adaptation ou les objectifs de contrôle pour leur assemblage. À partir de ces deux spécifications, la SCD peut générer automatiquement, si une solution existe, le contrôleur, qui constitue un composant de décision correct pour les objectifs. Notre contribution est la définition d'un nouveau constructeur de langage réactif synchrone, ajouté au langage HEPTAGON, pour définir des contrats comportementaux à imposer sur un nœud. La sémantique de ce constructeur est définie en termes d'un problème de SCD, où les comportements résultants sont contrôlés de façon à imposer le contrat (Delaval *et al.*, 2011). Nous présentons la mise en œuvre de l'extension du langage,

qui fait intervenir une encapsulation d'un outil de SCD dans la compilation (Delaval *et al.*, 2010a). Nous l'illustrons avec un exemple abstrait à partir de l'étude de cas d'un système robotique (Aboubekr *et al.*, 2011).

La position de notre contribution dans le processus de développement d'un système adaptatif ou reconfigurable est illustrée en Figure 2. Notre langage est utilisé pour spécifier la partie discrète, logique du contrôle, sous forme d'automates. Si elle est explicitement définie dans le langage hôte, cette partie peut être extraite automatiquement de la spécification par compilation ou transformations (comme c'est le cas depuis la langage de parallélisme de donées Gaspard2 (Yu *et al.*, 2010)). Dans ce cas le programmeur n'a pas besoin de connaître les technicités formelles de BZR ou de la SDC. Pour les autres parties des systèmes adaptatifs, concernant les aspect non réactifs liés par exemple aux données, d'autres langages généralistes sont plus appropriés. La partie contrôle est compilée avec une phase de SCD, et du code exécutable est généré vers des langages cibles comme C ou Java. Ce code est alors lié avec le reste du système, avec un interfaçage approprié avec les fonctionnalités des moniteurs et actions d'adaptation fournies par la plateforme d'exécution. De cette façon, la boucle de contrôle discret est en place comme en Figure 1(b). Nous avons des travaux en cours sur des instanciations de ce processus de développement dans différents contextes, et à différents niveaux, depuis les architectures dynamiquement partiellement reconfigurables à base de FPGA (Guillet *et al.*, 2011), en passant par les boucles d'administration autonomique dans les systèmes d'exploitation (de Palma *et al.*, 2010), et les intergiciels à base de composants (Delaval *et al.*, 2010b), jusqu'aux applications de contrôle-commande (Aboubekr *et al.*, 2011).

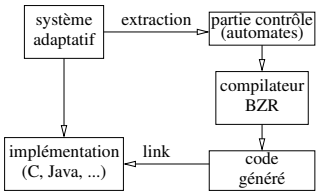


Figure 2. *Processus de développement utilisant BZR*

2. Systèmes réactifs et leur contrôle supervisé

2.1. *Langages réactifs*

Dans un souci d'abstraction et de passage à l'échelle, nous considérons des programmes réactifs synchrones structurés en nœuds, inspiré de LUCID SYNCHRONE (Colaço *et al.*, 2005), consistant en un nom, des variables d'entrée et de sortie représentant des flots de valeurs, et des équations définissant des sorties en fonctions des entrées. Le comportement de base est qu'à chaque pas de réaction, les valeurs du flot d'entrée sont utilisées de façon à calculer les valeurs des flots de sortie pour ce pas.

À l'intérieur des nœuds, ceci est exprimé par des déclarations D, ce qui prend la forme d'équations définissant, pour chaque sortie ou flot local, la valeur que le flot prend, en termes d'une expression sur d'autres flots, éventuellement en utilisant des flots locaux et des valeurs calculées aux pas précédents (auxquelles on se réfère comme des valeurs d'état). Un nœud d'équations simple est illustré en Figure 3, où pour des flots d'entrée a, b, c et d, tous Booléens, un flot de sortie Booléen m est à **true** quand plus de deux des quatre entrées sont à **true**.

```
node morethantwo(a,b,c,d:bool) = (m:bool)
  let
    m = (a and b and (c or d)) or ((a or b) and c and d)
  tel
```

Figure 3. *Nœud d'équations simple*

Un type particulier de nœud que nous considérerons dans cet article est utilisé pour encoder les Automates de Mode, qui donne la possibilité de mélanger programmation équationnelle et programmation plus impérative à base d'automates. Nous considérons des programes exprimés comme des machines de Moore synchrones, avec composition parallèle et hiérarchique. À chaque état d'un automate, un nœud peut être associé, avec des équations, ou un Automate de Mode. À chaque pas, en fonction des valeurs de ses entrées et de son état courant, les équations associées à l'état courant produisent des sorties, et les conditions sur les transitions sont évaluées de façon à déterminer l'état pour le pas suivant (c'est-à-dire que les transitions sont considérées comme faibles). On peut noter que de tels structures de haut niveau peuvent être compilées vers un noyau minimal (Colaço *et al.*, 2005), donc elles n'ont pas besoin d'être représentées explicitement dans la sémantique formelle.

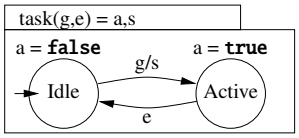
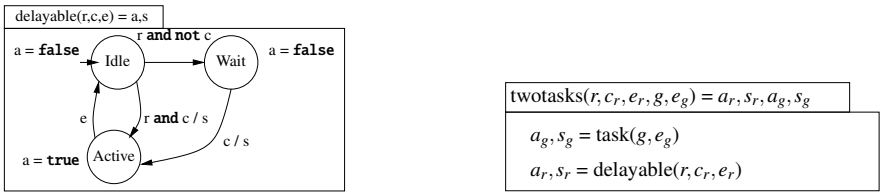


Figure 4. *Contrôle d'une tâche de base*

Un exemple d'Automate de Mode est un contrôleur de tâche basique, qui distingue entre ses états actif et inactif, illustré en Figure 4 dans une syntaxe graphique. Le nœud est nommé `task`. Une entrée de démarrage g provoque une transition de l'état initial inactif vers l'état actif, où le calcul a lieu, avec la consommation de ressources correspondante. Une sortie s est émise à **true** sur cette transition [2], qui sera transmise au système d'exploitation contrôlé pour invoquer l'action concrète de démarrage de tâche. Une autre entrée e signale la terminaison de la tâche, et fait retouner le contrôleur dans l'état inactif. Les équations associées aux états définissent les valeurs d'une sortie Booléenne a. Ce motif de base sera réutilisé de diverses manières.

2. De telles émissions sur des transitions, comme dans les automates de Mealy utilisés en syntaxe graphique par facilité, sont facilement encodées en équations associéees aux états.



(a) Tâche retardable. (b) Nœud composite avec tâche retardable.

Figure 5. *Exemple de programmation BZR*

Une variante en est la tâche retardable montrée en Figure 5(a). Un flot d'entrée additionnel c permet le contrôle de la requête r, soit en l'acceptant immédiatement et en allant dans l'état actif, soit en allant dans un état d'attente, d'où ce flot c peut ultérieurement déclencher le démarrage. Le flot de sortie a définit l'activité.

La composition des équations construit des systèmes d'équations avec la sémantique synchrone. Les nœuds peuvent être composés, par exemple des automates, se comportant comme un produit synchrone. La Figure 5(b) montre le nœud composite `twotasks`, construit par la composition d'instances des deux nœuds précédents.

2.2. *Systèmes de transition symboliques*

Nous représentons le comportement logique des programmes synchrones au moyen de systèmes de transition symboliques (STS). Le principe des STS est illustré en Figure 6. Typiquement, le rôle d'un compilateur de langage synchrone est de produire un tel système de transition, sous diverses formes (forme équationnelle, symbolique ou énumérée, ou sous forme de langage cible général tel que C ou Java). Pour un nœud particulier, une fonction de transition Trans prend, à chaque instant discret, un vecteur d'entrées X, et l'état courant State (vecteur de valeurs pour les STS), et produit la valeur de l'état suivant, mémorisée pour l'instant suivant. La fonction de sortie Out produit avec les mêmes entrées les valeurs de sorties Y.

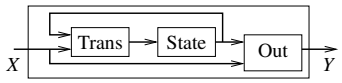


Figure 6. *Système de transition pour un programme*

Formellement, un STS $\mathscr{S}_f$ est un tuple (X,S,Y,T,O,Q,Q_0), qui décrit un programme synchrone dont les variables d'entrées sont $X \in \mathbb{B}^n$, les sorties $Y \in \mathbb{B}^p$, à l'aide de variables d'état $S = [S_1,\ldots,S_m] \in \mathbb{B}^m$. Ce tuple définit un comportement équationnel décrit en (1), où S et S' sont respectivement l'état courant et l'état suivant.

$$\begin{cases} S' = T(S,X) \\ Y = O(S,X) \\ Q(S,X) \\ Q_0(S) \end{cases} \quad [1]$$

– $T \in \mathbb{B}[S,X]$ est la fonction de transition. Il s'agit d'un vecteur de fonctions $[T_1,\ldots,T_n]$. Chaque fonction $T_i \in \mathbb{B}^{n+m} \to \mathbb{B}^m$ définit l'évolution de de la variable d'état S_i.

– $O \in \mathbb{B}[S,X]$ est la fonction de sortie.

– $Q_0 \in \mathbb{B}[S]$ est une contrainte définissant l'état initial du système.

– $Q \in \mathbb{B}[S,X]$ est une contrainte additionnelle sur l'état et les entrées admissibles du système, qui doit être vérifiée à chaque instant. Cette contrainte peut être utilisée pour définir des assertions sur les entrées, c'est-à-dire comme modèle d'environnement.

La sémantique d'un STS $\mathscr{S}_f = (S,Y,T,O,Q,Q_0)$ est définie par l'ensemble des séquences $(s_i,x_i,y_i)_{i\in\mathbb{N}}$ telles que :

$$Q_0(s_0) \wedge \forall i, Q(s_i,x_i) \wedge (s_{i+1} = T(s_i,x_i)) \wedge (y_i = O(s_i,x_i))$$

Cet ensemble de séquences est appelé Traces($\mathscr{S}_f$).

La composition parallèle synchrone de $\mathscr{S}_1 = (X_1,S_1,Y_1,T_1,O_1,Q_1,Q_{01})$ et $\mathscr{S}_2 = (X_2,S_2,Y_2,T_2,O_2,Q_2,Q_{02})$ est notée $\mathscr{S}_1 \| \mathscr{S}_2$, et consiste en la conjonction des prédicats composant les deux STSs. Ainsi, on a $\mathscr{S}_1 \| \mathscr{S}_2 = ((X_1 \cup X_2) \setminus (Y_1 \cup Y_2), S_1 \cup S_2, (Y_1 \cup Y_2), (T_1,T_2), (O_1,O_2), Q_1 \wedge Q_2, Q_{01} \wedge Q_{02})$ avec :

$$\begin{cases} S'_1, S'_2 = (T_1(S_1,X_1), T_2(S_2,X_2)) \\ Y_1, Y_2 = (O_1(S_1,X_1), O_2(S_2,X_2)) \\ Q_1(S_1,X_1) \wedge Q_2(S_2,X_2) \\ Q_{01}(S_1) \wedge Q_{02}(S_2) \end{cases}$$

Cette opération est définie seulement si les variables d'état et de sorties sont exclusives. Les variables de sorties d'un STS peuvent apparaître en entrée de l'autre STS : les communications entre deux systèmes sont ainsi exprimées à l'aide d'entrées/sorties communes, qui deviennent alors des sorties de la composition.

On note $\mathscr{S}_f \rhd A$ l'extension des contraintes de $\mathscr{S}_f$ avec le prédicat $A \in \mathbb{B}[S,X]$, soit $\mathscr{S}_f \rhd A = (X,S,Y,T,O,Q \wedge A, Q_0)$.

2.3. *Synthèse de contrôleurs discrets*

Les méthodes de SCD (Ramadge *et al.*, 1987) sont des méthodes constructives, qui permettent d'imposer des propriétés requises sur des systèmes ne les assurant pas

a priori. À partir d'une représentation de l'ensemble des comportements possibles du système (sous la forme d'automates finis, de réseaux de Petri, de STS, etc.) et des propriétés à satisfaire, le but de la SCD est de produire un système contraint ne présentant que les comportements satisfaisant les propriétés requises.

Dans notre contexte, la SCD s'applique sur un STS (initialement non contrôlé), pour lequel les entrées X ont été partitionnées en entrées incontrôlables (X^u) et contrôlables (X^c). Les entrées incontrôlables proviennent typiquement de l'environnement (capteurs, actions de l'utilisateur, évènements extérieurs au système), les entrées contrôlables seront contraintes par le contrôleur synthétisé. La SCD est appliquée avec un *objectif de synthèse* donné : une propriété logique temporelle G sur le comportement du système. Le but est d'obtenir un contrôleur $K \in \mathbb{B}[S, X^u, X^c]$, permettant de contraindre les valeurs des entrées contrôlables, telles que tous les comportements de $\mathscr{S}_f \rhd K$ satisfont la propriété donnée comme objectif de synthèse. Concernant les objectifs de synthèse, nous nous restreindrons ici aux propriétés d'invariance d'un sous-ensemble de l'espace d'état : $G \in \mathbb{B}[S]$. L'utilisation d'observateurs synchrones permet cependant l'expression de propriétés de sûreté plus générales.

Le fait que K soit une relation permet de conserver le caractère maximalement permissif de la synthèse : X^c reste en entrée du système contrôlé $\mathscr{S}_f \rhd K$, et tous les comportements du système initial respectant l'objectif de synthèse sont conservés. Cependant, dans une optique opérationnelle, il est possible à partir de K de dériver une fonction $C \in \mathbb{B}^{|S|+|X^u|} \to \mathbb{B}^{|X^c|}$ permettant de construire le système contrôlé $\mathscr{S}_f/C$ dans lequel X^c n'apparaît plus en entrée :

$$\mathscr{S}_f/C = \begin{cases} S' = T(S, C(S, X^u), X^u) \\ Y = O(S, C(S, X^u), X^u) \\ Q(S, C(S, X^u), X^u) \\ Q_0(S) \end{cases} \qquad [2]$$

Le résultat (2) est un STS contrôlé tel que $\forall (s, x^u, y) \in \text{Traces}(\hat{S}_f/C), G(s)$.

Du fait du caractère incontrôlable de certaines entrées, pris en compte lors de la synthèse, il peut arriver qu'il n'existe pas de contrôleur K admissible pour un STS et un objectif de contrôle donné. Cependant, une fois K obtenu, il existe toujours (au moins) une fonction C dérivée de K. Cette dérivation restreint le système contrôlé : $\mathscr{S}_f/C$ contient moins de comportements que $\mathscr{S} \rhd K$. Le calcul d'une fonction C à partir de K, et les problématiques de programmation afférentes (choix de valeurs « par défaut » pour les valeurs contrôlables non contraintes à un instant donné, définition de « préférences » de certains comportement par le programmeur, etc.), ne sont pas l'objet de cet article.

Nous utilisons ici l'outil SIGALI (Marchand *et al.*, 2000), qui implémente des méthodes de SCD, rendant la procédure de calcul du contrôleur automatique. Cet outil manipule des STSs encodés sous forme de BDD [3]. Le caractère symbolique évite

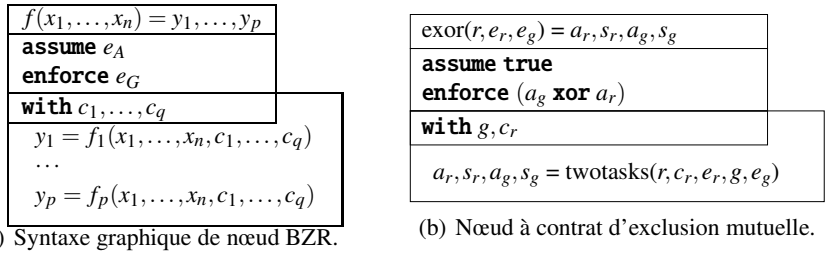
3. Binary Decision Diagram

l'énumération de l'espace d'état lors du calcul du contrôleur. Le processus complet de compilation du langage BZR inclut cet outil, rendant automatique la traduction des nœuds synchrones en STSs, les contrats sur ces nœuds en objectifs de synthèse, le calcul du contrôleur C et sa composition avec le programme initial compilé.

3. Le langage BZR : contrats comportementaux

Nous introduisons un nouveau constructeur de langage, permettant la séparation des problèmes entre description de composants à contrôler, et politique de contrôle à appliquer. L'avantage est que le programmeur n'écrit pas la solution, mais pose le problème de contrôle. Donc, quand la politique change pour le même système, ou quand des aspects du système changent mais sont gérés selon la même politique, les modifications sont limitées, la réutilisation facilitée, et la clarté favorisée. La sémantique de traces du langage est présentée en détail ailleurs (Delaval *et al.*, 2011), ainsi que sa compilation, et particulièrement ses aspects modulaires (Delaval *et al.*, 2010a).

3.1. *Constructeur de contrat*



(a) Syntaxe graphique de nœud BZR.

(b) Nœud à contrat d'exclusion mutuelle.

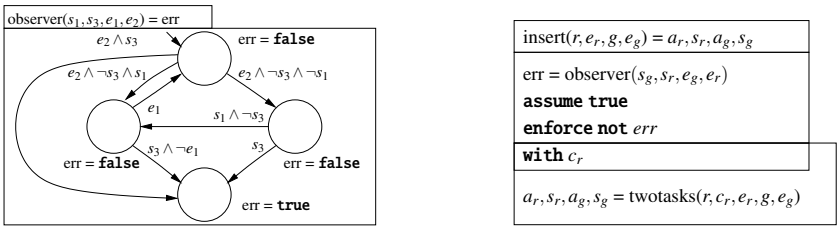
Figure 7. *Nœud BZR*

Nœud à contrat simple. Comme le montre la Figure 7(a), nous associons à un nœud un *contrat*, qui est un programme à deux sorties : une sortie e_A représente le modèle de l'environnement du nœud, et un prédicat e_G qui devra être satisfait par le nœud. Au niveau du nœud, le programmeur déclare des variables contrôlables $c_1,\ldots,c_q$, qui seront utilisées pour imposer cet objectif. Ce contrat signifie que le nœud sera contrôlé, c'est-à-dire que des valeurs seront données à $c_1,\ldots,c_q$ de telle sorte que, pour toute trace d'entrée respectant e_A, la trace de sortie respecte e_G. Ceci sera obtenu en calculant un contrôleur par SCD.

La Figure 7(b) montre un exemple simple de complémentarité entre les activités de deux tâches : une tâche de fond et une tâche retardable. Le nœud à contrat `exor` instancie le nœud `twotasks` de la Figure 5(b). Nous supposons pour cet exemple et le suivant que cette instanciation donne accès au corps du sous-nœud (cette option est disponible dans le compilateur). Un contrat est donné spécifiant que l'hypothèse sur

l'environnement est vide (ou **true**), et que la propriété à imposer est qu'une et une seule des deux tâches soit active à tout moment : $(a_g$ **xor** $a_r)$. De façon à imposer ce contrat, g et c_r sont définis localement au nœud à contrat, et sont concrètement utilisés pour retarder le démarrage de la tâche retardable quand la tâche de fond est déjà active, et réciproquement si la tâche retardable se termine, l'autre est démarrée.

Plusieurs nœuds peuvent avoir le même corps, et être spécialisés par différentes hypothèses et objectifs. Dans l'autre sens, il est possible d'appliquer le même contrat pour un corps différent, échangeant un sous-composant par sa description raffinée par exemple, et de ré-obtenir le contrôleur mis à jour simplement par compilation. Le contrat lui-même peut comporter un programme, typiquement un automate observant des traces et définissant des états, comme mentionné en Section 2.3, pour exprimer des propriétés de sûreté variées. Par exemple, un état d'erreur peut être défini où la propriété recherchée est fausse, avec l'intention de le garder en dehors du sous-espace d'états rendu invariant. Un tel observateur est illustré en Figure 8(a) : étant donné des flots d'entrée pour les événements de démarrage et d'arrêt des trois tâches, il a une valeur de sortie **true** sur le flot *err* quand une séquence est observée telle que la tâche 3 est démarrée après la tâche 2, sans qu'une exécution complète de la tâche 1 ait eu lieu entretemps, depuis s_1 jusqu'à e_1. Le contrat en Figure 8(b) utilise cet observateur pour avoir toujours une exécution de la tâche simple entre deux exécutions de la tâche retardable ; ceci revient à rendre invariant l'espace d'état où *err* est **false**. Pour imposer ceci, c_r est utilisé pour retarder le démarrage de la tâche retardable jusqu'à ce qu'une exécution complète de l'autre termine.



(a) Observateur : toujours t1 entre t2 et t3. (b) Nœud à contrat pour l'insertion de tâche.

Figure 8. *Nœud à contrat et observateur*

Nœud contrat composite. Un nœud BZR composite a un contrat par lui-même, et des sous-nœuds BZR avec leurs propres contrats, comme en Figure 9. Les sous-nœuds peuvent communiquer, par exemple, certains des x_{pi} être en y_{1i} et en x_i. C'est là que la modularité intervient, et l'information sur les contrats des sous-nœuds, qui est visible au niveau du composite, sera utilisée pour la compilation du composite. L'objectif est toujours de contrôler le corps du nœud, en utilisant les variables contrôlables $c_1, \ldots, c_q$, de façon à ce que e_G soit vrai, en supposant que e_A est vrai. Mais ici nous disposons d'information sur les sous-nœuds : nous ne gardons pas l'information sur leurs corps, mais ils sont abstraits à leurs contrats, qui peuvent alors être utilisés dans la SCD à ce niveau. Nous pouvons alors utiliser non seulement l'hypothèse e_A, mais

aussi, dans le cas de deus sous-nœuds, $(e_{A1} \Rightarrow e_{G1})$ et $(e_{A2} \Rightarrow e_{G2})$. Le problème de SCD devient alors : supposant e_A et $(e_{A1} \Rightarrow e_{G1})$ et $(e_{A2} \Rightarrow e_{G2})$, nous voulons imposer e_G, ainsi que e_{A1} et e_{A2}, de façon à ce que les contrats des sous-nœuds puissent être effectivement satisfaits. En particulier, une partie du contrôle au niveau du composite prend en charge de rendre vraies les hypothèses des sous-nœuds.

4. Exemple du contrôle d'un bras robotique

Notre langage induit une méthode de programmation différente et légèrement inhabituelle. Là où de façon classique la programmation consiste à écrire une solution, et ensuite à en vérifier éventuellement la correction, en BZR nous spécifions le problème, et la solution en est dérivée. Ceci est lié à l'approche de l'automatique :

– d'abord écrire des nœuds qui décrivent le procédé à contrôler, avec tous ses comportements possibles, en l'absence même de contrôle ; ce faisant, identifier les points de contrôle possibles, indépendemment de leur utilisation ;

– ensuite écrire sous forme de contrats les objectifs de contrôle ; on peut noter que différents objectifs peuvent avoir du sens pour un même procédé, et que la contrôlabilité du procédé pour un objectif donné n'est pas toujours acquise ;

– compiler ce programme pour dériver le contrôleur, en utilisant la SCD ; de même que l'inférence de type, la SCD peut être vue comme une technique de complétion de l'automate de contrôle, qui n'était que partiellement spécifié.

Cette section illustre ces points, dans le domaine particulier des systèmes embarqués, par l'exemple d'un bras robotique manipulateur.

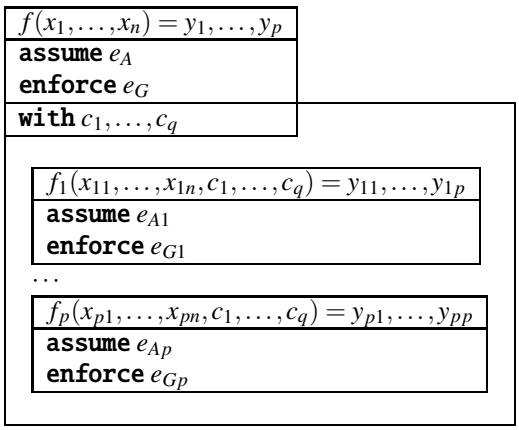


Figure 9. *Nœud BZR composite*

4.1. *L'étude de cas du bras robotique*

Notre exemple est une forme simplifiée d'une étude de cas (Aboubekr *et al.*, 2011) concernant un bras robotique, dont les articulations définissent un espace de travail mécaniquement accessible. Un tel robot doit toujours être sous le contrôle d'une boucle de commande (sinon ses mouvements deviendraient erratiques, dépendant de la gravité, du vent ou de toute force mécanique environnante) ; il y a aussi une exclusion entre ces lois de commande, l'actionneur étant une ressource exclusive entre elles. Elle sont mises en œuvre dans des tâches temps-réel, qui ont leurs contraintes d'échéance temps-réel, mais nous nous concentrons ici sur leurs enchaînements, et les caractérisons par le fait qu'elles peuvent être démarrées, et émettre un événement signalant leur fin quand leur objectif est atteint, ou des exception prédéfinies.

Nous considérons six tâches de cette sorte. Le bras robotique peut déplacer son extrémité, portant un outil, à l'intérieur de l'espace de travail, en utilisant une commande fondée sur ces coordonnées cartésiennes (tâche C). Toutefois, certains mouvements peuvent mener les articulations à leurs limites, en butée, appelées singularités. Ceci requiert de faire un mouvement intermédiaire pour contourner la singularité, en utilisant un autre commande, fondée sur des coordonnées articulaires comportant des valeurs d'angles des articulations (tâche J). Ces deux lois de commandes sont regroupées dans une même tâche CJ. Une autre commande possible consiste à procéder à un suivi de trajectoire, ce qui est utilisé typiquement pour pointer vers une cible à l'extérieur de l'espace de travail (tâche F). Une seconde tâche du même type est en charge des positions en bordure de l'espace de travail (tâche B). Une autre tâche est définie pour le changement d'outil (tâche CT) : elle inclut un mouvement vers le support à outils, et l'échange d'outils lui-même. Il y a deux outils disponibles : l'un est une pince, et peut être utilisé pour saisir la cible quand elle est à l'intérieur de l'espace de travail ; l'autre est une caméra, qui peut être pointée vers la cible quand elle est à l'extérieur. Enfin, une tâche de fond peut être activée en l'absence d'autres lois de commande, pour maintenir la position courante du bras robotique (tâche M) (un robot ne doit en effet jamais être hors de contrôle).

L'application consiste, pour ce système robotique, à, quand une cible est indiquée par un utilisateur via une interface graphique :

– si elle est à l'intérieur de l'espace de travail, aller la saisir avec la pince ;

– si elle est en bordure, aller en position centrale avec la caméra pointée vers elle ;

– sinon, si elle est à l'extérieur de l'espace de travail, aller jusqu'au bord de l'espace de travail et pointer la caméra vers elle.

4.2. *Modélisation des comportements*

La Figure 10 montre le nœud BZR pour cette étude de cas. Le corps du nœud décrit les comportements des différentes tâches temps-réel sous-jacentes, chacune au niveau d'abstraction de leur activation, ce qui est approprié pour la gestion de leurs

interactions. De gauche à droite, nous avons d'abord, pour la tâche F, une simple variation de la tâche retardable de la Figure 5(a) : depuis un état initial inactif, sur réception de l'entrée `outWork` signalant une cible en dehors de l'espace de travail, une transition est prise selon la variable de choix (à contrôler) nommée `cF` (pour : contrôle `c` de la tâche F) : si elle est à **`true`** alors la sortie `startF` est émise à **`true`** vers le gestionnaire de tâches temps-réel, la loi de commande de suivi de trajectoire est lancée, et l'état suivant est `Active` ; sinon si elle est à **`false`**, nous allons à l'état `Wait`, d'où, quand `cF` est à **`true`**, la tâche peut être démarrée. Depuis `Active` et `Wait`, la réception de `stopF` enclenche une transition vers `Idle`. Pour B, quand la cible est en bordure, nous avons une deuxième instance de ce comportement. L'automate suivant décrit CJ, les mouvements à l'intérieur de l'espace de travail : il suit lui aussi le schéma de la tâche retardable, avec la variable de choix `cCJ`, et un état actif plus élaboré : il est raffiné hiérarchiquement en un sous-automate où initialement la loi de commande en corrdonnées cartésiennes est active, et sur réception de l'exception `singularity`, une commutation est faite vers la tâche de commande articulaire ; quand elle termine, le contrôle retourne vers la commande cartésienne. En bas de la figure, dans le corps du nœud, nous avons l'automate de la tâche M, où le démarrage et la terminaison sont contrôlables au travers de `cM`. En-desssous, nous avons des équations définissant l'arrêt des tâches. A côté, on voit l'automate contrôlant la tâche de changement d'outil CT : elle peut être démarrée par le contrôleur en utilisant la variable `cCT`. Une fois active, quand on arrive au support à outils sur l'événement `endCT`, soit le bras prend l'outil s'il est disponible (quand `take` est à **`true`**), soit il attend jusqu'à ce qu'il le soit. Nous avons aussi un observateur *Obs* pour l'outil courant, qui change d'état à chaque fois qu'un nouvel outil est pris.

Cet automate parallèle décrit toutes les séquences possibles pour les tâches : il ne prend pas en charge explicitement leurs exclusions, ou la gestion de l'outil courant. Ceci sera montré dans la suite dans le contrat déclaratif, et compilé en utilisant la SCD.

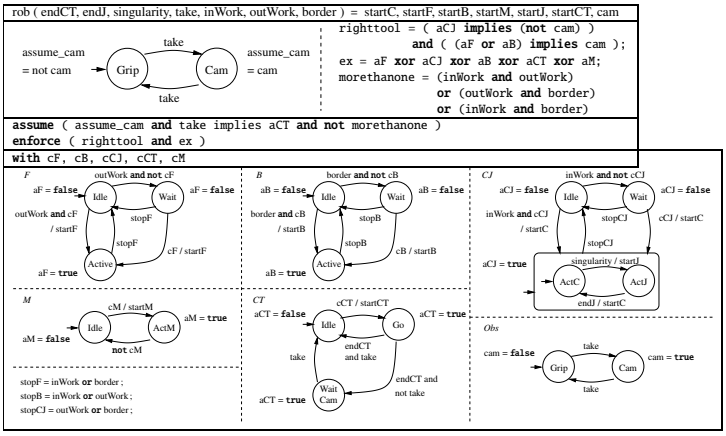


Figure 10. *Exemple du contrôleur de bras robotique : nœud BZR, avec son contrat*

4.3. *Contrat*

L'application doit lancer la tâche du robot correspondant à l'état courant de la cible (à l'intérieur, à l'extérieur ou en bordure de l'espace de travail) et changer l'outil pour avoir celui adéquat pour chaque tâche. Donc l'objectif de contrôle est d'abord d'assurer d'avoir le bon outil, et ensuite de ne permettre qu'une seule tâche active à la fois, et au moins une comme mentionné en Section 4.1. L'ensemble de variables contrôlables $\{cF, cB, cCJ, cCT, cM\}$, défini dans la partie `with` du contrat, est utilisé pour réaliser cet objectif.

Le contrat spécifie que le nœud sera contrôlé de telle sorte que, pour toute trace d'entrées incontrôlables, la trace de sortie satisfera les deux objectifs. On peut voir le contrat dans la partie supérieure de la Figure 10 : c'est lui-même un programme, définissant des variables locales par des équations.

– pour avoir le bon outil pour la bonne tâche, une variable Booléenne `goodtool` est définie comme la conjonction de deux implications : elles disent que quand une tâche est active (`aCJ`, respectivement `aF or aB`), ça implique que le bras porte le bon outil (`not cam`, respectivement `cam`).

– pour l'exclusion et la commande par défaut, une équation définit `ex`, qui est la disjonction exclusive des états actifs entre les tâches : il y a une seule tâche qui commande l'actionneur, et au moins une.

Ce contrat est aussi doté d'un automate, qui sera visible quand le nœud sera réutilisé, et fait la relation entre `cam` et `take`. Étant donné que seul le corps du nœud peut produire des sorties, l'observateur est aussi là, mais ces deux automates jouent des rôles différents. L'hypothèse sur l'environnement est que `assume_cam` est à **`true`**, ce qui fait la relation entre les deux automates mentionnés ci-dessus ; l'entrée `take` est seulement présente quend *CT* a été activée (c'est-à-dire, correspond à un changement d'outil effectif) ; une seul des entrées à l'intérieur, à l'extérieur et en bordure est à **`true`** à la fois. Le contrat doit imposer que les deux Booléens soient vrais.

4.4. *Simulation et scénario typique*

Ce scénario montre l'intervention du contrôleur sur le système, pour assurer la satisfaction des objectifs. Nous partons d'une situation où la tâche *CJ* est active, la cible à l'intérieur de l'espace de travail, et l'outil porté par le bras robotique correspond à `not cam`. L'utilisateur désigne une cible hors de l'espace de travail, l'application reçoit l'entrée `outWork` à **`true`**. Ceci cause une transition de l'automate de contrôle de *CJ* vers son état initial. Ceci force aussi l'automate de *F* à quitter son état initial ; ici, nous avons un point de choix conditionné par `cF`. Du fait de la première propriété du contrat, `righttool` doit être maintenu à **`true`**, de sorte que étant donné que l'outil courant correspond à `not cam`, le contrôleur ne peut pas permettre la transition de *F* vers `Active`, et doit conserver la valeur **`false`** à `cF`. Donc la tâche *F* va dans son état `Wait`. Du fait de l'autre propriété du contrat, `ex` doit être maintenue à **`true`**, ce qui

force le contrôleur à maintenir au moins une tâche dans l'état actif. Donc il lance la tâche CT, en utilisant la variable contrôlable `cCT`, et elle changera l'outil.

Dans une réaction ultérieure, à la fin de la tâche CT, sur l'événement `endCT`, si `take` est à **`true`**, l'automate observant l'outil courant va dans l'état où `cam` est à **`true`**. Donc, nous avons le bon outil pour la tâche F, et le contrôleur peut libérer F de son état `Wait` vers `Active`, en donnant la valeur **`true`** à la variable contrôlable `cF`.

Ces simulations valident la logique d'enchaînement des tâches ; elles doivent être complétées par une validation des aspects temps-réel quantitatif.

4.5. *Contrats modulaires*

Nous illustrons les contrats modulaires en considérant deux systèmes robotiques, partageant le même outil caméra, alors qu'ils ont chacun sa pince. Le modèle du contrôle pour un tel atelier robotique est illustré en Figure 11, où deux instances du nœud `rob` sont en parallèle. Le contrat dit simplement que l'exclusivité doit être imposée entre `cam1` et `cam2`, sans plus d'hypothèse, avec les contrôlables `take1` et `take2`.

```
tworobs ( endCT1, ... border1, endCT2, ... border2 )
=  startC1, ... startCT1, startC2, ... startCT2
enforce ( not ( cam1 and cam2 ))
with take1, take2
  startC1, ... startCT1, cam1 = rob (endCT1, ... border1, take1) ;
  startC2, ... startCT2, cam2 = rob (endCT2, ... border2, take2) ;
```

Figure 11. *Deux robots partageant une caméra exclusive*

Un autre exemple de contrat modulaire a été développé, avec des comportements simplifiés ne comportant que des tâches retardables, mais illustrant l'utilisation de la modularité : nous avons d'abord construit un nœud à contrat avec n instances de cette tâche, puis construit un nœud pour $2n$ tâches. Sur cet exemple, des évaluations de performance du calcul du contrôleur ont montré une forte amélioration du passage à l'échelle de notre approche (Delaval *et al.*, 2010a).

5. Conclusion

Nous avons proposé une intégration de la Synthèse de Contrôleurs Discrets (SCD) dans le processus de compilation d'un langage réactif synchrone : BZR, et avons traité une étude de cas. Notre contribution concerne donc la conception de langages de programmation, l'application de la SCD, et le contrôle de systèmes de calcul adaptatifs et reconfigurables. Les perspectives comportent des enrichissements du langage et de son utilisation de la SCD. Nous envisageons de considérer des contrats définis par des objectifs de synthèse autres que d'invariance, par exemple d'accessibilité, ou de contrôle optimal. Une autre point intéressant serait d'en augmenter en direction

des domaines de valeurs, l'expressivité en y associant des techniques d'interprétation abstraite. Nous continuons d'explorer la problématique de contrôle de reconfiguration dans les systèmes adaptatifs et autonomiques, dans différents contextes, et à différents niveaux, depuis les architectures dynamiquement partiellement reconfigurables à base de FPGA (Guillet *et al.*, 2011), en passant par les boucles d'administration autonomique dans les systèmes d'exploitation (de Palma *et al.*, 2010), et les intergiciels à base de composants (Delaval *et al.*, 2010b).

6. Bibliographie

Aboubekr S., Delaval G., Pissard-Gibollet R., Rutten E., Simon D., « Automatic generation of discrete handlers of real-time continuous control tasks », *Proc. 18th World Congress of the International Federation of Automatic Control (IFAC),* Milano, Italy, aug. 28 – sep. 2, 2011.

Altisen K., Clodic A., Maraninchi F., Rutten E., « Using Controller Synthesis to Build Property-Enforcing Layers », *European Symposium on Programming*, Warsaw, April, 2003.

Colaço J.-L., Pagano B., Pouzet M., « A Conservative Extension of Synchronous Data-flow with State Machines », *Embedded Software (EMSOFT)*, New Jersey, USA, Sept., 2005.

de Palma N., Delaval G., Rutten E., « QoS and Energy Management Coordination using Discrete Controller Synthesis », *Proc. 1st International Workshop on Green Computing Middleware (GCM'2010),* Bangalore, India, November 29, 2010, 2010.

Delaval G., Marchand H., Rutten E., « Contracts for Modular Discrete Controller Synthesis », *Languages, Compilers and Tools for Embedded Systems, Stockholm, Apr.*, 2010a.

Delaval G., Rutten E., « Reactive model-based control of reconfiguration in the Fractal component-based model », *Component Based Software Engineering,* Prague, June, 2010b.

Delaval G., Rutten E., Marchand H., Integrating Discrete Controller Synthesis in a Reactive Programming Language Compiler (full version), Technical report, INRIA, available from `http://proton.inrialpes.fr/~rutten/docs/bzr-lang-long.pdf`, 2011.

Guillet S., de Lamotte F., Rutten E., Gogniat G., Diguet J.-P., « Modélisation et contrôle de reconfiguration partielle dynamique », *Actes du Symposium en Architecture de Machines, SympA'14,* 10-13 mai 2011, Saint-Malo, France, 2011.

Marchand H., Bournai P., Le Borgne M., Le Guernic P., « Synthesis of Discrete-Event Controllers based on the Signal Environment », *j. Discrete Event Dynamic System*, 2000.

Ramadge P. J., Wonham W. M., « Supervisory control of a class of discrete event processes », *SIAM J. Control Optim.*, 1987.

Wang Y., Lafortune S., Kelly T., Kudlur M., Mahlke S., « The theory of deadlock avoidance via discrete control », *Principles of programming languages, POPL, Savannah, USA*, 2009.

Yu H., Gamatié A., Rutten E., Dekeyser J., « Adaptivity in High-Performance Embedded Systems : a Reactive Control Model for Reliable and Flexible Design », *The Knowledge Engineering Review (jKER), Special Issue on Trends in High-Performance Computing and Communications for Ubiquitous Computing*, 2010. *(to appear).*