Validation temporelle et déploiement d'une application de contrôle industrielle à base de composants

Mohamed Khalgui

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Validation temporelle et déploiement d'une application de contrôle industriel à base de composants

THÈSE

présentée et soutenue publiquement le Vendredi 2 février 2007

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine
(spécialité informatique)

par

Mohamed KHALGUI

Composition du jury

Président :	Professeur Jean-Marc Jézéquel. Université de Renne1. France.
Rapporteurs :	Professeur Jean-Marc Faure. École Normale Supérieur de Cachan. France. Professeur Charles André. Université de Nice - Sophia Antipolis. France.
Examinateurs :	Professeur Samir Ben Ahmed. Université Tunis El Manar. Tunisie. Professeure Françoise Simonot-Lion. Institut Nationale Polytechnique de Lorraine. France. MdC Anne Boyer. Université Nancy 2. France. MdC Xavier Rebeuf. Institut Nationale Polytechnique de Lorraine. France.

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Je tiens à remercier ma directrice de thèse ***Madame Françoise Simonot-Lion***, pour l'acceuil dans l'équipe TRIO et pour ses remarques intéressantes durant mes années de thèse.

Je tiens à remercier particulièrement mon encadrant de thèse ***Monsieur Xavier Rebeuf*** d'avoir dirigé tous mes travaux dans un cadre de travail communicatif et amical. Je suis reconnaîssant pour son effort et son suivi durant toutes mes années de thèse. Je le remercie également pour ses conseils, ses encouragements, son support, son soutien et surtout sa gentillesse... Bravo Xavier pour ton talent d'homme de terrain *diplomate* et correct....

Je tiens à remercier également,

- ***Monsieur Jean-Marc Jézéquel*** pour sa présidence de mon jury de thèse.
- ***Mon Professeur*** *à l'université de Tunis* ***Monsieur Samir Ben Ahmed*** pour ses conseils, son soutien et ses encouragements depuis les années de la "fac" ainsi que sa participation au jury de ma thèse.
- mes rapporteurs ***Monsieur Charles André*** et ***Monsieur Jean-Marc Faure*** pour les remarques intéressantes proposées pour l'amélioration du manuscrit.
- **Madame Anne Boyer** pour sa participation au jury de ma thèse.

Aujourd'hui, je rends un hommage particulier à tous mes enseignants des études primaires (le feu Khabthani en particulier), secondaires et aussi supérieurs. je n'oublis pas à remercier ***Monsieur Adel Khalfallah*** de m'avoir aidé à venir ici en France pour faire un DEA et des travaux de thèse....

Je tiens aussi à exprimer mes sincères remerciements à,

- ***Madame Laurence Benini*** de m'avoir facilité les tâches administratives pour aller à des conférences et des séminaires.
- tous les membres de l'équipe TRIO et à tous les amis au Loria, à l'EEIGM et à l'ESIAL.

Enfin, ce document n'est pas un idéal scientifique, je remercie le lecteur pour ses analyses et surtout ses critiques pour le progrès et la gloire scientifique....

Je dédie ce travail,
à mes parents...
à ma femme Olfa...
à toute ma grande famille...

Résumé

Dans cette thèse, nous nous intéressons à la validation temporelle ainsi qu'au déploiement d'applications de contrôle industriel à base de composants. La technologie des composants retenue est celle des Blocs Fonctionnels définie dans la norme industrielle IEC 61499. Un Bloc Fonctionnel est défini comme un composant réactif supportant des fonctionnalités d'une application. L'avantage de cette norme, connue dans l'industrie, est la description statique de l'application ainsi que de son support d'exécution.

Une première contribution de la thèse est l'interprétation des différents concepts définis dans la norme. Nous précisons, en particulier, la dynamique du composant en vue de décrire un comportement déterministe de l'application. Pour appliquer une validation temporelle exhaustive, nous proposons un modèle de comportement d'un Bloc Fonctionnel à l'aide du formalisme des automates temporisés. D'autre part, nous fournissons une sémantique au concept de réseau de Blocs Fonctionnels pour décrire une application comme une composition de Blocs.

Une deuxième contribution de la thèse est le déploiement de tels réseaux sur une architecture distribuée multi-tâches tout en respectant des propriétés sur les temps de réponse de bout en bout. Nous transformons un réseau de Blocs Fonctionnels vers un ensemble de tâches élémentaires dépendantes, appelées actions. Cette transformation permet l'exploitation de résultats d'ordonnancement pour valider la correction temporelle de l'application. Pour déployer les blocs d'une application, nous proposons une approche hybride alliant un ordonnancement statique non-préemptif et un autre ordonnancement en ligne préemptif. L'ordonnancement statique permet la construction des tâches s'exécutant sur chaque calculateur. Ces tâches sont vues comme des séquencements statiques d'actions. Elles sont alors à ordonnancer dynamiquement selon une politique préemptive reposant sur EDF (Earliest Deadline First). Grâce à cette approche, nous réduisons le nombre de commutation de contexte en regroupant les actions au sein des tâches. De plus l'ordonnancement dynamique préemptif augmente la faisabilité du système.

Enfin, une dernière contribution est une extension de la deuxième. Nous proposons une approche d'allocation de réseaux de blocs fonctionnels sur un support d'exécution distribué. Cette allocation, basée sur une heuristique de Liste, se repose sur la méthode hybride pour assurer un déploiement faisable de l'application. Le problème d'allocation est de trouver pour chaque bloc fonctionnel le calculateur capable de l'exécuter tout en respectant des contraintes fonctionnelles, temporelles et de support d'exécution. Notons enfin que l'heuristique proposée se base sur une technique de retour-arrière pour augmenter l'espace de solutions.

Mots-clés: Approche par composants, Systèmes de Contrôle Industriel, Systèmes Temps-Réel Embarqués, Blocs Fonctionnels, IEC 61499, Model-checking, Ordonnancement temps-réel, Déploiement et Allocation.

Abstract

This thesis deals with the temporal validation and the deployment of component-based industrial control applications. We are interested in the Function Blocks approach, defined in the IEC 61499 standard, as a well known component based technology in the industry. A Function Block is an event triggered component owning data to support the application functionalities. The advantage of this technology is the taking into account of the application and also its execution support.

The first thesis contribution deals with the interpretation of the different concepts defined in the standard. In particular, we propose a policy defining a deterministic behavior of a FB. To apply an exhaustive temporal validation of the application, we propose a behavioral model of a Block as Timed Automata. On the other hand, we propose a semantic for the concept of FBs networks to develop industrial control applications.

The second thesis contribution deals with the deployment of FBs networks in a distributed multi-tasking architecture. Such deployment has to respect classical End to End Response Time Bounds as temporal constraints. To validate the temporal behavior of an application, we propose an approach transforming its blocks into an actions system with precedence constraints. The purpose is to exploit previous theories on the scheduling of real-time systems. To deploy FBs networks in feasible OS tasks, we propose a Hybrid scheduling approach combining an off-line non-preemptive scheduling and an on-line preemptive one. The off-line scheduling allows to construct OS tasks from FBs, whereas the on-line one allows to schedule these tasks according to the classical EDF policy. A constructed OS task is an actions sequence defining an execution scenario of the application. Thanks to this approach, we reduce the context switching at run-time by merging application actions in OS tasks. In addition, the system feasibility is increased by applying an on-line preemptive policy.

Finally, the last thesis contribution is an extension of the previous one. We propose an approach allocating FBs networks in a distributed architecture. Based on a heuristic, such approach uses the hybrid method to construct feasible OS tasks in calculators. The allocation problem of a particular application FB is to look for a corresponding calculator while respecting functional, temporal and execution support constraints. We note that the proposed heuristic is based on a back-tracking technic to increase the solutions space.

Keywords: Component approach, Industrial Control Systems, Embedded Real-Time Systems, Function Blocks, IEC 61499, Model-checking, Real-Time scheduling Deployment and Allocation.

Table des matières

Chapitre 8
Conclusion Générale et Perspectives

Chapitre 1

Introduction

De nos jours, le développement d'applications de contrôle industriel sûres de fonctionnement est devenu de plus en plus complexe [Crnkovic and Larsson, 2002]. En effet, ces applications assurent de plus en plus de fonctions et doivent vérifier des propriétés fonctionnelles mais également extra-fonctionnelles [Projet-Artist, 2003] ; parmi ce dernier type de propriétés, nous nous intéressons aux propriétés temporelles dont la vérification est requise avant toute mise en service de systèmes dont la mission est le contrôle de procédés physiques critiques. De plus, la durée de développement de telles applications se raccourcit en raison de la pression économique et de la mise sur le marché de plus en plus rapide des produits.

Pour répondre à ces différents besoins, le concept de "composant" [Heinman and Council, 2001] commence à se diffuser dans les milieux industriels [Projet-Artist, 2003] et plusieurs définitions, technologies et approches ont été proposées, dans ce contexte, pour le développement d'applications par intégration de composants.

Deux concepts fondamentaux sont indispensables dans toute approche de composant :

- **Le modèle de composant :** définit les caractéristiques d'un composant selon une technologie donnée [Projet-Artist, 2003].
- **Le framework :** représente une couche logicielle entre le système d'exploitation et l'application à base de composants [Crnkovic and Larsson, 2002]. Cette couche définit toute l'infrastructure gérant les interactions entre les composants.
  Les systèmes de contrôle industriel, sont en fait implémentés sur des équipements ou dispositifs de calcul ("device") qui peuvent être des automates programmables, mono ou multi-tâches ou des automates de régulation, généralement multi-tâches. Ces équipements sont de plus en plus souvent connectés via une architecture de réseaux de communication.
  Dans le cadre de ces dispositifs, le framework assure l'interaction avec :
  ** Les procédés physiques (grâce à des capteurs/actionneurs).
  ** D'autres composants placés dans d'autres dispositifs.

Dans la figure 1.1, nous présentons le modèle d'un tel dispositif de contrôle industriel. Le framework gère tous les composants de l'application en assurant leurs interactions avec les procédés physiques ("interface du procédé") et avec les autres dispositifs de contrôle ("interface de communication").

De nos jours, un des problèmes majeurs pour le développement de systèmes de contrôle industriel est le déploiement d'une application logicielle construite sous forme d'un ensemble de composants interagissant (architecture fonctionnelle) dans les dispositifs d'une architecture matérielle distribuée (figure 1.2). Ce déploiement relève de deux activités qui peuvent être coordonnées :

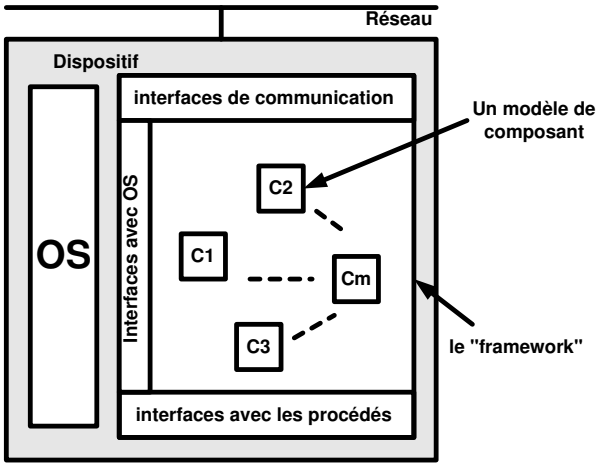

FIG. 1.1 – le modèle d'un dispositif de contrôle

– la distribution ou allocation de composants sur un dispositif (un calculateur) ; cette allocation peut être contrainte par des propriétés de résidence, de co-résidence, d'exclusion de résidence, de ressources mémoire, etc. ; elle est également contrainte par les propriétés de performances requises par l'application ;
– le déploiement des composants logiciels eux-mêmes sur une ou des tâches, selon les possibilités fournies par le système d'exploitation.

Ce déploiement doit, en particulier, garantir le respect des contraintes fonctionnelles mais aussi temporelles. De plus, il doit prendre en compte certaines contraintes opérationnelles comme, par exemple dans le cas d'un système sous-jacent multi-tâches, le nombre de tâches maximal supporté par les systèmes d'exploitation des dispositifs. La question qui se pose est alors : comment peut on construire un tel déploiement et, plus spécifiquement, comment construire, dans le cas multi-tâches, les tâches qui supportent l'exécution des composants fonctionnels tout en respectant toutes les contraintes décrites dans le cahier des charges.

Deux solutions existent pour résoudre le problème.

* On construit explicitement (hors-ligne) l'allocation des composants sur les calculateurs ainsi que, sur chaque calculateur, le séquencement de l'exécution des composants logiciels ; c'est ce qui est fait pour un déploiement sur des automates programmables mono-tâches ; ceci peut être étendu dans le cas des dispositifs multi-tâches ; les avantages et inconvénients de cette approche sont les suivants.

- Dans le cas d'applications soumises à des contraintes de temps et à des contraintes imposées par le systèmes d'exploitation (un nombre de tâches borné, par exemple), les techniques utilisées pour ce faire reposent souvent sur une exploration des solutions possibles jusqu'à trouver une solution correcte ou sur une exploration exhaustive pour trouver la meilleure solution correcte au sens d'un critère donné ; ce problème est, dans le cas général, NP-complet,

\+ Par contre, un avantage de cette approche est qu'elle ne demande pas la disponibilité de services coûteux supportant l'exécution des composants (par exemple, une politique de choix dynamique pour l'ordonnanceur), chacune des tâches ayant un comportement entièrement prédéterminé *a priori* : séquencement des algorithmes à l'intérieur des

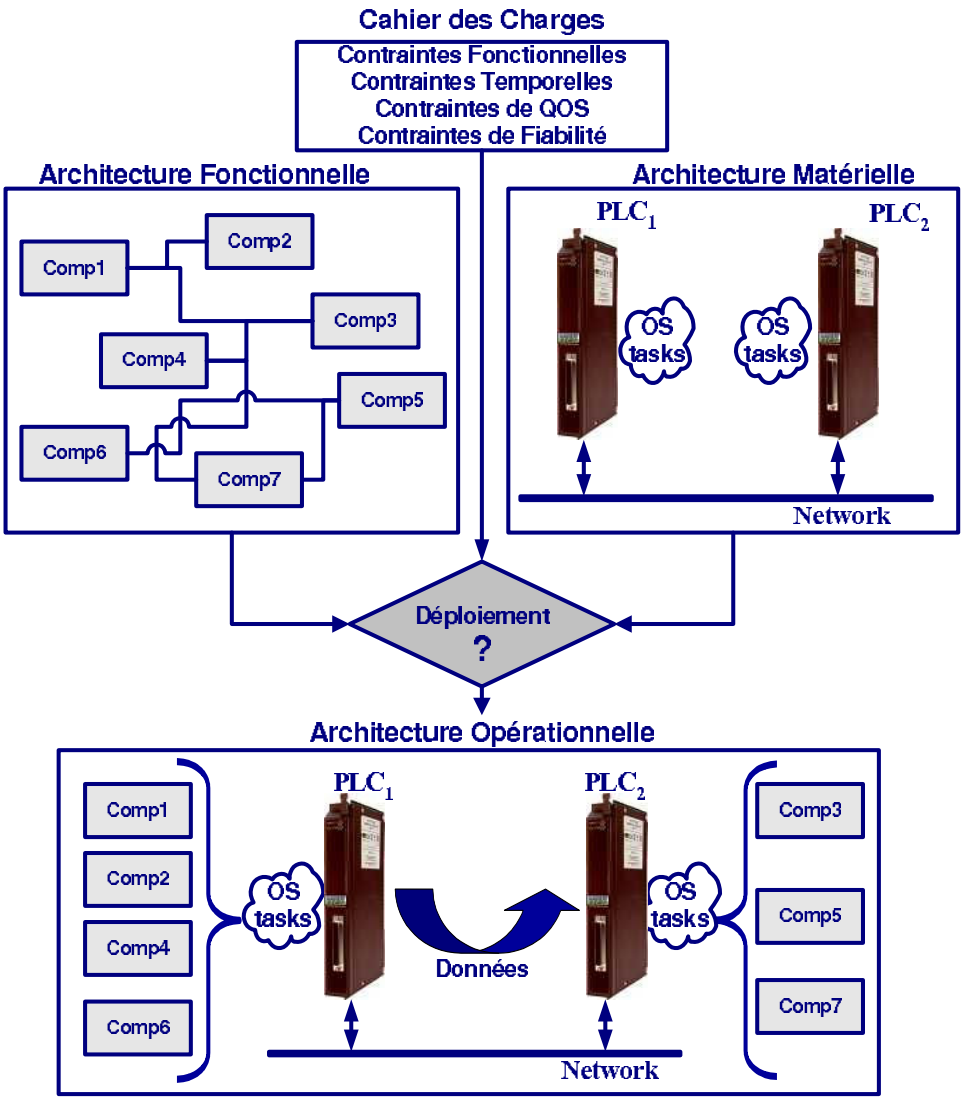

FIG. 1.2 – Approche de déploiement

tâches et priorité fixe statique pour chaque tâche.

* Le déploiement des composants est dynamique et se fait en-ligne pendant l'exécution de l'application. Ce déploiement peut être total : allocation des composants sur les calculateurs et déploiement des composants sur des tâches à l'intérieur de chaque calculateur ; il peut être, au contraire, limité à la deuxième activité, l'allocation elle-même étant statiquement définie. Dans ce cas, les tâches sont construites implicitement (en ligne). Une brêve synthèse fait apparaitre les points suivants :

  - Il est nécessaire d'ajouter à la plate-forme d'exécution des services complexes de migration de composants, de politiques de calcul des caractéristiques des tâches en ligne (évaluation de son WCET selon la configuration courante, priorité, règles d'émission, etc.) assorties de test de faisabilité pour garantir les contraintes de temps. Ces services peuvent s'avérer trés coûteux en temps d'exécution et en ressource mémoire.

  + Il est évident que pour la partie "ordonnancement" proprement dite, sous certaines conditions, les politiques dynamiques ordonnancent plus de configurations que les politiques statiques [Cottet *et al.*, 2000]. De plus, si on intégre une allocation dynamique des composants aux calculateurs disponibles, cette solution permet d'implanter plus facilement des mécanismes de tolérance aux fautes par recouvrement (migration de composants).

Le challenge est donc de trouver un bon compromis entre déploiement entièrement statique hors ligne et déploiement entièrement dynamique en ligne.

Cette thèse apporte une solution au problème de déploiement de composants dans le cas d'applications construites à partir de la norme industrielle IEC 61499 [IEC61499-2, 2004] (dans cette norme, un composant est appelé un Bloc Fonctionnel) et pour lesquelles les stimuli venant de l'environnement (capteurs, opérateurs, autres applications) arrivent périodiquement. Notons que cette hypothèse peut conduire à un surdimensionnement dans le cas où le système doit être réactif à des alarmes de nature aléatoire car il doit alors consulter périodiquement son environnement pour savoir si l'alarme est ou non présente. Néanmoins, l'hypothèse de périodicité présente des intérêts bien connus en termes de prédictabilité et de déterminisme ; de plus, elle s'applique bien à des systèmes contrôlés (au sens de l'automatique).

Un Bloc Fonctionnel est un composant réactif [André, 2003] proposé dans le standard industriel IEC 61499 [IEC61499-2, 2004]. Ce standard industriel définit précisément le modèle du composant et aussi son contrôle (à l'aide d'un module appelé "Execution Control Chart") ce qui distingue ce modèle des autres modèles académiques qui eux décrivent explicitement le comportement du composant. Notons de plus, qu'une application selon ce standard est considérée comme un réseau de Blocs Fonctionnels distribués sur plusieurs "ressources" de calculateurs [Crnkovic and Larsson, 2002]. Une ressource est définie dans le standard comme une unité logique, d'un calculateur, contrôlant l'exécution d'un ensemble de Bloc Fonctionnels. Basée sur une heuristique, nous proposons une méthode allouant les différents blocs de l'application dans une architecture hétérogène. Cette allocation prend en compte certaines contraintes fonctionnelles et extra fonctionnelles décrites dans le cahier des charges [Khalgui *et al.*, 2004b]. La méthode déploie statiquement les Bloc Fonctionnels sur des calculateurs ainsi que la construction des tâches OS sur chacun. Cette construction se base sur un séquencement hors-ligne de l'exécution des blocs.

En se basant une politique à priorité dynamique, la méthode vérifie en plus la faisabilité de ces tâches pour valider l'ordonnançabilité de l'application. Grâce à cette méthode, nous montrons dans une étude de cas l'optimisation de la distribution de l'application en réduisant le nombre de tâches construites ainsi que de calculateurs utilisés.

Le plan de ce document est le suivant :

Le chapitre 2 dresse un rapide état de l'art sur les modèles et technologies de conception de composants et d'architectures de composants.

Dans le chapitre suivant (chapitre 3), nous présentons la technologie des Bloc Fonctionnels selon le standard IEC 61499. Nous présentons en particulier les différents concepts définis pour le développement d'applications de contrôle industriel. Néanmoins, certains de ces concepts ne sont pas formellement étudiés dans les travaux proposés par la communauté, à cause du manque de précisions dans la norme. Dans ce manuscrit, nous proposons des interprétations plus restreintes de ces concepts permettant leur exploitation sûre [Khalgui *et al.*, 2004a].

Dans le chapitre 4, nous proposons une approche analysant l'ordonnançabilité d'un sous ensemble de Bloc Fonctionnels pour la construction d'un séquencement hors-ligne à l'intérieur d'une ressource [Khalgui *et al.*, 2005a]. Cette approche est possible à appliquer en mode dégradé si la spécification de l'application peut exhiber formellement des règles sur le non-respect acceptable de contraintes temporelles [Khalgui *et al.*, 2006d; 2006c].

Dans le chapitre 5, nous proposons une approche configurant, dans un calculateur, les tâches qui seront gérées par un système d'exploitation (tâches OS) et dont la structure est obtenue à partir des résultats du chapitre précédent. Ces tâches OS sont à structure conditionnelle ; elles sont modélisées à l'aide d'un modèle de tâches, développé par Sanjoy Baruah et utilisé pour l'implémentation de codes conditionnels temps-réel [Khalgui *et al.*, 2006b; 2006a].

Dans le chapitre 6, nous proposons l'approche d'allocation/distribution d'une application à base de Bloc Fonctionnels sur une architecture hétérogène. Cette approche, tenant compte des contributions proposées dans les chapitres précédents, est basée sur une heuristique [Khalgui and Rebeuf, 2006].

Dans le chapitre 7, nous présentons une étude de cas industrielle (exemple de chaîne de production) sur laquelle nous appliquons les contributions de cette thèse. Nous montrons en particulier l'apport de l'approche de déploiement/allocation proposée.

Enfin, nous finissons le manuscrit avec des perspectives ouvertes pour le développement d'applications de contrôle et nous montrons en particulier l'utilité des apports de la thèse dans nos prochains travaux de recherches.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre, nous présentons un état de l'art sur les approches de conception d'applications à partir de composants [Defour *et al.*, 2004; Jézéquel, 2006]. Dans un premier temps, nous citons des travaux d'ordre général sur les composants et nous abordons les formalisations qui ont été proposées pour modéliser des composants et des applications conçues à l'aide de composants. Ensuite, la conception d'applications amène à parler de "composition" de composants ; dans ce contexte, nous présentons un rapide état de l'art sur les langages de description d'architectures les plus pertinents dans le contexte de cette thèse. Ensuite, avant de conclure, nous donnons un panorama des technologies de composants les plus connues parmi lesquelles on peut trouver la technologie des Blocs Fonctionnels choisie comme cadre d'étude de cette thèse.

Les principales contributions de cette thèse portent sur la spécification, pour certains types d'applications, d'approches de déploiement. Ces approches reposent sur le concept de tâche ainsi que sur des techniques d'ordonnancement et d'ordonnançabilité connues. Néanmoins, pour une lecture plus facile de ce document, nous avons choisi de ne pas introduire l'état de l'art sur ces concepts dans ce chapitre mais, au contraire, de les présenter lorsque le besoin s'en fait sentir dans les chapitres concernés.

De la même manière, de nombreux travaux ont été réalisés sur les Blocs Fonctionnels. Il nous a semblé plus naturel et lisible de les présenter dans le chapitre 3 aprés que la description résumée du standard les définissant ait été fournie.

Ainsi, ce chapitre ne concerne que les approches composants au sens général du terme, les tentatives de formalisation et d'utilisation qui existent.

2.2 Définitions du concept de composant

Dans [Projet-Artist, 2003], un composant est défini comme "une unité logicielle ré-utilisable dans différentes applications". Il est généralement caractérisé par une implémentation et une interface. Certains experts associent souvent le concept de composant au concept d'objet dans le développement orienté objet [Crnkovic and Larsson, 2002]. Pourtant, plusieurs différences existent entre les deux concepts comme la granularité, la notion de composition et le processus de développement [Crnkovic and Larsson, 2002]. L'application est considérée alors comme une composition de composants [Szyperski, 1998; Crnkovic and Larsson, 2002].

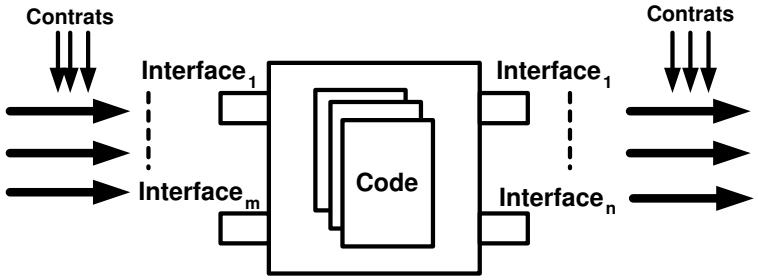

FIG. 2.1 – le modèle de composant

Dans le rapport "Component Object Model" de Microsoft, un composant est défini comme "une pièce de logiciel compilé offrant un service" [Microsoft, 1996]. Cette définition n'est pas claire car elle ne caractérise pas le comportement interne ainsi que l'interaction du composant avec son environnement. De plus, elle ne tient pas compte du support d'exécution. De ce fait, des simples programmes peuvent être considérés dans ce cas comme des composants !

Dans [D'Souza and Wills, 1998], D'Souza et Wills définissent un composant comme "une partie de logiciel ré-utilisable". Cette partie, à développer, est composée avec d'autres composants pour former l'application. Elle peut être adaptée à une application mais son code ne peut pas être modifié.

Dans [Aoyama, 1998], Aoyama distingue le concept de composant de celui d'un objet. Il considère que les composants peuvent être composés en exécution sans le besoin d'être compilés. De plus, il sépare les interfaces de l'implementation de sorte que la composition est réalisée sans prendre en compte l'implementation en détail.

De nos jours, la définition jugée (par les experts du génie logiciel) la plus précise sur les composants est proposée dans [Szyperski, 1998] : "*un composant est une unité de composition ayant* ***des interfaces*** *spécifiés à l'aide* ***de contrats*** *et* ***des dépendances de contexte explicites****. Cette unité à déployer indépendamment des autres composants est le sujet d'une composition à travers ses interfaces*".

Dans ce manuscrit, nous considérons un composant comme une boîte noire caractérisée par (figure 2.1) :

- **Une implémentation** sous la forme d'un code exécutable supportant ses fonctionnalités. Selon la technologie utilisée, ce code est sous la forme binaire, bytecode, C compilable, JAVA compilable, etc.
- **Un ensemble d'interfaces** (point de contacts) d'entrée/sortie assurant les interactions du composant avec son environnement. Ces interactions doivent respecter un contrat (contraintes) défini dans le cahier des charges. Ce contrat représente un ensemble de propriétés fonctionnelles et extra-fonctionnelles (temporelles, Qualité de Service, fiabilité) [Projet-Artist, 2003].

D'autre part, le contrat d'un composant est généralement spécifié sur ses interfaces [Projet-

Artist, 2003]. Dans [Beugnard *et al.*, 1999], on propose une classification partageant un tel contrat en 4 niveaux de propriétés.

- **Niveau 1. propriétés syntaxiques.** Pour garantir une interaction correcte avec l'environnement, le premier niveau décrit les propriétés définissant la syntaxe des procédures à exécuter dans le composant.
- **Niveau 2. propriétés fonctionnelles sur les données.** Ces propriétés définissent les plages des valeurs des données à échanger avec l'environnement.
- **Niveau 3. propriétés fonctionnelles sur les opérations.** Ces propriétés définissent l'ordre des différentes interactions entre le composant et son environnement. Plusieurs formalismes existent pour modéliser ce type de propriétés comme les machines à état (les automates temporisés, les réseaux de Petri), la logique temporelle, l'algèbre de processus, etc.
- **Niveau 4. propriétés extra-fonctionnelles.** Les propriétés de ce niveau définissent les contraintes temporelles sur les interactions entre le composant et son environnement. Des bornes minimale et maximale sont décrites pour définir la date d'exécution au plus tôt et au plus tard de l'interaction. Dans ce même niveau, d'autres propriétés sont à décrire comme les propriétés de qualité de service (QoS). Une qualité d'un composant est considérée comme une fonction associant à chacune de ses réactions un degré de son comportement idéal selon un échelle bien défini. Toujours, dans le cadre des propriétés extra-fonctionnelles, on décrit aussi des propriétés de fiabilité sur le composant. Une telle propriété permet de garantir son fonctionnement correct durant un certain temps.
  De nos jours, il n'existe aucun travail tenant compte des propriétés de ce niveau [Crnkovic and Larsson, 2002]. Dans ce manuscrit, nous nous intéresserons à la définition de propriétés temporelles. Ces propriétés définissent des bornes maximales sur les temps de réponse d'un composant donné.

2.3 Formalisation de la notion de composants

De nos jours, plusieurs formalisations ont été proposées pour spécifier des composants ; elles s'adressent toutes à des composants logiciels.

2.3.1 Modèles de composants

Dans [Moschoyiannis *et al.*, 2003], les auteurs ont défini un composant comme une unité fournissant et demandant des services à d'autres composants. Les services offerts sont disponibles dans des interfaces de *sortie* alors que les services demandés sont obtenus grâce à des interfaces d'*entrée* [Filipe, 2002a; Luders and Lau, 2002].

Selon cette formalisation, un composant est une boîte noire interagissant avec le monde extérieur grâce à des ports d'entrée et de sortie. Chaque port correspond à une interface. On dénote par I l'ensemble (fini ou infini) d'interface du composant. On dénote par O_p l'ensemble des méthodes interagissant avec ces interfaces. Le composant est formalisé à l'aide de ce tuple.

$$\Sigma = (P_\Sigma, R_\Sigma, \beta_\Sigma) \text{ avec,}$$

- $P_\Sigma \subseteq I$: l'ensemble des interfaces d'entrée.
- $R_\Sigma \subseteq I$: l'ensemble des interfaces de sortie.
- β_Σ : $P_\Sigma \cup R_\Sigma \rightarrow O_p$ l'ensemble des méthodes du composant interagissant avec ces interfaces.

Dans [Jifeng *et al.*, 2003], le composant est formalisé comme un tuple :

$$P =< C, O, I > \text{ avec,}$$

- C : un contrat sur les différentes interfaces du composant. Chaque interface contient des méthodes correspondant à des programmes localisés dans le composant.
- O : un ensemble d'interfaces de sortie du composant. Ces interfaces offrent des services à l'environnement externe. Elles ne contiennent que des méthodes publiques.
- I : un ensemble d'interfaces d'entrée au composant.

Dans [Ehrig *et al.*, 2002], un composant $Comp$ est défini comme suit :

$$Comp = (IMP, EXP, BOD, imp, exp) \text{ avec,}$$

- IMP : l'interface d'entrée. Elle fournit l'accès aux différentes fonctionnalités du composant.
- EXP : l'interface de sortie. Elle supporte les services offerts par le composants.
- BOD : l'ensemble de toutes les fonctionnalités du composant.
- imp : une fonction associant à une méthode de l'interface d'entrée l'algorithme correspondant. Cet algorithme implémente une fonctionnalité du composant.
- exp : une fonction associant à un algorithme du composant l'ensemble des méthodes de sortie à lancer pour interagir avec l'environnement externe.

Dans [Filipe, 2002b], un composant CS est spécifié à l'aide de la logique temporelle MDTL [Filipe, 2000]. Il est formalisé de la façon suivante :

$$CS = (\theta, Ax)$$

- θ : l'ensemble des signatures du composant. Chaque signature représente une vocabulaire de symboles décrivant son comportement. Ces signatures sont ou bien d'entrée ou bien de sortie. Les signatures d'entrée représentent les interfaces supportant l'accès au composant. Les signatures de sortie représentent les interfaces offrant des services du composant.
- AX : l'ensemble des axiomes du composant. Ces axiomes sont des formules à déduire à partir des signatures. Ces formules décrivent le comportement du composant.

Dans toutes les formalisations citées, la spécification des interactions entre composants n'a pas été clairement détaillée. En effet, la composition des composants n'est pas suffisamment formalisée pour spécifier de telles interactions. De ce fait, il n'est pas possible de représenter tout comportement fonctionnel d'une application à base de composants en utilisant ces formalisations. En particulier, la spécification d'un comportement conditionnel est difficile à spécifier et, donc, à réaliser.

Dans [Sifakis, 2005], une définition plus complète a été fournie ; le modèle proposé apporte une solution au problème identifié dans les travaux cités ci-dessus. C'est ce modèle que nous

présentons dans ce qui suit. La définion proposée dans [Sifakis, 2005] considère qu'un *composant* est composé :

- d'un ensemble de sous-composants,
- ou d'un ensemble d'actions supportant ses fonctionnalités. Ces actions, correspondant à l'exécution d'algorithmes du composant, peuvent être des actions internes ou des actions externes interagissant avec l'environnement.

Une *application* est un ensemble de composants interagissant ensemble. Cette modularité permet de définir différents niveaux de granularité réduisant ainsi la complexité de conception.

En prenant en compte cette caractérisation, un composant est alors la superposition de trois modèles :

- **un modèle de comportement :** il décrit le comportement dynamique du composant ; ce modèle est représenté à l'aide d'un système état-transition,
- **un modèle d'interaction :** il décrit les interactions entre sous composants ; ces interactions sont spécifiées sous la forme de *connecteurs*. Un connecteur est un ensemble d'actions de différents sous-composants. L'exécution simultanée de certaines actions d'un connecteur est une interaction entre les sous-composants correspondants.
- **un modèle d'exécution :** il réduit le non-déterminisme au niveau de l'exécution des sous composants en spécifiant un ordonnancement pour leurs différentes actions.

2.3.2 Modèle de composants pour leur composition

Le modèle de composant proposé dans [Goessler and Sifakis, 2002a; 2002b; 2003] sépare les différents aspects d'un composant en spécifiant séparément son comportement dynamique, les interactions de ses sous-composants et leurs exécutions. Cette séparation est rendue nécessaire pour spécifier tous les comportements possibles au sein d'une architecture donnée.

Nous présentons dans ce qui suit les différents modèles spécifiant un composant d'une application donnée. Nous nous basons sur un exemple d'école proposé dans [Goessler and Sifakis, 2002a] pour expliquer les principes régissant ces modèles. L'application considérée spécifie un modèle "1 Producteur - 1 Consommateur". Cette application est décrite au niveau le plus haut par un seul composant qui est réalisé par la composition de deux sous-composants *Producteur* et *Consommateur* (voir figure 2.2). Ces deux sous-composants interagissent ensemble pour supporter les fonctionnalités du composant composé. En fait, le *Producteur* répète indéfiniment "produire une information ; déposer une information" tandis que le *consommateur* répète indéfiniment "retirer une information ; consommer une information". Dans cet exemple, les deux sous-composants doivent se synchroniser pour "échanger" l'information déposée par l'un et retirée par l'autre.

Le modèle d'interaction

Soit une application contenant K composants $Comp_1, ..., Comp_K$. Chaque composant $Comp_i$ ($i \in [1, K]$) contient un ensemble d'actions A_i.

Nous dénotons par A l'ensemble de toutes les actions de l'application ($A = \cup_{i \in K} A_i$).

On définit un connecteur c tout sous ensemble non vide de A tel que :

$$\forall i \in K, |A_i \cap c| \leq 1$$

Un connecteur définit un ensemble d'actions pouvant interagir ensemble. Étant donné un connecteur c, on définit une interaction α un sous ensemble d'actions de c.

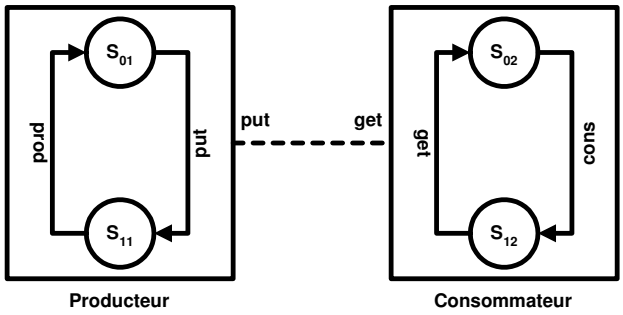

FIG. 2.2 – Exemple de composants *producteur/consommateur*

$$\alpha = a_1|....|a_n, \{a_1, ..., a_n\} \subseteq c$$

L'opérateur "|" est un opérateur associatif et commutatif. Il spécifie des actions interagissant ensemble dans l'application. D'autre part, si $\alpha = a_1|....|a_n$ est une interaction d'un connecteur c, alors tout sous ensemble de $\{a_1, ..., a_n\}$ est aussi une interaction de c.

D'autre part, une interaction est complète (resp, incomplète) si toutes (resp, quelques unes de) ses actions doivent être actives pour se réaliser. À titre d'exemple, l'interaction *send/receive* entre composants est complète. En effet, la réalisation de l'interaction n'est faite que si les deux actions sont actives. Dans certains cadres d'études (communications bufferisées asynchrones), les interactions de l'application ne sont pas complètes.

Dans l'exemple du *producteur/consommateur*, il existe un seul connecteur $c = \{put, get\}$. De ce fait, une seule interaction complète existe $\alpha = put|get$. Cette interaction sous la forme d'un rendez-vous est nécessaire pour assurer le comportement correct du composant composé.

Le modèle de comportement

Pour modéliser le comportement fonctionnel d'un composant, [Goessler and Sifakis, 2002a] propose un modèle basé sur le système de transition suivant :

$$B = (Q, I(A), \rightarrow) \text{ avec,}$$

- Q : un ensemble d'états,
- $I(A)$: un ensemble d'interactions,
- $\rightarrow$: une relation de transition dans $QxI(A)xQ$,

Dans [Goessler and Sifakis, 2002a], on propose une méthode générant le modèle de comportement d'un composant composé à partir de ses sous composants. Dans la figure 2.3, [Goessler and Sifakis, 2002a] présente le modèle de comportement du composant composé *producteur/consommateur*. Notons qu'une interaction (interne) entre les deux sous-composants est à réaliser en utilisant les actions *put* et *get*.

Le modèle d'exécution

Dans [Goessler and Sifakis, 2002a], les auteurs se basent sur une politique à priorité fixe pour générer le modèle d'exécution d'un composant. Ce modèle décrit un ordre sur l'exécution des différentes actions.

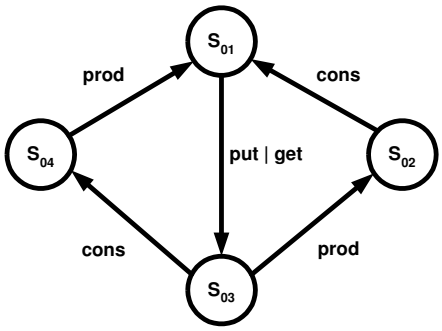

FIG. 2.3 – Le modèle de comportement composé des composants *producteur/consommateur*

Pour ordonnancer toute l'application, une composition des différents modèles d'exécution permet la génération d'un ordre d'exécution de toutes ses actions. Cet ordre garantit la correction temporelle tenant compte des contraintes temporelles.

Conclusion

En conclusion, le modèle proposé dans [Goessler and Sifakis, 2002a] a bien défini le modèle d'un composant. De plus, il a fourni une solution nécessaire pour la spécification des interactions inter-composants dans une application.

Néanmoins, il n'est pas facile d'utiliser ce modèle académique pour le développement d'applications *à comportement conditionnel* imprévisible hors-ligne [Baruah, 2003]. En effet, il n'est pas possible de prévoir tous les comportements possibles d'un composant pour valider le comportement de toute l'application. De plus, le modèle d'exécution comme proposé dans [Goessler and Sifakis, 2002a] n'est pas souvent utile pour satisfaire les contraintes temporelles considérées. En effet, l'application d'une politique d'ordonnancement à priorité fixe ne sert pas souvent à l'optimisation des temps de réponses de tout le système [Cottet *et al.*, 2000].

De ce fait, l'utilisation de ce modèle formel de composant n'est pas possible pour le développement d'applications de contrôle industriel critiques et à comportement conditionnel.

2.4 Les langages de description d'architecture

Un langage de description d'architectures *LDA* est un langage décrivant l'architecture logicielle d'une application sans présenter les détails algorithmiques de ses différentes parties. Cette description supporte les activités de structuration d'une description lors de toutes les étapes intermédiaires entre la phase d'expression des besoins et la phase d'implémentation.

Une architecture logicielle est composée en général d'éléments complexes ou primitifs interconnectés. Un *élément* (appelé généralement tâche, module, composant selon les langages) est une unité de traitement possédant une interface constituée de *points d'interaction* ou *ports*. Cette interface lui permet de communiquer avec son environnement. L'interconnexion entre les éléments d'une application peut être réalisée simplement par une liaison ou par des connecteurs. Ces connecteurs spécifient les mécanismes de synchronisation et d'échange des informations entre éléments.

Pour prendre en compte le cahier des charges d'une application, les *LDA* permettent aussi la description de plusieurs propriétés sur des éléments la constituant. Ces propriétés, implémentant en particulier des contrats, sont spécifiées sur les interfaces des éléments.

2.4.1 Caractéristiques

Les langages de description d'architectures ont plusieurs caractéristiques rendant leur exploitation applicable par l'industrie du logiciel, en particulier. En effet, la nature graphique de ces langages, la méthodologie de description, le concept de modularité et de hiérarchisation qu'ils observent ainsi que la possibilité de ré-utilisabilité qu'ils favorisent sont des caractéristiques réduisant la complexité de description d'une application donnée.

Syntaxe graphique

Les langages *LDA* disposent, en plus d'une syntaxe textuelle, d'une syntaxe graphique pour décrire des architectures logicielles. De plus, l'existence d'éditeurs spécialisés supportant cette description graphique offre de nombreux avantages aux concepteurs d'applications. Grâce à eux, un utilisateur est capable de manipuler ces langages sans avoir à acquérir une syntaxe textuelle qui peut être lourde. Ceci facilite la lisibilité et donc la maintenance de telles architectures.

D'autre part, les *LDA* autorisent le développement des systèmes complexes ou de grande taille en offrant une vue globale de l'application qui spécifie toutes ses fonctionnalités. Cette vue globale permet au concepteur de se focaliser sur la squelette de l'application sans prendre en compte des détails algorithmiques [Deplanche and Durand, 1999].

Principes de description

Afin de réduire la complexité de la description architecturale, les *LDA* assurent la génération de plusieurs vues (facettes) de l'application décrivant ses différentes parties.

Ces différentes facettes permettent au concepteur de compléter progressivement la description de son application. Cela en séparant le comportement global des détails architecturaux spécifiques à chaque facette [Deplanche and Durand, 1999].

Dans le langage de description utilisé par la société "Rational Software" [Krutchen, 1995; Deplanche and Durand, 1999] il existe cinq vues :

- **la vue logique** décrivant les besoins logiciels ;
- **la vue procédurale** décrivant les aspects de concurrence et de synchronisation ;
- **la vue de distribution** spécifiant la projection du logiciel sur le matériel ;
- **la vue de développement** décrivant l'organisation statique des composants logiciels ;
- **les scénarios** représentant la mise en pratique des quatre vues précédentes.

Modularité et réutilisation

Les langages *LDA* supportent la notion de modularité apportée par le concept de composant logiciel. Un tel composant est une boîte noire interagissant avec l'environnement grâce à ses interfaces. Cette séparation entre le comportement et l'implémentation d'un composant permet au concepteur de spécifier le squelette de l'application comme une composition de composants.

D'autre part, les *LDA* supportent la réutilisation de composants spécifiés dans d'autres applications pour réduire la durée de développement. De ce fait, chaque composant doit être développé indépendamment du contexte de son application. La réutilisation est une propriété intrinsèque des *LDA* qui offrent des mécanismes de typage et de création d'instances de composants.

Hiérarchisation

Pour réduire la complexité de la description, les *LDA* supportent la décomposition de l'application à différents niveaux hiérarchiques. Chaque niveau représente un degré particulier d'abstraction. En complément, ces langages permettent la manipulation de composants logiciels composés ou élémentaires supportant ainsi la spécification d'une application par raffinements successifs.

Un composant élémentaire est alors défini comme une unité non décomposable supportant une fonctionnalité élémentaire de l'application. Un composant composé contient (ou encapsule) un ensemble de composants composés ou élémentaires. Un tel composant apparaît comme un composant élémentaire à son environnement. Son interface constitue une voie d'accés aux modules encapsulés [Deplanche and Durand, 1999].

2.4.2 Exemple de langages de Description d'Architectures

Au début des années 80, la fonction principale des langages de description d'architectures *LDA* consistait à identifier les atomes d'exécution d'une application [Deplanche and Durand, 1999]. L'objectif était tout simplement de spécifier la distribution de l'application sur divers dispositifs. Des langages de programmation étaient, alors, utilisés en conjonction avec les *LDA* pour concevoir complètement l'application.

Au début des années 90, certains *LDA* ont intégré une approche plus conceptuelle et plus formelle pour la description d'une architecture. Cette approche repose sur le concept de composant qui permet la description modulaire et méthodique de l'application. Cette description spécifie en particulier les connexions entre les différents composants pour vérifier les contrats correspondants cités dans le cahier des charges.

Nous présentons dans ce qui suit les langages de description d'architectures les plus connus dans la communauté du temps réel.

- **Le langage Darwin.** Ce langage est une extension du langage CONIC [Kramer *et al.*, 1989]. Il permet la description d'une application à base de composants hiérarchiquement interconnectés. Il autorise, en particulier, l'instanciation dynamique de ces composants.
  Deux types de composants sont utilisés :
  ** **Le composant composé :** encapsule des instances de composants composés ou primitifs.
  ** **Le composant primitif :** encapsule le code réalisant une fonctionnalité élémentaire de l'application.

  Dans chaque composant de l'application (composé ou primitif), une interface est définie en termes de services fournis et de services requis. Les connexions entre composants consistent à relier les services offerts et fournis des composants. Le protocole de communication sur ces connexions est uniquement de type "envoi de message".
  L'environnement *REGIS* [MAGEE *et al.*, 1995] est généralement utilisé avec le langage Darwin. Il permet le développement des composants primitifs en C++. Dans ce cas, l'implémentation des composants composés est spécifiée en langage Darwin.

- **Le langage Ocl.** C'est un langage s'inspirant du langage Darwin. Il est dédié à la description de composants logiciels dans l'environnement de développement OLAN [Bellissard and Riveill, 1995].
  Selon ce langage, une application est perçue comme une hiérarchie de composants interconnectés. Ces composants définissent certains niveau de la granularité.
  L'interface d'un composant peut contenir :
  - ** **des services :** fonctions qui peuvent être fournies ou demandées par un autre composant ;
  - ** **des notifications :** événements déclenchant l'exécution d'un ou de plusieurs algorithmes du composant ; Ces algorithmes définissent des réactions suite à des notifications de l'environnement ;
  - ** **des attributs :** variables typées définissant des propriétés (qui peuvent être changées au cours de l'exécution) du composant.

  D'autre part, une connexion entre instances de composants est assurée par une instance d'un *connecteur*. Ce connecteur spécifie des règles d'interaction ainsi que le protocole employé au niveau du système d'exploitation [Durand, 1998].
  Les règles d'interaction entre composants définissent :
  - ** **la conformité des interfaces :** type échangé, homogénéité des connexions.
  - ** **le type d'interaction :** asynchrone, synchrone, diffusion d'événements, "Remote Procedure Call", etc.
  - ** **la spécification du comportement :** contraintes sur la qualité de service.
- **Le langage Durra.** c'est un langage permettant la description d'applications distribuées sur des machines hétérogènes [Barbacci *et al.*, 1991; 1993]. Ces applications sont tolérantes aux fautes et elles sont décrites à l'aide de composants.
  Selon ce langage, un composant est soit une tâche composée ou primitive, soit un canal de communication entre tâches. Notons que des configurations exclusives sont à réaliser pour décrire la composition des composants.
  Enfin, nous précisons que le langage Durra est intégré dans un environnement dont le cycle de développement s'appuie sur le modèle en spirale. De plus, l'implémentation des composants primitifs est réalisée en langage Ada.
- **Le langage Rapide.** C'est un *LDA* permettant la description de l'architecture d'une application distribuée [Luckham *et al.*, 1995; Luckham and Vera, 1995]. Ce langage permet la création de prototypes exécutables. Le résultat de l'exécution forme un POSET (Partially Ordred Set) qui est l'ensemble des événements produits pendant l'exécution avec leurs relations causales et temporelles [Durand, 1998].
  Pour la description d'une application, le langage Rapide dispose de cinq langages de descriptions distincts.
  - ** **le langage des types :** pour la description des interfaces des composants ;
  - ** **le langage d'architecture :** pour la construction des connexions entre les interfaces ;
  - ** **le langage des modules exécutables :** pour la programmation des composants ;
  - ** **le langage des contraintes :** pour la description des composants en terme de POSET ;
  - ** **Le langage de gabarit :** pour la description des événements caractérisant le comportement de l'application.

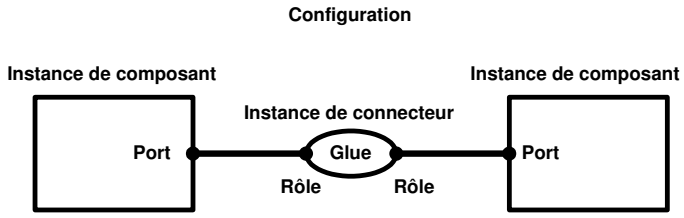

FIG. 2.4 – Une composition de composants selon le langage Wright

Selon le langage Rapide, un composant est défini par une interface et un module. L'interface définit les caractéristiques de toute interaction avec les autres composants. Ces caractéristiques sont exprimées en termes de services requis et offerts. D'autre part, le module contient,

** soit un code exécutable décrit par le langage des modules ;

** soit un ensemble de composants décrits par le langage d'architecture.

D'autre part, les contraintes définissent le comportement visible du module d'un composant. Elles sont décrites par le langage des contraintes alors que le comportement interne est décrit à l'aide d'une machine à état.
Cette machine à état permet de :

** remplacer le comportement réel du composant s'il n'a pas été implémenté en utilisant le langage des modules ;

** vérifier la conformité du code par rapport au cahier des charges si le composant est déjà implémenté.

– **Le langage Wright.** Il s'agit d'un langage permettant la description formelle d'une application à l'aide de l'algèbre de processus CSP [Allen, 1997].
Les abstractions que propose ce langage sont (figure 2.4) :

** **Les composants :** ce sont des modules supportant les fonctionnalités d'une application donnée. Chaque composant est décrit par une interface et un comportement. L'interface est constituée de ports assurant les interactions entre le composant et son environnement.

** **Les connecteurs :** spécifient les interactions entre les composants. Un connecteur est composé de "Rôles" et d'une "glue". Un rôle décrit le comportement d'un seul composant participant à l'interaction. La glue décrit comment l'interaction entre les participants est réalisée.

** **Les configurations :** décrivent les connexions entre les composants et les connecteurs. Plus précisément, une configuration décrit les liens entre les ports des composants et les connecteurs les connectant.

– **Le langage Metah.** Ce langage permet la description logicielle (et aussi matérielle) d'un système de contrôle embarqué [Vestel, 1993; 1995]. Une telle description est faite en général pour l'analyse de l'ordonnançabilité.
Selon le langage Metah, un système de contrôle est décrit comme une architecture logicielle projetée sur une architecture matérielle. Des attributs sont fournis par le langage pour spécifier une telle projection. Notons que le langage Metah offre la possibilité de spécifier

l'architecture matérielle en termes de mémoires, d'équipements utilisés, de processeurs et aussi de réseaux les reliant.
D'autre part, ce langage offre une syntaxe textuelle et aussi graphique. Une application est décrite comme un ensemble de composants connectés et qui sont *primitifs* (appelés "processus") ou *Composés* (appelés "macros"). Un processus est activé périodiquement ou apériodiquement sur occurrence d'un événement.
Selon Metah, un composant est défini par :
- Une interface spécifiant ses points d'accès typés (moniteurs partageables, ports d'événements).
  Notons que les ports d'entrée ou de sortie sont généralement assimilés à des tampons d'entrée et de sortie.
- Un ensemble d'algorithmes supportant ses fonctionnalités.

Notons enfin que le langage Metah définit un objet particulier : le *mode*. Un *mode* est un ensemble de processus réalisant une fonctionnalité de l'application. *Les modes de l'application s'excluent entre eux.* Si un *mode* est interrompu par un autre, alors tous ses processus sont aussi arrêtés.

- **Le langage AADL.** C'est un langage inspiré de Metah. Il permet la description architecturale (logicielle et aussi matérielle) de tout système embarqué temps-réel [Dissaux *et al.*, 2004]. Au début, il était proposé pour décrire des applications dans le secteur de l'avionique.
  Selon AADL, un composant est défini classiquement comme une interface et une implementation. Il appartient à trois types de catégories : matérielle, logicielle et composite. Il est caractérisé par un certain nombre de propriétés déduites du cahier des charges du système.
  L'interface d'un composant regroupe un certain nombre de ports d'entrée et de sortie. Ces ports sont associés à des flux de données et d'événements. Ils assurent l'interaction avec l'environnement externe.
  Nous nous limitons à cet ensemble de langages de description d'architectures *LDA*. Néanmoins, ils existent d'autres langages exploités comme ACME [Abi-Antoun *et al.*, 2005], UNICON [Unicon-homepage, 2006], AESOP [AES, 2006], C2SADL [Medvidovic *et al.*, 1999], SADL [Moriconi and Reimenschneider, 1997], LILEANNA [Tracz, 1993], UML, UML2, MODECHART [Mok *et al.*, 1996], ADML [Open-Group, 2000], CLARA [Durand, 1998], COTRE [Berthomieu *et al.*, 2004], EAST-ADL [Debruyne *et al.*, 2004], etc.

En conclusion de cette partie, les langages de description d'architectures sont de plus en plus utilisés dans la communauté industrielle. Les différents services supportés ainsi que la facilité d'utilisation ont fait de ces langages un moyen efficace pour la spécification d'architectures [André *et al.*, 2005].

Néanmoins, ces langages ne fournissent pas des bibliothèques de composants ce qui ne garantit pas le critère de la réutilisabilité. De plus, les architectures de contrôle industriel se basent souvent sur des standards et des conventions confirmés par la communauté. Dans des cas industriels particuliers, certaines technologies de composants sont souvent exigées par le constructeur ce qui rend l'utilisation des LDAs non justifiée.

2.5 Les technologies de composants

De nos jours, deux grandes familles de technologies de composants existent. La première famille correspond aux technologies telles que la composition des composants est à réaliser en-ligne. La deuxième famille correspond aux technologies telles que la composition est définie statiquement hors-ligne.

2.5.1 Les technologies à composition en-ligne

La première famille à présenter s'adresse aux systèmes où l'instanciation des composants est imprévisible hors-ligne. Ces systèmes ne sont pas généralement soumis à contraintes fonctionnelles et extra-fonctionnelles fortes. Nous présentons les technologies les plus connues de cette famille.

Les "Java beans" sont des composants logiciels réutilisables. Proposés par Sun-Microsystems, ces composants sont localisés dans des clients accédant à des serveurs fournissant des services [Sun-Microsystems, 1997].

Les "Enterprise Java Beans" (EJB), quant à eux, sont aussi des composants logiciels réutilisables proposés par Sun-Microsystems. Ils sont localisés dans des serveurs de systèmes transactionnels offrant des services à des clients distribués [Sun-Microsystems, 2002].

D'autre part, la compagnie Microsoft a proposé également des technologies de composants. COM, apparue en 1995, permet le développement d'applications à l'aide de langages hétérogènes [Microsoft-Corporation, 1995]. Pour assurer l'intéropérabilité entre composants, cette technologie propose pour eux des interfaces sous la forme binaire. En 2000, une nouvelle technologie de composants appelée DCOM [Pattison, 2000] est définie pour supporter la distribution des composants sur une architecture distribuée. La technologie COM+ [Box, 2000] est proposée aussi pour supporter le concept du *conteneur*. Un conteneur est un environnement logique contenant des composants de l'application. Cet environnement offre, en particulier, à ces composants, divers services au niveau de la sécurité, la gestion transactionnelle, le partage de ressources, etc. Enfin, la technologie .Net est la dernière technologie de composants proposée par la compagnie Microsoft [.net technology, 2006]. Cette technologie est une discontinuité des technologies précédentes (COM / DCOM / COM+). Pour assurer l'intéropérabilité des composants, cette technologie se base sur un compilateur particulier transformant tout code source vers un code intermédiaire appelé "Microsoft intermediate Language" (MSIL).

D'autre part, le groupe OMG a spécifié, en 2001, un modèle complet de composant appelé CCM (CORBA Component Model) [CORBA, 1998]. Ce modèle est à utiliser dans une architecture CORBA [CORBA, 1997]. Il prend avantage des caractéristiques de cette architecture pour assembler des composants développés en différents langages. Pour gérer tous les composants d'une application, le modèle a proposé aussi le concept de conteneur. Un conteneur regroupe et sert un ensemble de composants de l'application (services de sécurités, accès à des bases de données, partages de ressources, etc).

2.5.2 Les technologies à composition hors-ligne

La deuxième famille des technologies de composants s'adresse particulièrement aux systèmes de contrôle critiques. Ces systèmes doivent respecter des contraintes strictes déduites du cahier des charges. Une composition des composants spécifiée hors-ligne permet de prédire le comportement temporel de l'application avant sa mise en service et donc, de garantir, *a priori*, les propriétés requises.

La technologie Koala

Koala est une technologie de composants proposée par la compagnie Philips, pour le développement de systèmes embarqués au sein d'appareils électroniques : télévisions, lecteurs CD, lecteurs DVD, etc. [Fioukov *et al.*, 2002; Ommering, 2002; Ommering *et al.*, 2000].

Un composant Koala est une pièce de code interagissant avec son environnement externe à travers ses interfaces. Ce code est sous la forme d'un répertoire ayant un ensemble de fichiers en C et de fichiers d'en-tête.

Deux types d'interfaces existent dans un composant :

- Les *interfaces offertes* : fournit les services du composant ;
- Les *interfaces des appels* : utilisées pour appeler les services des autres composants ainsi que les services offerts par l'environnement.

Pour assurer les interactions entre composants, la technologie propose le concept de connecteurs liant les interfaces d'appel à celles offertes.

La technologie Port Based Objects

La technologie PBO est utilisée pour le développement de systèmes de contrôle embarqués (en particulier en robotique) [Stewart *et al.*, 1997].

Un composant PBO est défini par une implémentation, des ports d'entrée et des ports de sortie. Ces ports assurent le transfert de données entre le composant et son environnement.

D'autre part, la technologie définit pour certains composants des interfaces assurant les interactions avec des capteurs et des actionneurs correspondants.

Selon cette technologie, une application est définie comme un ensemble de tâches. Une tâche PBO est un ensemble de composants. Deux tâches sont indépendantes si elles n'utilisent pas les mêmes instances de composants. Cette solution rend la ré-utilisation de composants plus facile à effectuer.

La technologie Rubus

Rubus est un des rares systèmes d'exploitation temps réel (RTOS) permettant l'utilisation des composants [Articus, 1996]. Cet RTOS applique :

- un ordonnancement statique préemptif (partie rouge de Rubus) si le système temps-réel est à contraintes strictes (Hard real-time systems) ;
- un ordonnancement à priorité fixe (partie bleue de Rubus) si le système temps-réel est à contraintes faibles (Soft real-time systems).

Dans la partie rouge de cet RTOS, un composant est considéré comme une tâche réalisant une fonctionnalité bien déterminée de l'application. Une tâche Rubus est définie par :

- une fonction implémentant sa fonctionnalité.
- un ensemble de ports d'entrée/sortie ; les ports d'entrée fournissent à la fonction les données nécessaires pour son exécution ; les ports de sortie recueillent les résultats d'exécution de la fonction ; notons que la communication entre une tâche et son environnement est non "bufferisée".

Selon le cahier des charges de l'application, toute tâche (composant) est caractérisée par :

- une date d'activation initiale ;
- une période d'exécution ;
- une échéance de fin d'exécution ;

– un pire temps d'exécution de sa fonction (WCET).

Selon la technologie Rubus, une application est un ensemble de tâches connectées et sous contraintes de précédence.

Notons enfin qu'une des limitations de cette technologie est qu'elle ne propose pas un concept de composant composé afin de fournir à l'application plusieurs niveaux de granularité.

La technologie Pecos

La technologie Pecos est proposée dans un projet industriel de recherche portant le même nom [Pecos-project, 2006].

Cette technologie permet le développement d'applications de contrôle industrielles à base de composants. Ces applications sont embarquées dans des petits dipositifs à usage personnel (téléphones mobiles, PDA, etc.) ou souvent industriel.

Le projet Pecos propose une méthode permettant le développement de telles applications tout en respectant leurs contraintes extra-fonctionnelles (contraintes temporelles, limitations de mémoires, fréquences de processeurs).

Un composant Pecos est caractérisé classiquement par :

– une interface : elle contient tous les ports d'entrée/sortie qui interagissent avec l'environnement. Une telle interaction est supportée par des *connecteurs* ;
– une implémentation supportant ses différentes fonctionnalités ;
– des attributs décrivant toutes les caractéristiques d'exécution de son implémentation, à savoir :

** date d'activation intiale.

** période.

** WCET (Worst Case Execution Time ou Pire Temps d'Exécution).

** échéance relative.

La technologie des Blocs Fonctionnels (Function Blocks)

Les Blocs Fonctionnels représentent une approche de composants proposée pour le développement de systèmes automatisés. Elle permet la modélisation de l'application ainsi que de son support d'exécution. Cette approche est normalisée et largement répandue dans le monde industriel des systèmes temps réels [Projet-Artist, 2003].

Un Bloc Fonctionnel est un module réactif supportant des fonctionnalités d'une application. Il est caractérisé par :

– un ensemble d'algorithmes supportant ses fonctionnalités ;
– un ensemble de ports d'entrée/sortie supportant ses interactions avec son environnement.

De nos jours, trois normes sur les Blocs Fonctionnels existent [Rockwell, 2006; Christensen, 2000b; Lewis, 2002]. Ces normes correspondent à trois utilisations différentes :

– **La norme IEC61131-3** : présente une approche permettant le développement de systèmes de contrôle centralisés (Programmable Logic Controllers, PLC) à base de Blocs Fonctionnels. Les blocs sont ordonnés en séquence et la synchronisation entre eux est totale.
– **La norme IEC61499** : propose une extension de la norme IEC 61131-3. Elle s'adresse aux systèmes de contrôle industriels distribués en proposant certains concepts supportant une telle distribution.

– **La norme IEC61804** : définit une méthodologie de conception d'une application distribuée à base de Blocs Fonctionnels.

Ce modèle de composants étant le sujet de nos travaux, nous renvoyons le lecteur au chapitre suivant pour une description plus complète de cette technologie et un état de l'art des travaux existants sur les Functions Blocks. Dans ce chapitre figurera plus loin une analyse critique de la norme ainsi qu'une première contribution apportant des définitions précises et imposant certaines restrictions afin de lever certains points restés obscurs dans la norme.

2.5.3 Analyses des deux technologies de composants

Dans cette section, nous avons fait un parcours sur les différentes technologies de composants. Chaque technologie est dédiée à un type particulier d'applications à développer.

Dans cette thèse, nous nous intéressons à des applications de contrôle industriel. Le comportement de ces applications est souvent critique dans certains cas. Pour garantir une correction fonctionnelle et aussi temporelle, il est classiquement nécessaire d'appliquer une validation *a priori* vérifiant les différentes contraintes décrites dans le cahier des charges. De ce fait, il est nécessaire de considérer la composition des composants hors-ligne.

En tenant en compte cet argument, nous proposons d'utiliser une technologie à composition hors-ligne. Nous tenons en particulier la technologie des blocs fonctionnels comme étant la technologie de composants la plus connue dans l'industrie.

2.6 Conclusions

Dans ce chapitre, nous avons présenté les tentatives de formalisation et d'utilisation qui existent autour de l'approche par composant. L'objectif est de prendre en compte les différentes contributions académiques et industrielles proposées.

Dans une première étape, nous avons présenté les formalisations existantes sur les composants. Vu que le concept de composition est nécessaire pour la modélisation d'une application, nous avons présenté en détail une formalisation partageant la spécification d'un composant en trois modèles. L'un de ces modèles défini l'interaction inter-composants pour assurer la composition de toute l'application.

Dans une seconde étape, nous avons présenté les langages existant pour la description d'architectures logicielles. L'intérêt de cette étude est de présenter les caractéristiques de certains langages, couramment utilisés de nos jours, dans des cadres particuliers d'applications.

Enfin, nous avons présenté les différentes technologies industrielles proposées de nos jours pour le développement d'applications à base de composants. En classifiant ces technologies en deux familles, nous avons retenu une technologie à appliquer tout au long de ce manuscrit pour le développement d'applications de contrôle industriel.

Chapitre 3

Cadre normatif de définition des Blocs Fonctionnels et proposition d'interprétation de la norme

3.1 Introduction

Ainsi que nous l'avons évoqué dans le chapitre 2, la technologie des Blocs Fonctionnels a été proposée pour supporter le développement d'applications de contrôle industriel selon une approche par composant.

La norme IEC61131 et plus particulièrement IEC61131-3 définit la notion de Bloc Fonctionnel comme un composant élémentaire, et plus précisément comme un module réactif réalisant une fonctionnalité particulière de l'application. Ce module est constitué de variables internes, de paramètres (paramètres de contrôle des algorithmes), d'algorithmes et de ports d'entrée et de sortie d'événements valués.

Elle précise également la notion de composition de composants sous la forme de Diagrammes de Blocs Fonctionnels (Function Block Diagram (FBD)).

Ces diagrammes représentent graphiquement une architecture statique par simple connexion des ports d'entrée et de sortie des différents blocs (voir figure 3.1).

La norme IEC 61131 introduit également des niveaux de granularité sous la forme de :

- Blocs Fonctionnels élémentaires : ils supportent des fonctionnalités indécomposables (atomiques) de l'application.
- Blocs Fonctionnels composés : ils sont constitués aussi de variables internes, de paramètres (paramètres de contrôles d'algorithmes), de ports d'entrée/sortie et d'algorithmes. On implémente chaque algorithme d'un tel composant à l'aide d'un langage graphique (diagramme de Blocs Fonctionnels) ou aussi à l'aide du langage SFC (Sequential Function

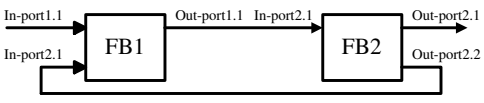

FIG. 3.1 – Un exemple d'un diagramme de Blocs Fonctionnels selon la norme IEC 61131

Chart).

Enfin, une "configuration" définit la partie logicielle d'un dispositif de contrôle, automate programmable, automate de régulation, calculateur, etc. (figure 3.3). Elle est constituée d'une ou de plusieurs ressources. Chaque ressource est une *unité d'exécution logique* regroupant un ensemble de *tâches* de l'application. Une tâche est un ou plusieurs blocs fonctionnels implémentant une fonctionalité particulière de l'application.

Nous reviendrons plus tard dans ce chapitre sur la définition du terme “ressource” ainsi que présentée dans la norme et sur l'interprétation que nous en donnerons dans le cadre de nos travaux.

La norme IEC 61131 ne définit pas une méthodologie de conception d'applications de contrôle à partir de diagramme de Blocs Fonctionnels. Pour ceci, la norme IEC 61804 propose d'harmoniser les modèles d'un système de contrôle entre l'utilisateur final, le fournisseur et le fabricant de périphériques [IEC61804-1, 2006; IEC61804-2, 2006]. Elle spécifie un système de contrôle en termes d'une architecture, de modèles et de cycle de vie [IEC61804, 2006].

Dans la figure 3.2, nous présentons le cycle de développement d'une application de contrôle et de surveillance. Les opérations suivantes sont à considérer durant tout le cycle :

- Sur le *process flow diagram*, on identifie les opérations élémentaires à contrôler.
- Sur le *P&ID*, on définit les fonctions de contrôle des opérations élémentaires du processus.
- Sur le *Control Hierarchy Diagram*, on structure la documentation des fonctions de contrôle en différentes parties.
- Pour chaque partie, les fonctions de contrôle sont spécifiées en détail grâce à un réseau de Blocs Fonctionnels qui constituent le *Functional Requirement Diagram (FRD)*. Les diagrammes *FRD* serviront de cahier des charges aux acteurs du développement (concepteurs, constructeurs et utilisateurs) pour la réalisation de l'application.

Notons enfin que les deux normes citées ci-dessus restent floues sur certains points. Dans un objectif de vérifier des propriétés comportementales et de performances, elles n'offrent pas de sémantique assez précise (en particulier sur les événements et leurs occurrences) et de plus n'adressent pas la distribution d'applications sur des architectures matérielles réparties.

En 2002, les concepts introduits par ces deux normes ont été complétés et, pour certains, précisés dans la norme IEC 61499. C'est sur cette norme que porte notre travail. Nous la détaillons par la suite (section 3.2). Dans la section 3.3, nous présentons un état de l'art sur les différents travaux et recherches effectués sur les Blocs Fonctionnels.

Comme nous l'avons déjà évoqué, les normes concernées n'apportent pas de modèle formel qui permettrait de valider un comportement d'applications ni de construire un déploiement et une allocation de ces applications qui respectent les propriétés temporelles. Aussi, dans la section 3.4, nous présentons une première contribution de cette thèse pour lever toute ambiguité dans la modélisation et le déploiement validé d'applications conçues par intégration de Blocs Fonctionnels. Enfin, nous concluons en section 3.5

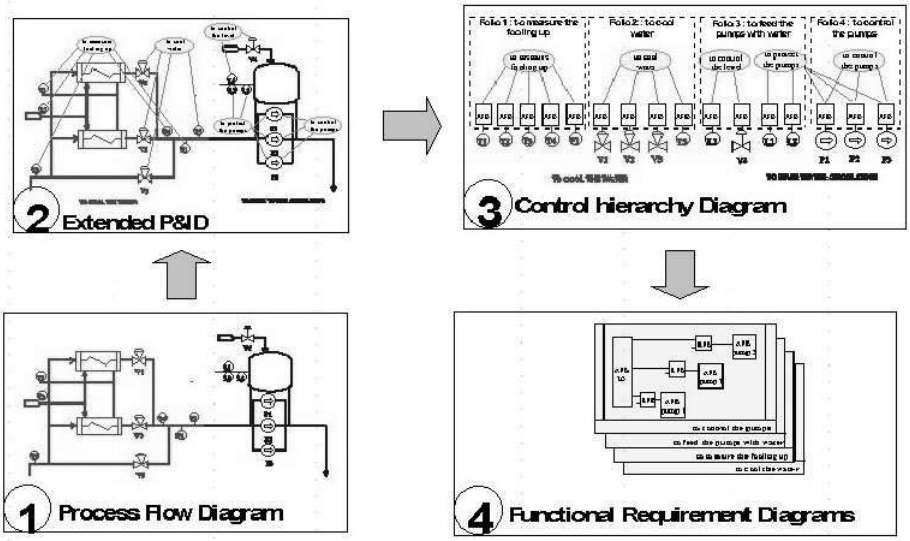

FIG. 3.2 – La méthodologie de développement d'un système de contrôle industriel selon le standard IEC 61804

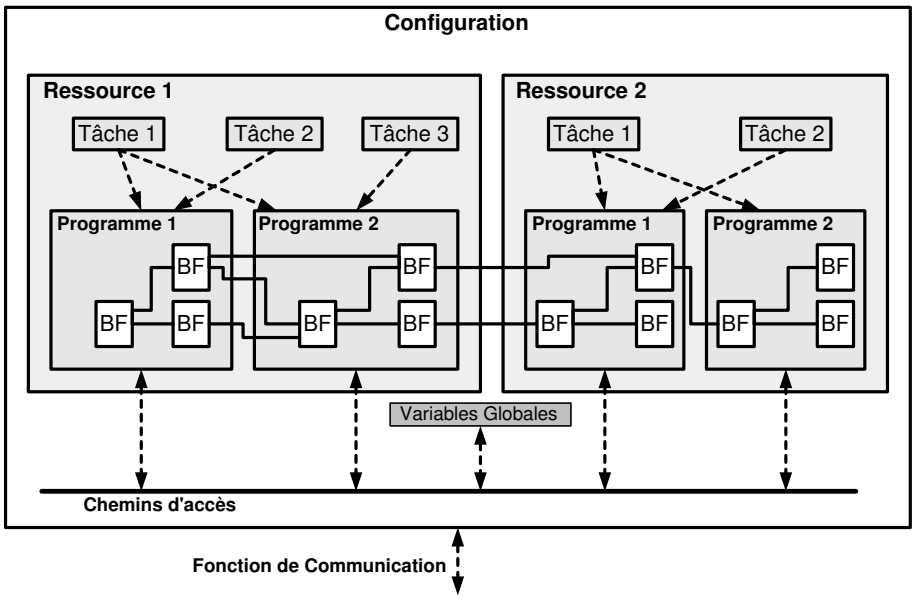

FIG. 3.3 – L'architecture d'un système industriel de contrôle selon la norme IEC 61131.

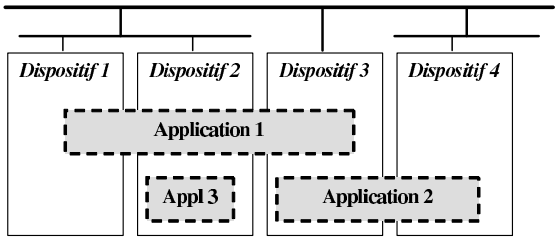

FIG. 3.4 – L'architecture d'un système de contrôle industriel selon la norme IEC 61499

3.2 Présentation générale de la norme 61499

Nous donnons ci-dessous les définitions introduites dans la norme ainsi que certains concepts de base. Ensuite, nous présentons le modèle d'un Bloc Fonctionnel élémentaire tel qu'il est proposé dans la norme. Enfin, nous présentons la notion de Bloc Fonctionnel composé [Lewis, 2002].

3.2.1 Définitions et concepts de base d'un système de contrôle industriel

Selon la norme IEC 61499, un système de contrôle industriel est un ensemble de dispositifs physiques supportant des applications logicielles distribuées. Ces applications sont spécifiées sous la forme de réseaux de Blocs Fonctionnels (figure 3.4).

Dispositif physique

Il s'agit de capteurs, actionneurs et contrôleurs munis de capacités de traitement, mémorisation et d'interfaces Entrée/Sortie.

Un dispositif physique peut supporter une ou plusieurs ressources (figure 3.5). Déjà définie dans la norme IEC 61131, une ressource regroupe des traitements spécifiés par des Blocs Fonctionnels et assure à ces traitements des services de séquencement, partage de données et partage d'E/S sur le processus physique.

D'autre part, la norme IEC 61499 propose, pour chaque dispositif, d'une part,un "Process Interface" fournissant des services aux ressources pour échanger des données avec des procédés physiques contrôlés et, d'autre part, une ou plusieurs "Ressources Communications Interfaces" fournissant des services qui supportent l'échange d'informations avec d'autres ressources d'autres dispositifs via des réseaux de communications. Les comportements internes de ces deux types d'interfaces ne sont pas décrits dans le standard.

Enfin, notons qu'une application de contrôle à base de Blocs Fonctionnels peut être distribuée sur plusieurs ressources d'un ou de plusieurs dispositifs.

Ressource

Selon la norme IEC 61499, une ressource d'un dispositif est une unité d'exécution logique (figure 3.6). Elle fournit des Blocs Fonctionnels d'interface ("Service Interface Function Blocks") standardisés et supportant les interactions entre les Blocs Fonctionnels internes et l'environnement externe.

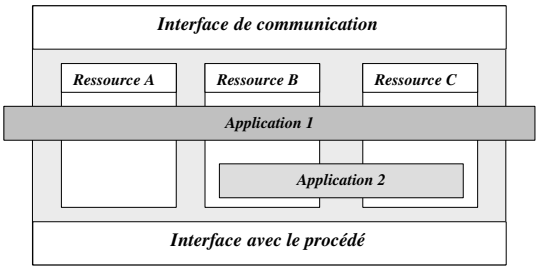

FIG. 3.5 – L'architecture d'un dispositif de contrôle selon la norme IEC 61499.

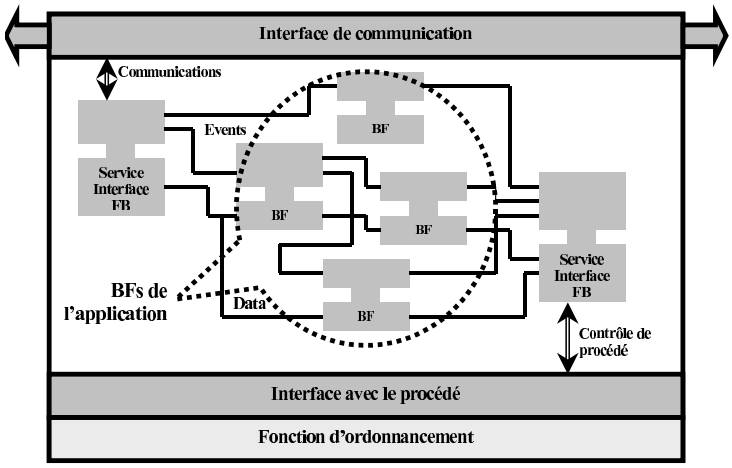

FIG. 3.6 – Le concept de ressource.

D'autre part, toute ressource d'un dispositif dispose d'une fonction d'ordonnancement contrôlant l'exécution de ses Blocs Fonctionnels internes. Cette fonction n'a pas été étudiée en détail dans les travaux existants sur les Blocs Fonctionnels.

Enfin, notons que le concept de ressource est très important vu qu'il représente l'interface entre le point de vue fonctionnel et/ou logiciel et tout ce qui concerne les systèmes d'exploitation et les protocoles de communications du support d'exécution.

Application

Elle est définie comme un réseau de Blocs Fonctionnels distribué sur une ou plusieurs ressources de plusieurs dipositifs (figure 3.7). Ainsi que nous le verrons dans la section suivante, ces Blocs Fonctionnels sont interconnectés par des flux d'évènements (contrôle de comportement) et des flux de données. Dans certains cas où l'application est distribuée sur différentes ressources, certaines interconnexions sont définies sous formes d'instances de Blocs Fonctionnels d'interfaces.

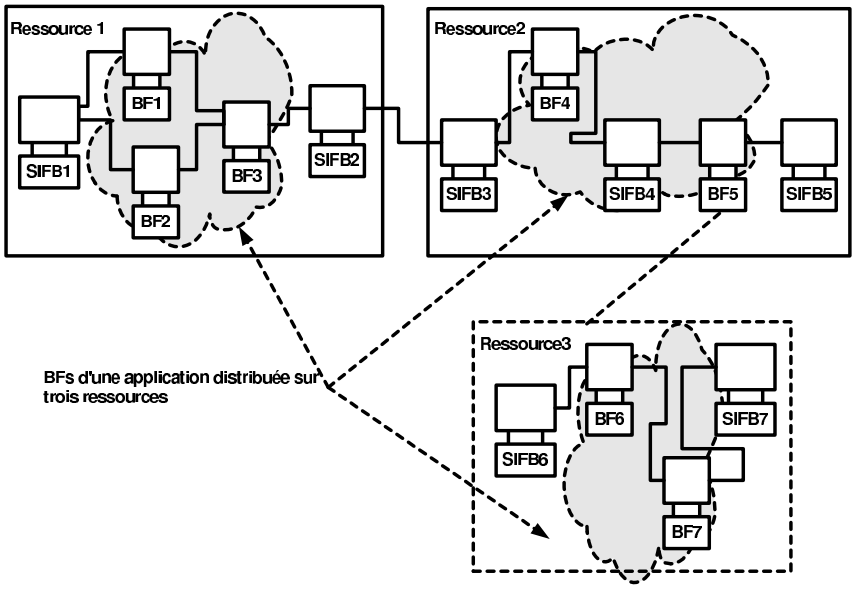

FIG. 3.7 – Le modèle d'une application de contrôle.

3.2.2 Modèle d'un Bloc Fonctionnel

Un Bloc Fonctionnel est défini comme une unité logicielle fournissant des fonctionnalités à son environnement [IEC61499-1, 2003; IEC61499-2, 2004]. Ce composant possède ses propres structures de données pouvant être manipulées par un ou plusieurs de ses algorithmes.

Deux notions sont proposées : le *type* d'un Bloc Fonctionnel analogue au concept de classe dans l'approche objet et les instances d'un Bloc Fonctionnel analogues au concept d'objet.

Selon la norme, les principales caractéristiques d'une instance de Bloc Fonctionnel sont :

- Nom de *type*.
- Nom d'*instance*.
- Des *ports d'événements (resp. de données) d'entrée*. Ces ports reçoivent des événements (resp. des données) d'autres Blocs Fonctionnels. Chaque port d'événement est associé à 0, 1 ou plusieurs ports de données.
- Des *ports d'événements (resp. de données) de sortie*. Ces ports sont utilisés pour envoyer des événements (resp. des données) à d'autres Blocs Fonctionnels. Chaque port d'événement est associé à 0, 1 ou plusieurs ports de données.
- Des *algorithmes* réalisant ses fonctionnalités.
- Des *données internes* nécessaires pour l'exécution de ses algorithmes.

La norme IEC 61499 propose deux versions différentes [IEC61499-1, 2003; IEC61499-2, 2004] pour caractériser le comportement d'un Bloc Fonctionnel. La première version ne considère que la mémorisation d'un seul événement en entrée au plus. Ceci signifie qu'à l'arrivée d'événements simultanés, un seul sera traité et les autres seront perdus. La deuxième version [IEC61499-2, 2004] spécifie la possibilité de mémorisation de plusieurs événements et données en entrée d'un Bloc Fonctionnel en considérant la solution d'un "buffer" les stockant. Cette solution permet d'éviter toute perte de données utiles pour l'application.

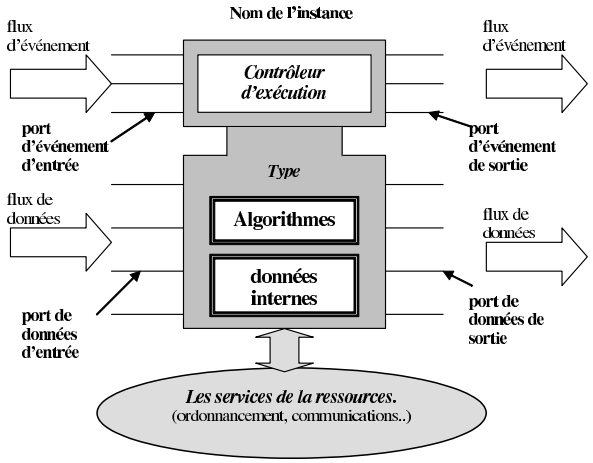

FIG. 3.8 – Le modèle graphique d'un Bloc Fonctionnel selon la norme IEC 61499.

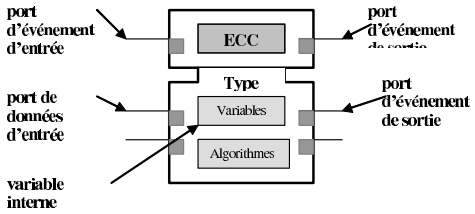

FIG. 3.9 – Exemple d'un Bloc Fonctionnel élémentaire

Un exemple de Bloc Fonctionnel est présenté dans la Figure 3.8. Le module intitulé *Contrôleur d'exécution*, noté ECC dans la suite ("Execution Control Chart") a pour rôle de contrôler le comportement du Bloc Fonctionnel (contrôle interne de flux d'événements). En effet, il associe à des événements reçus en entrée les algorithmes correspondants à exécuter. De plus, il signale des événements correspondants de sortie à la fin de l'exécution de tels algorithmes.

Un Bloc Fonctionnel interagit implicitement avec la ressource pour appeler sa fonction d'ordonnancement ou consulter ses interfaces. Notons que dans certaines technologies comme la technologie "Fieldbus", on peut accéder aux variables internes d'un Bloc Fonctionnel pour des raisons de maintenances [Lewis, 2002]. Des mécanismes d'interaction avec la ressource feront l'objet des chapitres 4 et 5.

Dans la norme IEC 61499, trois formes de Blocs Fonctionnels ont été définies [IEC61499-1, 2003; IEC61499-2, 2004; Lewis, 2002] :

- **Le Bloc Fonctionnel élémentaire :** c'est la forme la plus simple d'un Bloc Fonctionnel à instancier dans une seule ressource. Son comportement est géré par un contrôleur ECC et ses algorithmes sont implémentés dans un langage, texte structuré ou Java (figure 3.9).
- **Le Bloc Fonctionnel composé :** contient un réseau de Blocs Fonctionnels élémentaires ou

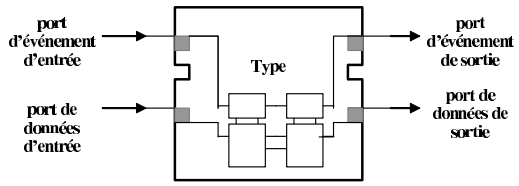

FIG. 3.10 – Exemple d'un Bloc Fonctionnel composé

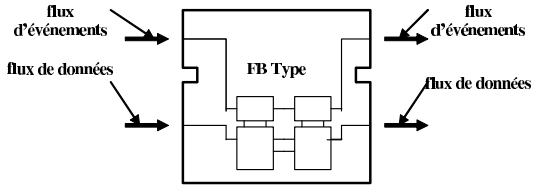

FIG. 3.11 – Exemple d'une sous-application

aussi des Blocs Fonctionnels composés. Un tel composant est à instancier impérativement dans une seule ressource. En plus et comme le Bloc Fonctionnel élémentaire, il dispose de ports d'entrée et de sortie (données et événements) supportant ses intéractions avec l'environnement (figure 3.10).

– **Une sous-application :** contient un réseau de Blocs Fonctionnels de base, Blocs Fonctionnels élémentaires, composés et même des sous-applications. Bien qu'elle puisse être distribuée sur différentes ressources, une sous-application ne dispose pas de ports d'entrée et de sortie supportant ses intéractions (figure 3.11).

Bloc Fonctionnel élémentaire

Ainsi que nous l'avons vu précédemment, un Bloc Fonctionnel élémentaire est spécifié soit textuellement, soit graphiquement [Lewis, 2002]. Deux représentations graphiques existent pour décrire son comportement : la représentation de ses interfaces externes et la représentation de son contrôleur ECC.

Dans la Figure 3.12, nous présentons une représentation graphique des interfaces externes d'un *type* particulier de Bloc Fonctionnel intitulé *Ramp* [Lewis, 2002].

Les événements d'entrée E_init et E_Run sont respectivement de type $INIT\_EVENT$ et $EVENT$. Les événements de sortie E_Rdy et E_Ex0 sont de type $INIT\_EVENT$ et $EVENT$.

L'événement E_init est associé aux variables $X0$, $X1$, $Cycle$ et $Duration$. En effet, l'occurrence de cet événement signale l'apparition de telles données sur les ports correspondants. De même, l'événement E_Run est associé à la variable PV.

D'autre part, l'événement E_Exo est associé aux deux variables Out et $Hold$, de type $REAL$ et $BOOL$, respectivement. En effet, ces données sont mises à disposition du Bloc Fonctionnel vers lequel l'événement E_Exo est émis.

La spécification du comportement interne englobe :

** les algorithmes eux-mêmes ;

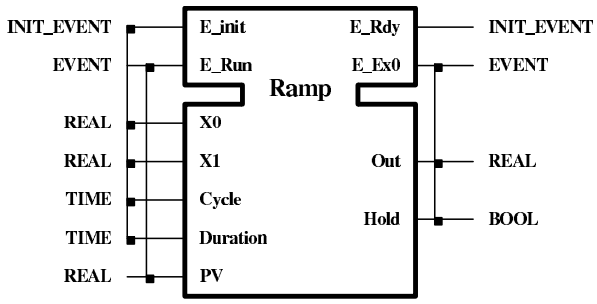

FIG. 3.12 – Le Bloc Fonctionnel Ramp

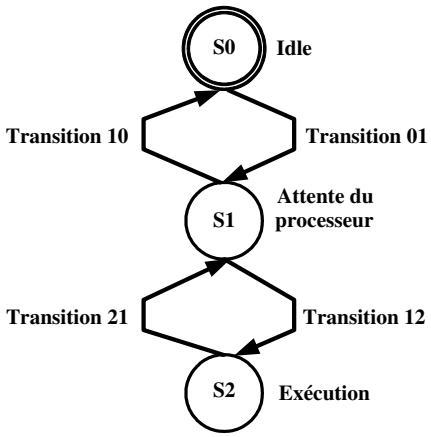

FIG. 3.13 – Le comportement d'un ECC

** le contrôle interne des flux d'événements.

Les algorithmes d'un Bloc Fonctionnel peuvent être spécifiés dans un langage de haut niveau comme le langage de texte structuré proposé dans la norme IEC 61131 [IEC61131, 1993] ou le langage Java.

Le module de contrôle d'exécution *ECC* est une machine à état similaire au SFC (Sequential Function Chart) graphique déjà défini dans la norme IEC 61131.3. Ce module spécifie (voir figure 3.13) :

** les états internes du bloc,

** comment le bloc réagit pour chaque événement en entrée,

** quels algorithmes seront exécutés à l'arrivée de chaque événement en entrée,

** quels événements de sortie à envoyer lorsque les algorithmes sont exécutés.

Selon la norme IEC 61499, un ECC peut être dans les 3 états présentés dans l'automate de la Figure 3.13. L'état *Idle* (état $S0$), est l'état pendant lequel l'ECC est inactif. À l'arrivée d'un événement d'entrée, la transition $Transition01$ est tirée. L'ECC passe à un état d'attente du processeur pour exécuter les algorithmes correspondants (état $S1$). Lorsque l'ECC gagne le processeur, la transition $Transition12$ est tirée. Il passe alors à l'état $S2$ dans lequel des algorithmes du bloc sont exécutés.

Une fois que cette exécution est terminée, la transition $Transition21$ est tirée. L'ECC se

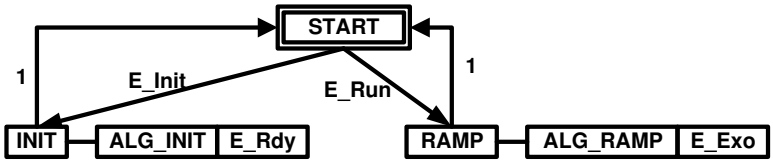

FIG. 3.14 – Le modèle d'un ECC d'un Bloc Fonctionnel

retrouve alors de nouveau dans l'état $S1$ pour envoyer les événements de sortie correspondants (ainsi que les données de sortie associées). Une fois l'envoi fait, la transition $Transition10$ est tirée et l'ECC regagne alors son état inactif (état $S0$).

Dans la figure 3.14, nous présentons l'automate, selon la syntaxe proposée dans la norme, implémentant le comportement de l'ECC du Bloc Fonctionnel RAMP présenté ci-dessus.

Dans cet automate, l'état $START$ est l'état Idle pendant lequel l'ECC est inactif. À l'arrivée d'un événement d'initialisation E_Init, l'ECC passe à l'état $INIT$ pour exécuter l'algorithme ALG_INIT et pour envoyer par la suite l'événement E_Rdy. À l'arrivée de l'événement E_Run, l'ECC passe à l'état $RAMP$ dans lequel l'algorithme ALG_RAMP est exécuté. À la fin d'une telle exécution, l'événement de sortie E_Exo est envoyé.

Nous précisons que les contraintes fonctionnelles sur les transitions d'un ECC peuvent contenir, en plus des événements d'entrée, des variables internes du Bloc Fonctionnel. Un état de l'ECC peut avoir 0 ou plusieurs actions à réaliser. En général, chaque action correspond à un algorithme et un événement de sortie signalant la fin de son exécution.

Lorsqu'un état d'un Bloc Fonctionnel est actif, toutes les actions correspondantes sont exécutées. Mais dans certains cas particuliers, un état peut ne pas avoir d'algorithme ni d'événement de sortie.

Dans la norme IEC 61499 figure une identification des événements significatifs d'un Bloc Fonctionnel. Rappelons que ce composant est réactif au sens où il manipule des flux d'événements lors de son exécution.

Plus précisémment, lorsqu'il reçoit une occurrence d'un événement $EvtIN$ associé à un port d'entrée P_{EvtIN} (figure 3.15), il effectue un traitement en utilisant les variables internes du Bloc Fonctionnel et les données présentes sur les ports de données d'entrée associés à P_{EvtIN} . A la fin d'exécution de ce traitement, il peut signaler l'événement $EvtOUT$ associé à un port de sortie P_{EvtOUT} et mettre à disposition les données correspondantes sur le ou les ports de données de sortie associés à cet événement. On distingue plusieurs dates significatives dans ce comportement (figure 3.16) :

- t_{EvtIN} : l'instant d'occurrence de l'événement $EvtIN$;
- pour chaque donnée IN_i, correspondant au port d'entrée P_i^{IN} associé à P_{EvtIN}, t_i^{IN} définit la dernière date de mise à disposition de la donnée sur P_{EvtIN} avant t_{EvtIN} ; on note t_{max}^{IN} la plus tardive de ces dates : $t_{max}^{IN} = \max_i(t_i^{IN})$;
- t_{activ} : l'instant où le module de contrôle du Bloc Fonctionnel (ECC) lance l'activation du traitement (algorithme) correspondant à une telle occurrence d'événement ;
- t_{exec} : l'instant de démarrage effectif de l'exécution ; cet instant dépend des performances du dispositif physique et de la politique d'ordonnancement ;
- $t_{finexec}$: l'instant de fin d'exécution du traitement ;

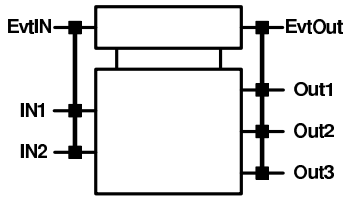

FIG. 3.15 – Un Bloc Fonctionnel élémentaire

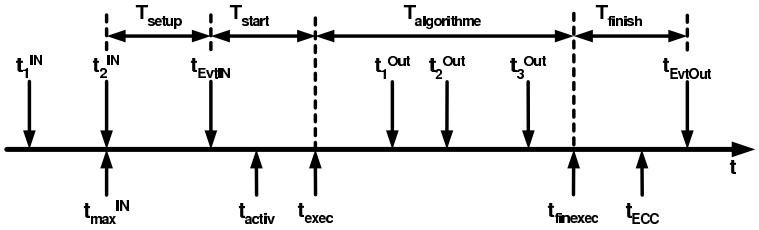

FIG. 3.16 – Le comportement d'un Bloc Fonctionnel élémentaire

- t_{ECC} : l'instant où l'ECC s'active pour conclure la réaction ;
- t_{EvtOUT} : l'instant où l'événement de sortie $EvtOUT$ est signalé sur le port P_{EvtOUT} ;
- pour chaque donnée OUT_i, correspondant au port de sortie P_i^{OUT} associé à P_{EvtOUT}, t_i^{OUT} définit la dernière date de mise à disposition, par le Bloc Fonctionnel lui-même, de la donnée sur ce port avant $t_{finexec}$.

Plusieurs intervalles de temps sont significatifs dans un tel déroulement :
- $T_{setup} = t_{EvtIN} - t_{max}^{IN}$: la durée entre la plus tardive date de réception de données et la date d'occurrence de l'événement correspondent ;
- $T_{start} = t_{exec} - t_{EvtIN}$: la durée entre l'occurrence de l'événement et le début d'exécution du traitement ;
- $T_{algorithme} = t_{finexec} - t_{exec}$: le temps d'exécution du traitement ;
- $T_{finish} = t_{EvtOUT} - t_{finexec}$: la durée entre la fin d'exécution du traitement et la production de l'événement en sortie.

Ces intervalles significatifs font l'objet d'analyse de performances pour la vérification de propriétés temporelles dans l'architecture de Blocs Fonctionnels.

Bloc Fonctionnel composé

Un Bloc Fonctionnel composé est une forme de Blocs Fonctionnels dédiée à contenir un réseau de Blocs Fonctionnels élémentaires et aussi composés (les Blocs Fonctionnels intégrés dans un Bloc Fonctionnel composé sont appelés des composants). La définition d'un *type* de Bloc Fonctionnel composé contient la déclaration des instances des Blocs Fonctionnels le constituant ainsi que des connexions de données et d'événements entre eux.

D'autre part, une application peut être décomposée en plusieurs sous-applications. Une sous-application a les mêmes caractéristiques externes qu'un Bloc Fonctionnel, mais elle peut contenir

à l'intérieur un réseau de Blocs Fonctionnels. Ce réseau peut être distribué sur plusieurs ressources (figure 3.7).

Cette notion de sous-applications supporte la conception modulaire et hiérarchique d'une application afin de réduire la complexité de son développement.

La norme IEC 61499 a spécifié plusieurs règles sur les connexions des événements et des données.

- Les règles sur les connexions des événements.
  ** Chaque port d'événement d'entrée d'un Bloc Fonctionnel composé doit être connecté exactement à un port d'événement d'entrée d'un composant ou directement connecté à un port d'événement de sortie du Bloc Fonctionnel composé. Nous notons qu'il existe un Bloc Fonctionnel particulier appelé E_SPLIT générant à partir d'un port d'événement en entrée plusieurs ports d'événements en sortie.
  ** Chaque port d'événement d'entrée d'un composant doit être connecté exactement à un port d'événement de sortie d'un composant ou à un port d'événement d'entrée du Bloc Fonctionnel composé.
  ** Chaque port d'événement de sortie d'un composant doit être exactement connecté à un port d'événement d'entrée d'un composant ou à un port d'événement de sortie du Bloc Fonctionnel composé.
  ** Chaque port d'événement de sortie d'un Bloc Fonctionnel composé est connecté exactement à un port d'événement de sortie d'un composant ou à un port d'événement d'entrée du Bloc Fonctionnel composé.
- Les règles sur les connexions des données.
  ** Chaque port de données d'entrée d'un Bloc Fonctionnel composé peut être connecté :
    *** à un ou plusieurs ports de données d'entrée de composants.
    *** et/ou à un ou plusieurs ports de données de sortie du Bloc Fonctionnel composé.
  ** Chaque port de données d'entrée d'un composant peut être exactement :
    *** Non connecté.
    *** Ou connecté exclusivement à un seul port de données de sortie d'un composant.
    *** Ou connecté exclusivement à un seul port de données d'entrée du Bloc Fonctionnel composé.

    Afin d'éviter tout problème d'indéterminisme, la norme n'a pas permis la connexion d'un port de données d'entrée à plusieurs autres port de données.
  ** Chaque port de données de sortie d'un composant peut être :
    *** Non connecté.
    *** Connecté à un ou plusieurs ports de données d'entrée de composants.
    *** Connecté à un ou plusieurs ports de données de sortie du Bloc Fonctionnel composé.
  ** Chaque port de données de sortie d'un Bloc Fonctionnel composé peut être exactement connecté à :
    *** Un port de données de sortie d'un composant.
    *** ou exclusivement à un port de données d'entrée d'un Bloc Fonctionnel composé.

Un dernier point du standard à citer est que l'association entre événements et données dans un Bloc Fonctionnel composé n'est pas la même que dans un Bloc Fonctionnel élémentaire. En effet, l'association dans un Bloc composé signifie que les données associées doivent être reçues à l'arrivée de l'événement correspondant. Mais, l'enregistrement de ces données ainsi que de l'événement est fait dans un Bloc destinataire interne. De ce fait, les ports d'un Bloc composé ne sont pas utilisés pour enregistrer mais juste pour contrôler la réception.

3.3 Travaux existants

De nos jours, plusieurs travaux de recherche ont été effectués sur la technologie des Blocs Fonctionnels [Thramboulidis, 2005b]. Ces travaux, pour certains, ont donné lieu à des outils permettant le développement d'applications de contrôle distribuées. Nous classons ces travaux en trois classes : travaux de modélisation, de génération de code et enfin d'ordre général.

Travaux sur la modélisation du comportement

Vu que le contexte général de la thèse est la validation d'applications à base de blocs fonctionnels, nous nous intéressons dans un premier temps aux travaux sur la modélisation.

Dans [Vyatkin and Hanisch, 2001], une méthode a été proposée pour la validation d'une application de contrôle développée en utilisant le standard IEC 61499. Cette méthode propose une modélisation d'un Bloc Fonctionnel à l'aide du formalisme "NET CONDITION / EVENT SYSTEMS" [Lobov *et al.*, 2003] qui est une extension du formalisme des réseaux de Petri [Peterson, 1981].

Pour spécifier le comportement d'un Bloc Fonctionnel, les modèles suivants sont à utiliser :

- Le modèle d'un port d'événement d'entrée (Event Input State Machine (EI-SM)) : c'est une machine à état spécifiant le comportement externe du port.
- Le modèle de variables d'un port d'événement d'entrée (Event Input Variables Storage (EI-VS)) : c'est un modèle spécifiant le comportement interne d'un port d'événement d'entrée en terme d'interaction avec l'ECC.
- Le modèle réactif de l'ECC (Model of EC Operation State Machine (ECO-SM)) : c'est le modèle spécifiant les interactions de l'ECC avec les ports d'événement d'entrée.
- Le modèle implémentant l'ECC (Model implementing the ECC) : c'est le modèle spécifiant le comportement de l'ECC en terme d'interaction avec la fonction d'ordonnancement et les ports d'événement de sortie.
- Le modèle d'un port d'événement de sortie (Event Output Variables (EOV) model) : c'est le modèle spécifiant le comportement du port.

Notons enfin que pour assurer une correcte interaction entre les ports d'événement d'entrée d'un Bloc Fonctionnel et son ECC, un module intitulé “ECC invocation” est proposé dans la ressource.

Dans [Stanica and Guéguen, 2004], une modélisation d'un Bloc Fonctionnel a été proposée pour valider le comportement d'une application de contrôle. Dans cette approche, un Bloc Fonctionnel est modélisé à l'aide du formalisme des automates temporisés [Alur and Dill, 1994]. La validation de l'application est faite grâce à l'outil de validation UPPAAL [Amnell *et al.*, 2001].

Les automates suivants sont utilisés pour spécifier le comportement d'un Bloc Fonctionnel :

- Pour chaque port d'événement d'entrée, un automate spécifie ses interactions avec l'environnement externe ainsi qu'avec l'ECC.
- Un automate spécifie le comportement de l'ECC. Les interactions de ce module, avec les différents ports d'événements et aussi avec la ressource, sont spécifiées dans cet automate.

Notons que des modules sont proposés dans la ressource. Ils sont modélisés à l'aide des automates suivants :

- Pour chaque Bloc Fonctionnel de la ressource, un automate d'invocation de l'ECC est à modéliser pour garantir les interactions entre les ports d'événements du bloc et l'ECC.

- Pour utiliser l'outil UPPAAL (model-checker et simulateur de réseau d'automates temporisés [David *et al.*, 2003]) ne supportant pas la diffusion, un automate est proposé pour diffuser des messages de remise à zéro vers les ports d'événements d'entrée.
- Pour assurer le fonctionnement de l'application, un automate ordonnançant les différents Blocs Fonctionnels d'une ressource est proposée. La politique d'ordonnancement n'est pas spécifiée en détail dans ce travail.

Dans [Faure *et al.*, 2002], une approche est proposée pour modéliser un Bloc Fonctionnel à l'aide du langage synchrone SIGNAL [Benveniste *et al.*, 1991]. Des horloges ont été introduites aux blocs (pour assurer une synchronisation entre l'ECC et les événements d'entrée correspondants). Ces horloges préservent un comportement réactif de l'application lui permettant le traitement de tous les événements qu'elle peut recevoir en entrée.

Dans [Hagge and Wagner, 2005], une autre approche est proposée pour modéliser le concept de Bloc Fonctionnel selon la norme IEC 61499. En se basant sur les réseaux de Petri colorés, cette approche modélise les événements et les données échangés entre les Blocs Fonctionnels comme des jetons colorés pour vérifier la totalité de l'application.

Dans [XIA *et al.*, 2004], est présentée une approche déployant un réseau de Blocs Fonctionnels dans un ensemble de dispositifs. Ce réseau de blocs est considéré comme un ensemble de tâches sous contraintes de précédences. Cette approche prend en compte certaines contraintes temporelles ainsi que des contraintes matérielles.

[Brennan *et al.*, 2002] fournit un modèle de Blocs Fonctionnels permettant la configuration et la reconfiguration d'applications de contrôle. Ces applications sont développés en utilisant le java temps réel. Dans ce même papier, on décrit en détail le système d'exploitation exécutant de telles applications.

Travaux sur la génération de code

D'autres part, plusieurs études ont été effectuées pour la génération du code d'un réseau de Blocs Fonctionnels [Thramboulidis *et al.*, 2004; Brennan *et al.*, 2002; Ferrarini and Veber, 2004; Zoilt *et al.*, 2005]. Néanmoins, les environnements les plus connus de génértion de codes sont "FBDK" [FBDK, 2006] et "Archimedes" [Thramboulidis, 2005a; Archimedes-System, 2006] (figures 3.17).

Dans [Ferrarini and Veber, 2004], les auteurs présentent et analysent un ensemble d'approches générant des codes exécutables d'application à base de Blocs Fonctionnels. De même, les auteurs dans [Zoilt *et al.*, 2005] présentent et comparent des différentes approches implémentant et ordonnançant des applications à base de Blocs Fonctionnels.

Notons enfin que d'autres travaux, sur la norme IEC 61499, ont été présentés dans le papier [Thramboulidis, 2005b].

Environnement IEC	Phase d'analyse	Phase de conception	Environnement d'exécution	Langage d'implémentation
FBDK	FBDK	Manuelle	FBRT (J2ME)	Java
Framework CORFU	CORFU-FBDK	semi-automatique	RTAI-AXE RTSJ-AXE CCM-AXE	Java, C++
Plateforme Archimedes	Archimedes ESS	automatique	RT-Linux RT-Java CORBA-CCM	Java, C++, RT-Java

FIG. 3.17 – Les environnements les plus connus du standard IEC 61499

Travaux d'ordre général

Les outils les plus connus de nos jours sont le "Function Block Development Kit" (FBDK) [FBDK, 2006] et l'environnement "CORFU-FBDK" [CORFU-FBDK, 2006] proposés par Rockwell Automation. D'autres outils ont été proposés mais leur application est réduite dans des secteurs industriels [Western-Reserve-Controls, 2006; ICS-Triplex-ISaGRAF, 2006; DACHSview, 2006].

D'autre part, la technologie des Blocs Fonctionnels a été adoptée par les approches OOONEIDA [Vyatkin *et al.*, 2005], IMS [IMS-systems, 2006] pour le développement de systèmes intelligents transactionnels distribués. De plus, elle a été adoptée par MIM [Thramboulidis, 2005a] pour le développement de systèmes de contrôle industriels (Mechatronic Systems).

Pour concevoir une application de contrôle à l'aide de la technologie des Blocs Fonctionnels, Christensen [Christensen, 2000a] propose l'exploitation d'un patron classique "Model View Controller" (MVC) pour la modélisation, la simulation et le test d'une application distribuée.

Dans [Thramboulidis *et al.*, 2004], une approche de conception d'un Bloc Fonctionnel est proposée. Elle définit une sémantique du comportement de l'ECC et permet aux algorithmes internes de générer des événements de sortie. De ce fait, la complexité de contrôle dans un bloc est réduite. Notons que cette approche a été exploité pour générer l'implementation d'un Bloc Fonctionnel.

Dans le même axe de recherche, [Panjaitan *et al.*, 2005] propose aussi une approche permettant le développement d'une application de contrôle. Cette approche, basée sur l'ordonnancement des tâches, permet la reconfiguration d'un réseau de Blocs Fonctionnels.

Dans [Cengic *et al.*, 2005], une approche de contrôle industriel est définie. Cette approche, utilisant la technologie des Blocs Fonctionnels, permet le développement d'applications distribuées. Notons qu'un outil appelé *Supremica* est proposé pour le développement rapide de telles applications.

Dans [Hussain and Frey, 2004], les auteurs indiquent que la technologie des Blocs Fonctionnels réduit la différence entre le développement de systèmes transactionnels et le développement de systèmes industriels. Cette réduction permet la réutilisation de bibliothèque de Blocs Fonctionnels dans des applications de domaines différents. Dans ce même travail, une nouvelle génération de contrôleurs est considérée pour montrer l'application facile de cette technologie dans des systèmes de contrôles industriels.

Conclusions sur les travaux menés sur les Blocs Fonctionnels

Nous avons présenté dans cette section les différents travaux de recherche effectués sur la technologie IEC 61499. Bien que ces travaux aient donné des résultats considérables utilisés dans plusieurs applications industrielles, certains concepts de la technologie n'ont pas été exploités jusqu'à nos jours :

* le concept de *ressource* n'est pas pris en compte dans la description de la distribution d'une application de contrôle. Cette distribution est considérée alors dans des dispositifs mono-ressource.
* le comportement d'un bloc fonctionnel n'est pas déterministe. De ce fait, la validation de l'application n'implique pas une sûreté de fonctionnement demandée.
* la notion de contrat temporel (propriétés temporelles offertes et requises par un composant [Projet-Artist, 2003]) n'est pas introduite sur un bloc fonctionnel.

D'autre part, la validation temporelle d'un réseau de Bloc Fonctionnels sur une architecture distribuée n'est pas complètement traitée dans ces travaux. En effet, les contraintes temporelles à vérifier, sur une telle architecture, ne sont pas clairement définies.

L'analyse d'ordonnançabilité des blocs, sur un ou plusieurs dispositifs (mono-ressource), n'est pas suffisamment étudiée. Notons, en particulier, que ces travaux n'ont pas défini la fonction d'ordonnancement d'une ressource.

Enfin, notons que le problème de déploiement d'un réseau de Bloc Fonctionnels, tout en prenant en compte des contraintes fonctionnelles (résidence, d'exclusion de résidence, etc) et temporelles, n'a été pas abordé dans ces travaux.

Dans ce manuscrit, nous apportons quelques solutions à certains de ces problèmes en proposant tout d'abord une interprétation particulière de la norme IEC61499.

3.4 Proposition d'interprétation sémantique et restrictions sur la norme

Nous évoquons dans cette section les problèmes de déterminisme du comportement d'un bloc fonctionnel [Khalgui *et al.*, 2006c]. De nos jours, ces problèmes freinent l'utilisation de la technologie IEC61499 dans certains systèmes de contrôle industriel. Pour éviter tout déterminisme, nous proposons une modélisation formelle d'un bloc fonctionnel donnant une image non ambiguë de son comportement [Khalgui *et al.*, 2004a].

D'autre part, nous enrichissons la norme en donnant au concept de ressource, une sémantique *fonctionnelle* et une autre *opérationnelle*. Ces sémantiques donnent une solution pour la distribution d'une application sur des dispositifs multi-tâches [Khalgui *et al.*, 2006b; 2006a].

3.4.1 Identification de problèmes de déterminisme du comportement

En l'état actuel, la norme IEC 61499 présente des imprécisions qui ne permettent pas de considérer un comportement déterministe d'un Bloc Fonctionnel lorsqu'il est intégré dans un réseau de Blocs Fonctionnels implémentant une application soumise à des contraintes de temps. Ce problème majeur concerne la gestion des occurrences d'événements en entrée d'un Bloc Fonctionnel.

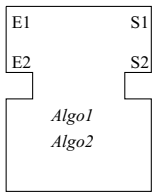

FIG. 3.18 – Le Bloc Fonctionnel BFreactif

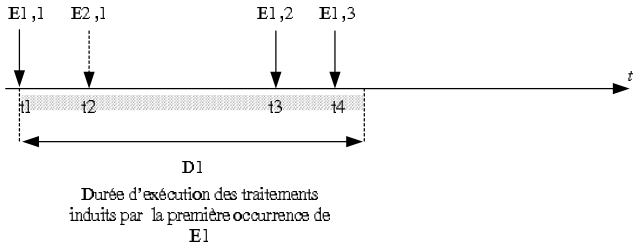

FIG. 3.19 – Un exemple de scénario

Considérons un Bloc Fonctionnel *BFreactif* (voir figure 3.18) utilisé dans une application et supposons que ce bloc est réactif à deux événements d'entrée $E1$ et $E2$. Le contrôle de *BFreactif* produit, lors de l'occurrence de l'événement $E1$ (resp. $E2$) et après exécution d'un algorithme $Algo1$ (resp. $Algo2$), une occurrence de l'événement $S1$ (resp. $S2$) ; dans cet exemple, nous ne montrons pas les données associées.

La norme IEC 61499 n'impose pas de propriétés sur le traitement des événements dans la description même d'un Bloc Fonctionnel (doivent-ils tous être traités ? toutes les occurrences doivent-elles être traitées ?). Il est juste fait référence à une possibilité de "buffer" mais sans donner d'information sur la notion de taille et à la possibilité d'affecter une priorité fixe aux événements. En particulier, il n'est pas prévu d'exprimer, lors d'une intégration au sein d'un réseau de Blocs Fonctionnels, une propriété sur la capacité de traitement par le Bloc Fonctionnel des occurrences d'événements d'entrée (toutes, toujours m sur k occurrences successives, une moyenne, etc.). Ceci induit un certain indéterminisme quant aux propriétés attendues d'une implantation ; plus exactement, cela illustre le manque de moyens formels pour caractériser un Bloc Fonctionnel de manière générique. Nous pouvons illustrer ce phénomène sur le scénario présenté à la figure 3.19.

Soit une application A décrite sous forme d'un réseau de Blocs Fonctionnels dont l'un est une instance du bloc *BFreactif* présenté ci-dessus. Le déploiement de ce réseau sur une architecture opérationnelle peut conduire à une exécution où l'événement $E1$ arrive en $t1$, $t3$, $t4$ et l'événement $E2$ en $t2$. Supposons que ce déploiement conduise à une durée d'exécution $D1$ incluant l'exécution de l'algorithme $Algo1$ pour la première occurrence de $E1$, rien dans la norme ne spécifie de règles de prise en compte des occurrences se produisant dans cet intervalle, ni même de moyens de spécifier ses propres règles.

Nous présentons dans le chapitre 4 et 5 une méthode de déploiement particulière d'un réseau de Blocs Fonctionnels soumis à des contraintes de temps ; cette méthode est dédiée à une classe d'applications et permet de garantir la prise en compte de toutes les occurrences d'événements

sous le respect des contraintes.

En nous appuyant sur cette méthode, nous proposons d'ajouter, à chaque Bloc Fonctionnel disponible au sein d'une bibliothèque, un attribut représentant une taille de buffer, pouvant prendre une valeur supérieure ou égale à 0. La valeur minimale de cet attribut qui est exigée pour respecter les propriétés temporelles exprimées sur l'application, sera calculée lors de l'instanciation du Bloc Fonctionnel au sein d'un réseau de Blocs Fonctionnels et du déploiement de celui-ci sur une architecture opérationnelle. Ce résultat permettra de vérifier si l'instance du Bloc Fonctionnel offre les ressources suffisantes en termes de taille de buffer d'événements, dans le cas où ce buffer est dimensionné dans la spécification du bloc. Dans le cas où la taille est laissée libre lors de l'instanciation, cette valeur entrera en compte alors pour l'évaluation de la ressource mémoire nécessaire. Nous fournissons au chapitre 4 une méthode pour calculer la valeur minimale de tels buffers dans le cas du type d'applications considérées.

De plus, nous proposons d'affecter des priorités relatives aux événements d'entrée d'un Bloc Fonctionnel.

3.4.2 Interprétation du concept de "Ressource" et proposition de définition

Dans la norme IEC 61499, le concept de ressource est défini ainsi :

> *Ressource* : unité d'exécution logique qui fournit des blocs d'interface ("Service Interface FBs") standardisés et supportant les interactions entre les Blocs Fonctionnels internes et l'environnement externe.

De plus, une caractéristique de la ressource, citée dans la norme est l'existence d'un ordonnancement non préemptif entre les activités des Blocs Fonctionnels qu'elle supporte.

Une première extension a été proposée dans [Lewis, 2002] :

> *Ressource* : unité logique d'exécution correspondant à des slots de temps de l'unité de calcul qui la supporte.

Cette définition reste trop pauvre pour évaluer les performances d'une application décrite sous forme de réseaux de Blocs Fonctionnels et déployée sur une plate-forme d'exécution. Aussi, nous proposons, dans le cadre de nos travaux [Khalgui *et al.*, 2006b; 2006a], une interprétation plus restreinte de la ressource qui tienne compte des spécificités que nous adressons dans cette thèse, à savoir :

- déploiement sur une plate-forme d'exécution munie de systèmes d'exploitation multitâches sur chaque station (de plus en plus d'automates programmables industriels sont munis de tels systèmes),
- obligation pour l'application de respecter des contraintes de temps de bout en bout (intervalle de temps borné entre un stimulus de l'environnement et la réponse).

> *Ressource* : unité logique d'exécution correspondant à des slots de temps de l'unité de calcul qui la supporte. **Fonctionnellement**, la ressource supporte l'exécution des algorithmes de Blocs Fonctionnels d'une application qui partagent des données et/ou contribuent au contrôle de un ou plusieurs procédés physiques spécifiques. **Opérationnellement**, une ressource est réalisée par une tâche gérée par le système d'exploitation multitâches, dénommée dans la suite du manuscrit tâche OS.

Et nous précisons les caractéristiques suivantes :
- A l'intérieur d'une même ressource, l'exécution d'un algorithme ne peut préempter l'exécution d'un autre algorithme (déjà cité dans la norme). Ceci assure l'exclusion mutuelle à des ressources partageables séquentiellement mais non simultanément par les algorithmes des Blocs Fonctionnels affectés à la ressource.
- Le cas échéant, la ressource fournit des blocs d'interface ("Service Interface FBs") standardisés et supportant les interactions entre les Blocs Fonctionnels internes et l'environnement externe.
- Enfin, les tâches OS sont indépendantes. Ceci signifie qu'une tâche OS associée à une ressource peut préempter une tâche OS associée à une autre ressource.

3.4.3 Modélisation formelle du comportement d'un Bloc Fonctionnel

Dans [Vyatkin and Hanisch, 2001; Stanica and Guéguen, 2004], le comportement considéré d'un bloc fonctionnel n'est pas déterministe (logiquement et temporellement). En effet, on n'applique pas une politique de sélection d'événements d'entrée au bloc. De ce fait, le choix d'un événement à traiter parmi plusieurs est imprévisible. D'autre part, en associant à chaque port d'événement d'entrée un port de sortie, le comportement du bloc n'est pas conditionnel. Vu ces limitations, les modèles proposés dans ces travaux ne permettent pas la modélisation de systèmes de contrôle industriel critiques.

Pour avoir la possibilité de valider le comportement de tels systèmes (fonctionnellement et temporellement), nous proposons dans un premier temps de fournir un modèle générique formel d'un Bloc Fonctionnel tel qu'il puisse donner une image non ambiguë de son comportement.

Pour séparer l'interface du composant de son comportement interne tel que géré par l'ECC, nous proposons de modéliser un Bloc Fonctionnel à l'aide :
- d'un modèle de ses ports d'entrée d'événements,
- d'un modèle de son contrôleur d'exécution, l'ECC.

Modélisation des ports d'entrée d'événements

A chaque port correspond un évènement. Comme indiqué précédemment, chaque événement possède une priorité pour ce Bloc Fonctionnel. Nous noterons *nport* le nombre de ports d'entrée du Bloc Fonctionnel et ee_1, ee_2, ..., ee_{nport} les événements d'entrée du Bloc Fonctionnel avec *priorité*$(ee_i) >$ *priorité*(ee_j) si $i < j$. Chaque port est représenté par deux modules décrivant ses réactions à l'arrivée d'une occurrence de l'événement qui lui est associé. Si on considère l'événement d'entrée ee_i, on a alors, pour le port auquel il est associé, $Port_i$:
- MR_i : module de réception des occurrences de ee_i ; ce module est en fait l'interface du Bloc Fonctionnel avec son environnement ; son rôle est de mémoriser les occurrences de

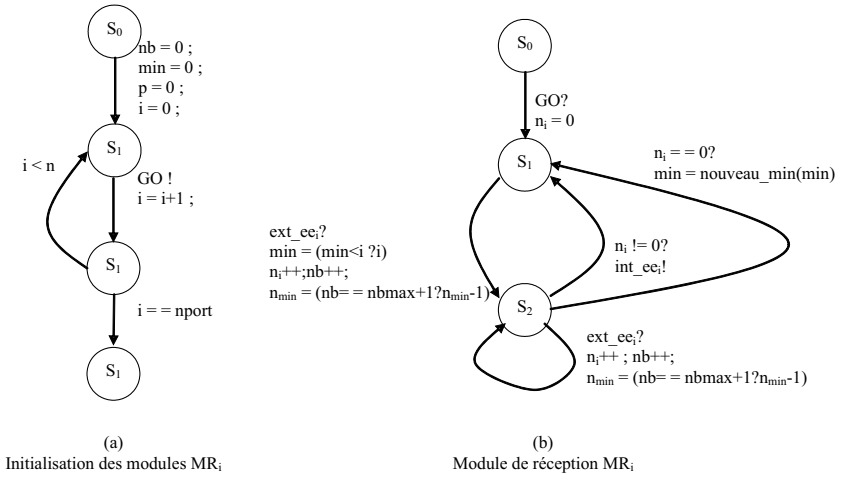

FIG. 3.20 – Modélisation du comportement sur les ports d'entrée :MR_i

l'événement qui doivent être mémorisées en attendant que l'ECC soit libre et qu'il n'y ait plus d'événements plus prioritaires non traités ;

- MI_i : module d'interaction avec l'ECC ; son rôle est la sélection d'une occurrence à traiter par l'ECC.

Ces deux modules sont modélisés par des automates temporisés. Le Bloc Fonctionnel est caractérisé entre autres par une variable indiquant la capacité du buffer c'est-à-dire de la zone de mémorisation des occurrences d'événements en entrée. Soit *nbmax* cette taille. Rappelons que nous proposerons au chapitre 4 le calcul de la valeur optimale de cette taille de buffer (valeur minimale) à effectuer lors du processus de déploiement (voir chapitres 4 et 5) afin de garantir les propriétés d'ordonnançabilité de l'application multitâches sans perte d'occurrences d'événements. Les deux automates proposés [Khalgui *et al.*, 2004a] font évoluer plusieurs variables globales au Bloc Fonctionnel :

- min : elle donne à tout instant la valeur courante de la priorité de l'événement de plus faible priorité dont une occurrence est en attente de traitement dans le buffer ; cette variable est initialisée à 0 ;
- nb : la valeur courante du nombre d'occurrences d'événements dans le buffer ; cette valeur est initialisée à 0 ;
- p : la priorité de l'événement le plus prioritaire en attente de traitement ;

et une variable locale au port $Port_i$:

- n_i : la valeur courante du nombre d'occurrences de l'événement ee_i en attente dans le buffer ; cette valeur est initialisée à 0 pour chaque événement ee_i en entrée du Bloc Fonctionnel.

L'automate proposé à la figure 3.20 modélise le comportement suivant du module MR_i : à l'arrivée de l'événement ee_i,

- n_i, le nombre d'occurrences non traitées de cet événement est incrémenté de 1 ; nb, le nombre total d'occurrences d'événement est incrémenté de 1 ;
- si le nombre d'occurrences nb dépasse la taille du buffer ($nb > nbmax$), il faut enlever une occurrence d'un événement moins prioritaire du buffer ; ceci se fait avec une mise à jour de la variable globale min ;
- si le port $Port_i$ est le plus prioritaire du Bloc Fonctionnel, le signal int_ee_i est envoyé au

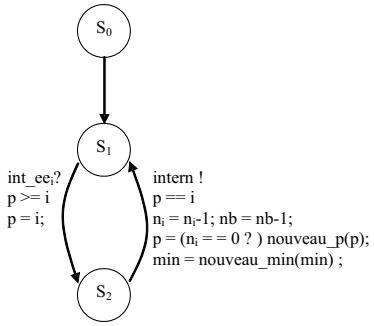

FIG. 3.21 – Modélisation du comportement sur les ports d'entrée :MI_i

module MI_i

Le module MI_i est modélisé par un automate présenté à la figure 3.21 Il réagit à l'émission du signal int_ee_i de la part du module MR_i si le port $Port_i$ est le port prioritaire courant ($p >= i$). Dans ce cas, le module MI_i émet un signal au module ECC de gestion du Bloc Fonctionnel dont une modélisation est proposée à la section suivante.

Dans les automates modélisant les comportements de MR_i et MI_i, apparaissent deux fonctions nouveau_min et nouveau_p. La fonction nouveau_min donne la valeur actualisée de *min*, lorsqu'une occurrence de l'événement ee_i a été retirée du buffer.

```
fonction : entier nouveau_min(entier i) : entier
Debut
  entier j = i ;
  booléen asuivre = (j>=0) ;
  Tant que (asuivre) faire
    Si (nj !=0)
      Alors
        asuivre = false ;
    Sinon
        j=j-1 ;
        asuivre = (j>=0) ;
  retourner j ;
Fin entier nouveau.
```

La fonction nouveau_p donne la valeur actualisée de p lors du retrait du buffer de l'occurrence de l'événement à transmettre à l'ECC pour traitement. L'algorithme de cette fonction est similaire à celui de la fonction nouveau_min.

Modélisation de l'ECC : comportement interne du Bloc Fonctionnel

Nous proposons une modélisation du comportement de l'ECC à l'aide du formalisme des automates temporisés [Khalgui *et al.*, 2004a]. Le comportement du module, lorsqu'une occurrence de ee_i ($i \in [1, nport]$) a le plus grand degré de priorité, est le suivant (figure 3.22),

* Une fois la synchronisation est effectuée avec le module MI_i (à l'aide de *intern*?), l'ECC demande le séquenceur de la ressource d'exécuter le ou les algorithmes correspondent à ee_i

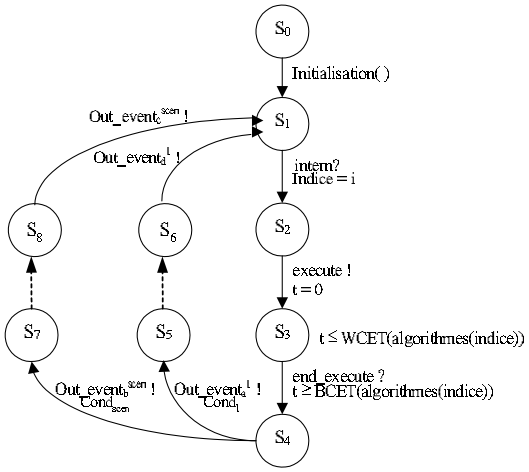

FIG. 3.22 – Modélisation du comportement de l'ECC

(à l'aide de *execute*!).

* À l'état S_3, la durée d'exécution de ces algorithmes est classiquement entre deux valeurs : le pire temps d'exécution $WCET$ et le meilleur temps d'exécution $BCET$.

* Une fois l'exécution est terminée, l'ECC (se retrouvant à l'état S_4) envoie des événements de sortie aux autres Bloc Fonctionnels. Ces événements sont à envoyer simultanément ou en exclusion : cela dépend de certaines conditions à vérifier (variables internes du bloc, résultats de l'exécution des algorithmes, etc). De ce fait, plusieurs scénarios d'envois sont possibles où chacun regroupe des événements à envoyer simultanément. Dans la figure 3.22, l'ECC a *scen* scénarios d'envois d'événements où chaque scénario $k \in [1, scen]$ vérifie une condition particulière $Cond_k$.

Analyse de l'approche de modélisation

Contrairement aux travaux proposés dans [Stanica and Guéguen, 2004; Vyatkin and Hanisch, 2001], Cette approche de modélisation propose un comportement déterministe d'un bloc fonctionnel. En effet, la politique de sélection d'événements rend un tel comportement prévisible. Ce besoin est souvent demandé dans le cadre d'un système de contrôle industriel critique.

Une fois le comportement d'un bloc fonctionnel déterministe défini, nous proposons de valider formellement une application. En tenant compte cette approche de modélisation, l'application est considérée alors comme un réseau d'automates temporisés. Nous proposons d'utiliser l'outil UPPAAL pour vérifier des propriétés fonctionnelles (absence de blocage et atteignabilité) et temporelles (bornes sur les temps de réponse de bout en bout) décrites dans le cahier des charges.

Bien que cette approche de validation exhaustive soit utile pour la validation d'un réseau de blocs fonctionnels, nous retenons les limites suivantes réduisant son efficacité :

- Le concept de "Ressource" n'est pas considéré dans cette approche. En fait, les dispositifs considérés sont uniquement des dispositifs mono-ressource.
- Il est difficile de réaliser une synthèse d'ordonnancement des blocs fonctionnels implémentant l'application. Or cette synthèse est indispensable au déploiement.
- Dans le cas d'applications complexes, la validation exhaustive entraîne une explosion combinatoire qui rend cette dernière difficilement faisable.

En considérant ces limites, nous proposons dans les chapitres suivants une approche analysant l'ordonnancement d'une application de contrôle distribuée sur des dispositifs multi-ressources. Cette approche, respectant les instructions de la norme, permet l'optimisation de l'ordonnancement des différents blocs de l'application.

3.5 Conclusions

Dans ce chapitre, nous avons présenté la technologie des Bloc Fonctionnels choisie comme la base de nos travaux de recherche. Nous avons présenté, en particulier, le standard le plus connu en industrie : l'IEC 61499. Ce standard fournit plusieurs concepts permettant le développement d'applications de contrôle distribuées. En particulier, un modèle de bloc fonctionnel composé est fourni pour assurer une modularité réduisant la complexité de conception de toute application critique.

De nos jours, plusieurs travaux de recherche ont été proposés autour de cette technologie. Nous avons présenté, dans ce chapitre, les différents travaux de modélisation, de vérification, de déploiement et de génération de code d'applications à base de Bloc Fonctionnels. Ces travaux ont donné des résultats considérables et pouvant être exploités dans l'industrie.

Néanmoins, malgré l'activité de recherche intensive sur cette technologie, plusieurs imprécisions dans le standard freinent son exploitation complète dans les milieux industriels. Dans ce chapitre, nous avons analysé ces imprécisions et nous avons défini des interprétations précisant des concepts de la norme. En particulier, nous avons proposé une modélisation du Bloc Fonctionnel assurant un comportement dynamique déterministe de toute application à développer.

Chapitre 4

Ordonnancement statique d'un réseau de Blocs Fonctionnels dans une ressource

4.1 Introduction

Selon la norme IEC 61499 [IEC61499-2, 2004], les Blocs Fonctionnels localisés dans une ressource d'un dispositif peuvent partager des données et aussi l'accès à des procédés physiques [Lewis, 2002; Christensen, 2000b]. De ce fait, leur exécution doit se faire en exclusion mutuelle [M. Herlihy, 1992].

Pour satisfaire cette contrainte, une première solution est d'appliquer des mécanismes dynamiques maintenant l'exclusion [M. Herlihy, 1992]. Dans la norme [Lewis, 2002], une autre solution est considérée en appliquant une politique d'ordonnancement non-préemptive [Cucu and Sorel, 2004; Stankovic *et al.*, 1998]. Cette solution est intéressante vu qu'elle évite le maintien explicite de l'exclusion mutuelle.

Dans ce chapitre, nous nous intéressons à l'analyse de la faisabilité temporelle [Grolleau and Choquet-Geniet, 2002] d'un réseau de Blocs Fonctionnels dans une ressource d'un dispositif donné [Khalgui *et al.*, 2005b]. L'étude de l'ordonnançabilité d'une architecture de Blocs Fonctionnels distribuée sur plusieurs ressources d'un dispositif sera étudiée dans le chapitre suivant.

Dans le cadre des systèmes de contrôle critiques, la perte d'occurrence d'événement peut être un grave problème [Khalgui *et al.*, 2006c].

Pour éviter ce problème, nous proposons une génération d'ordonnancement statique qui prend en compte :

- la taille fixe des buffers d'événements,
- les contraintes temporelles sur l'application, issues du cahier des charges.

Afin d'exploiter les divers travaux effectués dans le domaine de l'ordonnancement temps réel [Stankovic *et al.*, 1998; Baruah, 2003; Baruah *et al.*, 1999], nous proposons une approche transformant le réseau de Blocs Fonctionnels vers un système de tâches sous contraintes de précédence [Cucu and Sorel, 2005; Stankovic *et al.*, 1998]. La génération proposée d'un séquencement faisable se base sur la construction d'un graphe d'accessibilité vérifiant toutes les contraintes temporelles [Grolleau and Choquet-Geniet, 2002; M. Richard and Cottet, 2002]. Cette technique permet, en particulier, d'étudier tous les scénarios possibles d'exécution des Blocs Fonctionnels dans la ressource [Khalgui *et al.*, 2005a].

Dans le cas où un ordonnancement est faisable (i.e. tous les Blocs Fonctionnels sont ordonnançables), nous proposons une approche générant l'ordonnancement statique correspondant [Khalgui *et al.*, 2005a; Grolleau and Choquet-Geniet, 2002]. Cet ordonnancement implémente la fonction d'ordonnancement [IEC61499-2, 2004] de la ressource sous la forme d'un séquencement allouant le processeur aux Blocs Fonctionnels de cette ressource.

Notons enfin que dans le cas où la génération de l'ordonnancement est impossible pour les règles de séquencement que nous introduisons, sous des contraintes strictes de temps de réponse, nous avons développé une méthode de génération d'ordonnancement dans un mode dégradé, spécifiées par le concepteur. Dans le cas où un tel ordonnancement est trouvé, il peut correspondre à un fonctionnement éventuellement dégradé, mais admissible du système [Khalgui *et al.*, 2006d]. Dans tous les autres cas, la conception de l'application doit être revue (choisir un dispositif plus puissant, allouer le réseau de Blocs Fonctionnels à plusieurs ressources, etc.)

Enfin, rappelons que, d'une part nous nous intéressons uniquement à la classe d'applications pour laquelle les événements extérieurs arrivent périodiquement, et que, d'autre part, nous proposons une méthode qui détermine si une configuration est faisable selon nos règles de construction mais ne permet pas de conclure à la non faisabilité d'une configuration déployées à l'aide d'autres règles.

Exemple. *Tout au long de ce manuscrit, nous considérons un exemple simple d'une application de contrôle localisée dans une resource d'un dispositif. Cette application est implémentée à l'aide d'un réseau de six Blocs Fonctionnels (figure 4.1).*

Le principe de l'application est de prélever des données sur des capteurs à l'aide du bloc FB_1*. À chaque prélèvement, une exécution d'un ou plusieurs algorithmes de BFs est réalisée avant l'activation de l'actionneur correspondant (envoi de* oe_8*,* oe_9 *ou* oe_{10}*).*

Nous considérons un comportement conditionnel du bloc FB_1*. A la fin de l'exécution des algorithmes correspondant à* ie_1 *(resp.* ie_5*), l'ECC envoie, selon l'état du système, l'événement de sortie* oe_1 *ou les deux événements* oe_2 *et* oe_3 *(resp.* oe_7 *et* oe_8*). En considérant ce comportement, le fonctionnement du bloc* FB_1 *est imprévisible hors-ligne.*

D'autre part, nous supposons que le prélèvement des données est périodique. Nous caractérisons les deux événements ie_1 *et* ie_5 *de la façon suivante.*

- $r_1 = 10, j_1 = 1, p_1 = 100$
- $r_5 = 30, j_5 = 1, p_5 = 100$

où r, j, p sont respectivement la date de la première occurrence de l'événement d'entrée du réseau, la gigue sur chaque occurrence et la période entre deux occurrences successives de cet événement.

D'autre part, nous caractérisons chaque algorithme en supposant que le pire (resp. meilleur) temps d'exécution est égal à 8 (resp. 5) unités de temps.

Enfin, nous considérons des contraintes temporelles à respecter par les différents blocs de l'application. Nous supposons une borne sur le temps de réponse de bout en bout entre un prélèvement de données et l'activation de l'actionneur correspondent :

$$borne(ie_1, oe_8) = borne(ie_1, oe_9) = 100$$

$$borne(ie_5, oe_8) = borne(ie_5, oe_9) = borne(ie_5, oe_{10}) = 100$$

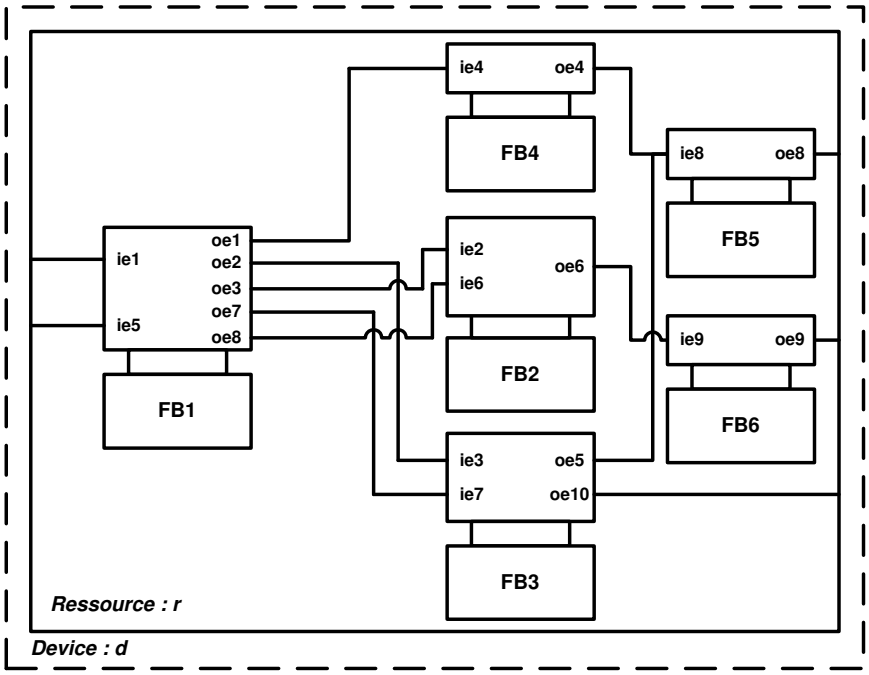

FIG. 4.1 – Exemple de réseau de Blocs Fonctionnels dans une ressource d'un dispositif donné

4.2 Quelques rappels sur l'ordonnancement de tâches dépendantes

De nos jours, plusieurs modèles de tâches (appelées actions dans ce chapitre) ont été proposés pour modéliser des systèmes temps réel [Stankovic *et al.*, 1998]. Ces tâches sont dites indépendantes lorsqu'elles n'ont entre elles ni relation de précédence ni partage de ressources [Cottet *et al.*, 2000]. Dans le cas contraire, elles sont dites dépendantes.

Dans ce chapitre, notre étude concerne les tâches dépendantes. Conformément à la norme IEC 61499, le problème de partage de ressource (données ou procédés physiques) est réglé par la politique non-préemptive statique définie pour le séquencement des actions dans une ressource (au sens de la norme IEC 61499).

Dans la théorie d'ordonnancement temps-réel, les travaux sur l'analyse de faisabilité ont été réalisés sur deux types de systèmes de tâches : les systèmes de tâches périodiques et les systèmes de tâches sous contraintes de précédence. Néanmoins, une seule contribution [Cucu and Sorel, 2005] a été proposée, jusqu'à nos jours, sur les systèmes sous contraintes de précédence et aussi de périodicité.

Dans [Cucu and Sorel, 2005], on s'adresse aux systèmes de tâches non-préemptives, périodiques et sous contraintes de précédence. Ces tâches sont soumises, de plus, à des bornes sur les temps de réponses de bout en bout. Dans [Cucu and Sorel, 2004], on propose un algorithme optimal analysant l'ordonnançabilité de ces tâches dans une hyper-période bien définie [Cucu and Sorel, 2004].

En ce qui concerne notre travail, nous précisons que ce modèle de tâches n'est pas utile pour modéliser tous les comportements possibles d'un réseau de BFs. En effet, il n'est pas possible de modéliser tous les scénarios d'envois d'événements de sortie par l'ECC d'un bloc particulier.

4.3 Le modèle d'actions d'un réseau de Blocs Fonctionnels

Dans cette partie, nous proposons de transformer un réseau de Blocs Fonctionnels localisé dans une ressource vers un modèle de tâches, appelées par la suite "actions", sous contraintes de précédence [Stankovic *et al.*, 1998; Cucu and Sorel, 2005; 2004]. Ce travail a été publié dans [Khalgui *et al.*, 2006c].

4.3.1 Action

Définition 1. Nous définissons une action *act* comme étant une exécution d'un Bloc Fonctionnel. Cette exécution correspond à l'arrivée d'une occurrence d'événement *ee* sur un des ports du bloc. Elle peut être constituée de l'exécution d'un ou plusieurs algorithmes. Ordonnancer l'exécution d'un réseau de Blocs Fonctionnels revient donc à ordonnancer les actions correspondantes.

Afin de réaliser cet ordonnancement, nous commençons par caractériser temporellement chaque action.

Caractérisation

Une action *act* est caractérisée par :

- $WCET(act)$ (resp. $BCET(act)$) : le pire (resp. meilleur) temps d'exécution de la séquence d'algorithmes correspondant à l'arrivée de *ee*. Cette durée dépend bien évidement des performances du support d'exécution (la faisabilité d'un ordonnancement et la construction statique de celui-ci sera calculée pour un support d'exécution donné).
- $pred(act)$: c'est l'ensemble d'actions à exécuter dans l'application juste avant l'exécution de *act*. Chacune de ces actions appartient à un Bloc Fonctionnel générant un événement de sortie *es* "connecté" à *ee* ($es = cause(ee)$).
- $succ(act)$: c'est un ensemble d'ensembles d'actions. Les ensembles appartenant à $succ(act)$ sont déduits de l'analyse de l'ECC présentée précédemment. Chacun d'eux correspond à une exécution possible de l'application et les actions qui le composent ne peuvent s'exécuter qu'aprés la fin d'exécution de *act*. Chacun des ensembles appartenant à $succ(act)$ correspond à un sous-ensemble d'événements de $follow(ee)$.
- (r, p, j, d) : ce sont les paramètres temporels caractérisant l'action ; les trois premiers paramètres r, p, j caractérisent l'activation de l'action (la date d'activation, la période et la gigue comme étant la variation de la période) ; l'échéance d représente une contrainte à déterminer à partir du cahier des charges et, en particulier comme nous le verrons plus loin dans ce chapitre, de la borne sur les temps de réponse de bout en bout à respecter par l'application [Liu and Layland, 1973]. Dans les sections suivantes, nous proposons une méthode calculant ces paramètres temporels. Comme décrit précédemment, nous nous intéressons aux applications activées périodiquement. Les actions correspondant à un bloc d'entrée de l'application sont périodiques. En revanche, les actions suivantes ne le sont pas forcément (voir analyse de l'ECC). Toutefois, afin de pouvoir faire une analyse *a priori*, nous "majorons" leur activation par une activation périodique. Sous cette hypothèse, en particulier, nous ne pourrons donner qu'une condition suffisante d'ordonnançabilité.

Enfin, nous introduisons les notations suivantes :

Notations.
- Σ : désigne l'ensemble de toutes les actions des Blocs Fonctionnels de l'application.
- $first(\sigma)$ (resp. $last(\sigma)$) : soit σ, ($\sigma \subset \Sigma$), $first(\sigma)$ (resp. $last(\sigma)$) désigne l'ensemble des actions sans prédécesseur (resp. successeurs) dans σ.
- act_i^j la j^{me} instance de l'action act_i ($act_i \in \Sigma$).

Exemple. *Dans l'exemple proposé ci-dessus, neuf actions ont été identifiées dans* σ ($\sigma = \{act_1, ...act_9\}$. *Chaque action* act_i *(*$i \in [1, 9]$*) correspond à l'événement d'entrée* ie_i.

Les ensembles $first(\sigma)$ et $last(\sigma)$ contiennent les actions suivantes,

$$first(\sigma) = \{act_1, act_5\}$$

$$last(\sigma) = \{act_7, act_8, act_9\}$$

Les prédécesseurs de l'action act_8 *sont* act_3 *et* act_4.

$$pred(act_8) = \{act_3, act_4\}$$

Les prédécesseurs de l'action act_9 *sont* act_2 *et* act_6.

$$pred(act_9) = \{act_2, act_6\}$$

Nous caractérisons les successeurs de act_1 *de la façon suivante,*

$$succ(act_1) = \{\{act_2, act_3\}, \{act_4\}\}$$

De même, lorsque l'action act_5 *finit son exécution, nous devons à exécuter les actions* act_6 *et* act_7,

$$succ(act_5) = \{\{act_6, act_7\}\}$$

D'autre part, nous considérons que pour chaque action act de σ,
- $BCET(act) = 5$
- $WCET(act) = 8$

L'occurrence des événements ie_1 *et* ie_5 *est périodique ; aussi, l'activation des actions* act_1 *et* act_5 *est périodique.*

Enfin, nous introduisons les caractérisations suivantes d'actions :

Action principale

Définition 2 : une action $act \in \sigma$ est principale si elle est exécutée toutes les fois que ses prédécesseurs sont exécutés.

Exemple. *Dans l'exemple, les actions* act_1, act_5, act_6 *et* act_7 *sont principales. Les actions* act_2, act_3 *et* act_4 *sont non principales. En effet, leur activation dépend du résultat d'exécution de* act_1.

Actions disjointes

Definition 3. dans une ressource, nous supposons que deux actions act_i et act_j ($\{act_i, act_j\} \subseteq first(\sigma)$)sont disjointes si leur exécution est disjointe (une des deux est à exécuter au plus).

4.3.2 Trace

Définition 4. Les contraintes de précédence, spécifiées par le réseau de Blocs Fonctionnels regroupés dans une ressource, définissent les contraintes de précédence entre les actions correspondantes. Nous définissons une "trace d'actions" pour un ensemble d'actions σ ($\sigma \subset \Sigma$), la séquence suivante :

$$tr = act_1, act_2, ..., act_n \text{ telle que}$$

- $act_1 \in first(\sigma)$.
- $act_n \in last(\sigma)$.
- $\forall\ act_i$ tel que $i \in]1, n]$, $act_{i-1} = pred(act_i)$.

Dans la ressource, la trace tr représente une chaîne de causalité entre actions appartenant à σ.

Exemple. *Dans l'exemple considéré ci-dessus, nous distinguons cinq traces,*

- $tr_1 = act_1, act_4, act_8$
- $tr_2 = act_1, act_2, act_9$
- $tr_3 = act_1, act_3, act_8$
- $tr_4 = act_5, act_6, act_9$
- $tr_5 = act_5, act_7$

Notation. nous dénotons dans la suite "$traces(\sigma)$" l'ensemble des traces d'actions dans la ressource de σ.

$$traces(\sigma) = \{tr / \forall act \in tr, act \in \sigma\}$$

Exemple. *Dans l'exemple ci-dessus, l'ensemble $traces(\sigma)$ contient les traces suivantes :*

$$traces(\sigma) = \{tr_1, tr_2, tr_3, tr_4, tr_5\}$$

De plus, nous caractérisons une trace tr de $traces(\sigma)$ à l'aide de sa première action $first\_action(tr)$ et de sa dernière action $last\_action(tr)$.

En considérant les contraintes temporelles du cahier des charges [Cucu and Sorel, 2004], nous associons à une trace tr une borne sur le temps de réponse entre l'activation de $first\_action(tr))$ et la réponse correspondant à la fin d'exécution de $last\_action(tr)$.

Nous précisons que dans la majorité des cas industriels, si $first\_action(tr))$ correspond à une lecture de données sur un capteur, chaque lecture n'est possible que si le traitement de la lecture précédente est complètement fini.

De ce fait, nous supposons dans tout ce manuscrit que les traces d'actions traitées dans une ressource sont non ré-entrantes [Klein *et al.*, 1993; Liu, 2000].

Par définition, une trace tr est non-réentrante si et seulement si : l'exécution de la j^{eme} instance de $first\_action(tr)$ ne peut démarrer qu'aprés la fin d'exécution de la $(j-1)^{eme}$ instance de $last\_action(tr)$.

4.3.3 Opération

Pour analyser le comportement d'un ensemble d'actions σ, dans une ressource nous devons étudier les scénarios d'exécution possibles à chaque activation d'une action act_i de $first(\sigma)$ ($act_i \in first(\sigma)$).

Définition 5. Nous définissons une opération op_i comme étant l'ensemble de toutes les traces ayant act_i comme première action.

$$op_i = \{tr \in traces(\sigma) \ / \ act_i \in first(\sigma), act_i = first\_action(tr)\}$$

En considérant cette caractérisation, le comportement d'une application dans une ressource est spécifié par l'ensemble de ses différentes opérations.

Exemple. *Dans l'exemple, nous distinguons deux opérations op_1 et op_5 correspondant respectivement à act_1 et act_5.*

$$op_1 = \{tr_1, tr_2, tr_3\}$$

$$op_5 = \{tr_4, tr_5\}$$

Notation. Dans toute la suite, nous dénotons par $oper(\sigma)$ l'ensemble de toutes les opérations de σ.

$$oper(\sigma) = \{op_i / act_i \in \sigma\}$$

Nous imposons, enfin, que deux opérations de σ ne contiennent pas la même action. Si une action appartient à plus qu'une opération, alors elle est dupliquée dans notre modèle. De plus, nous imposons que les opérations traitées soient des arbres. S'il existe dans une opération deux chemins entre la première action et une action act_i, alors elle est dupliquée.

Ce choix permet la simplification de la caractérisation temporelle des actions traitées. Ainsi, l'activation d'une action ne dépend que de l'arrivée d'un et un seul événement d'entrée de σ.

Exemple. *L'action act_9 appartient à deux opérations différentes (i.e. op_1 et op_5). Nous proposons de la dupliquer en deux actions (act_{91} et act_{92}) pour simplifier sa caractérisation temporelle.*

D'autre part, la structure de l'opération op_1 n'est pas un arbre. En effet, l'action act_8 a deux prédécesseurs act_3 et act_4. Nous proposons de la dupliquer en deux actions (act_{81} et act_{82}) pour simplifier l'analyse d'ordonnançabilité par la suite.

4.4 Contraintes temporelles

Le problème de perte d'événements peut être un problème critique dans une application de contrôle. En effet, la perte de certaines données importantes cause toujours le fonctionnement incorrect de certains procédés physiques.

D'autre part, les bornes sur les temps de réponse représentent des contraintes critiques que doit respecter l'application.

Afin de garantir un comportement correct d'un réseau de Blocs Fonctionnels dans une ressource, nous proposons, à partir d'une contrainte globale sur le temps de réponse de bout en bout, d'allouer des échéances d'exécution sur chacun des Blocs Fonctionnels du réseau considéré [Khalgui *et al.*, 2005b].

Chaque échéance sera calculée pour,

- Garantir le respect des bornes sur des temps de réponses de bout en bout ;
- Garantir le traitement de toutes les occurrences d'événements d'entrée du réseau.

En particulier, nous avons développé une méthode calculant des échéances aux différentes actions de σ. Cette méthode évalue pour chaque action act l'échéance,

- d^{borne} pour respecter des bornes sur les temps de réponse de bout en bout.
- d^{perte} pour éviter tout problème de saturation au niveau du bloc qui serait dû au fait que des occurrences d'événements se produiraient alors que l'ECC n'est pas libre et que le buffer de mémorisation associé au ports (voir chapitre 3) est saturé.

Pour garantir un comportement temporel correct de l'application, chaque action $act_i \in \sigma$ doit finir son exécution avant ses deux échéances d_i^{borne} et d_i^{perte}.

4.4.1 Allocation d'échéances aux actions pour respecter les bornes imposées sur les temps de réponse de bout en bout

Dans cette section, nous supposons que les bornes sur les temps de réponse de bout en bout sont issues du cahier des charges de l'application. Elles définissent un temps maximum entre l'arrivée d'un événement *ie* et la réponse de l'application sous forme d'un événement de sortie *oe*. Il y a donc dans l'application une chaîne de causalité entre *ie* et *oe*.

Dans notre modèle, cela se traduit par l'existence d'une trace entre l'action activée par l'arrivée de *ie* et l'action produisant l'occurrence correspondante de l'événement *oe*. Pour simplifier la lecture, nous notons ces actions respectivement act_1 et act_n, et tr cette trace d'actions de σ.

$$tr = act_1, \ldots, act_n$$

Nous définissons $borne(tr)$ comme étant la borne sur le temps de réponse entre l'activation de act_1 et la fin d'exécution de act_n.

Pour respecter une telle borne, nous proposons de calculer pour chaque action $act_i, i \in [1, n]$, de tr une échéance notée d_i^{borne} [Khalgui *et al.*, 2005a].

Considérons une action act_i. Il faut calculer son échéance de sorte que toutes les traces auxquelles elle appartient respectent leurs bornes de temps de réponse. Cela revient donc à intégrer dans l'échéance le pire temps d'exécution de tous ses successeurs.

L'algorithme proposé commence par affecter une échéance à la dernière action de chaque trace. Ensuite, on prend l'ensemble des prédécesseurs dans les traces et on leur affecte une échéance tenant compte du temps d'exécution des derniers. On ré-applique ainsi de suite la technique jusqu'à arriver aux actions appartenant à $first(\sigma)$.

Nous donnons, ci-dessous, la technique récursive calculant les échéances d_i^{borne} de l'action act_i de la trace tr,

- **Si** $i = n$, **alors**,

$$d_i^{borne} = borne(tr)$$

- **Sinon**,

$$d_i^{borne} = min_{s \in succ(act_i)}(min_{act_j \in s}(d_j^{borne} - \sum_{act_k \in s, d_k^{borne} \leq d_j^{borne}} WCET(act_k)))$$

Exemple. *Dans l'exemple, nous appliquons la technique proposée pour calculer les échéances d_i^{borne} des différentes actions de σ.*

Nous illustrons dans la figure 4.2 le calcul de l'échéance d_1^{borne} de l'action act_1. Cette échéance est égale à la date minimale permettant aux successeurs de respecter leurs échéances.

$$d_1^{borne} = min\{d_3^{borne} - WCET(act_3) - WCET(act_2), d_4^{borne} - WCET(act_4)\} = 76$$

4.4.2 Allocation d'échéances aux actions pour éviter les pertes d'occurrences d'événements

Soit bf un Bloc fonctionnel dans une ressource. Notons par m la taille du buffer d'occurrences d'événement en attente de l'ECC (voir chapitre 3). Le problème de perte d'événements apparaît lorsque le nombre d'occurrences en attente de l'ECC est supérieur à m.

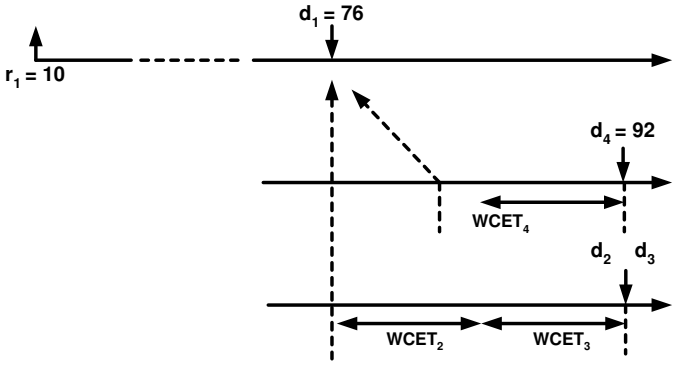

FIG. 4.2 – Le calcul de d_1^{borne}

Nous proposons d'allouer une échéance de fin d'exécution à chaque action de manière à ce que le respect de cette échéance par toute instance d'une telle action garantisse qu'à tout instant, il n'y a pas plus d'occurrences d'événements en attente de traitement que de place dans le buffer correspondant d'entrée des Blocs Fonctionnels [Khalgui *et al.*, 2005b].

Nous précisons que cette méthode peut être appliquée de manière constructive dans la spécification d'un réseau de Blocs Fonctionnels. En fait, dans ce cas, les mêmes principes nous permettent de déduire la taille du buffer qui assurera un fonctionnement sans perte d'occurrence et ce à partir de la caractérisation temporelle proposée. En effet, une fois l'ordonnancement construit (voir plus loin), il est possible d'évaluer le nombre maximum d'événements en attente pour fixer la taille du buffer.

Calcul des dates d'activation

Rappelons que nous considérons uniquement des événements périodiques en entrée. Toutefois, comme les calculs dans un bloc peuvent influer sur les événements en sortie, les actions suivantes risquent de ne pas être activées périodiquement. Aussi, ainsi que nous l'avons déjà dit, afin de réaliser l'analyse d'ordonnançabilité, nous considérons que les activations sont périodiques. Cela revient à considérer qu'à chaque instance d'événement d'entrée correspond des instances de tous les événements de sortie (Cela malgré le comportement conditionnel à l'intérieur d'un bloc). Malheureusement, cette contrainte rend notre condition d'ordonnançabilité plus pessimiste.

Rappelons également qu'une action appartient à une et une seule opération (sinon, elle est dupliquée).

Soit tr une trace de $traces(\sigma)$, avec act_1 et act_n ayant la même signification que précédemment :

$$tr = act_1, ..., act_n$$

Nous caractérisons chaque action act_i, $i \in [2, n]$, de tr de la façon suivante [Khalgui *et al.*, 2006c] :

– r_i : La date d'activation au plus tôt est égale à la date au plus tôt de fin d'exécution de tous ses prédécesseurs.

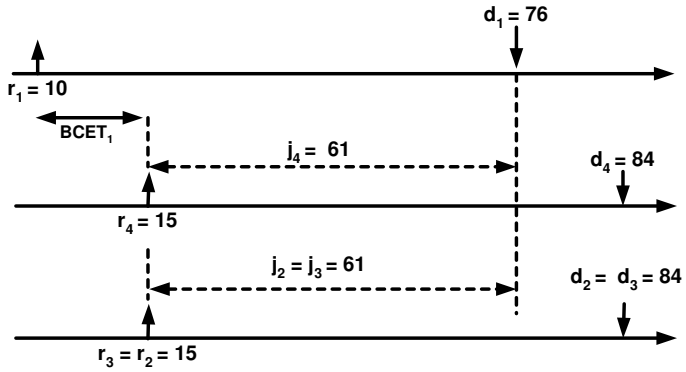

FIG. 4.3 – Les calculs des dates d'activation de act_2, act_3 et act_4

$$r_i = r_0 + \sum_{k=1}^{k=i-1} BCET(act_k)$$

– j_i : La gigue est la différence entre la date d'activation au plus tard et celle au plus tôt. La date d'activation au plus tard est la plus tardive échéance de ses prédécesseurs.

$$j_i = d_{i-1} - max_{act_j \in pred(act_i)}\{r_j\}$$

– p_i : La période est celle de act_1. En effet, les traces considérées sont non-réentrantes. De ce fait, toutes les actions ont la même période.

Exemple. *Sur l'exemple proposé précédemment, nous caractérisons les actions act_2, act_3 et act_4 avec ces paramètres (figure 4.3).*

En particulier, la date d'activation au plus tôt des ces actions est lorsque act_1 finit son exécution au plus tôt. La date d'activation au plus tard est lorsque act_1 finit à son échéance. La différence des deux dates est égale à la gigue des trois actions.

Évaluation de l'hyper période

Pour valider le comportement temporel d'un réseau de Blocs Fonctionnels dans une ressource, nous avons à vérifier le respect des échéances de toutes les instances de toutes les actions dans une hyper-période hp [Khalgui *et al.*, 2006c]. De ce fait, nous proposons de calculer ces échéances dans cet intervalle.

Soit lcm le plus petit commun multiple (ppcm) de toutes les périodes des actions de $first(\sigma)$. De plus, soient act_{min} et act_{max} deux actions de $first(\sigma)$ telles que,

$\forall\ act_i \in \sigma\ ;\ r_{min} + j_{min} \leq r_i + j_i \leq r_{max} + j_{max}$

Comme nous traitons des traces non-réentrantes, nous utilisons la formule donnée dans [Leung and Whitehead, 1982] pour évaluer l'hyper-période :

$$hp = [r_{min} + j_{min}\ ;\ r_{max} + j_{max} + 2.lcm].$$

Exemple. *L'hyper-période de l'exemple proposé est* $hp = [11, 221]$.

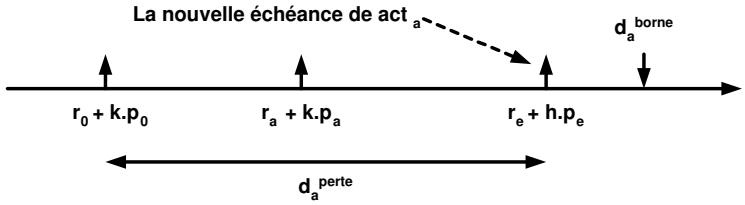

FIG. 4.4 – Le calcul de d_i^{perte}

Technique de calcul des échéances

Le but de cette technique est de calculer une échéance d_i^{perte} à chaque instance $act_i \in \sigma$. Ces échéances évitent toute saturation de buffers dans les Blocs Fonctionnels.

Soit act_a une action d'un Bloc Fonctionnel bf. Cette action appartient aussi à une trace tr de $trace(\sigma)$,

$$tr = act_1, \ldots, act_n$$

où act_1 et act_n désignent la première et dernière action de la trace tr.

Soit act_i^k l'instance k de act_i dans l'hyper-période. Soit de plus act_e^h l'instance de act_e tel que act_e^h est la $(m+1)^{eme}$ instance activée dans bf aprés act_i^k.

Pour éviter tout problème de perte d'événements dans le buffer de bf, l'exécution de l'instance act_i^k doit nécessairement finir avant l'activation au plus tôt de act_e^h. Dans le cas contraire, cela signifie qu'on se trouve dans la situation où $m+1$ occurrences d'événements sont en attente de l'ECC (figure 4.4).

Notons par $d_i^{perte,k}$ l'échéance sur l'exécution de l'instance act_i^k qui garantisse d'éviter tout problème de perte d'événements. Cette échéance est à calculer de la façon suivante :

$$d_i^{perte,k} = r_e + h.p_e - (r_0 + k.p_0)$$

En considèrant $(\lfloor (r_{max} + 2.lcm - r_i)/p_i \rfloor + 1)$ instances de chaque action act_i de bf dans l'hyper-période, nous calculons l'échéance correspondante sur toute instance de act_i pour éviter toute perte d'événements,

$$d_i^{perte} = min\{d_i^{perte,k}, k \in [0, \lfloor (r_{max} + 2.lcm - r_i)/p_i \rfloor]\}$$

Pour éviter tout problème de perte d'événements dans le Bloc bf, l'échéance de toute action doit respecter la condition suivante :

$$\forall\ act_i \in bf,\ d_i \leq d_i^{perte}$$

Finalement, pour garantir un comportement correct de l'application, nous devons appliquer cette technique pour tout Bloc Fonctionnel la constituant.

Exemple. *Dans l'exemple, nous supposons que la taille du buffer de* FB_2 *est* $m = 1$.

Nous calculons l'échéance d_2^{perte} *en appliquant la technique proposée. Pour ne pas avoir un problème de perte d'événements dans* FB_2, $d_2^{perte} = 105$ *(figure 4.5). Rappelons que, pour respecter la borne sur le temps de réponse correspondante,* $d_2^{borne} = 84$.

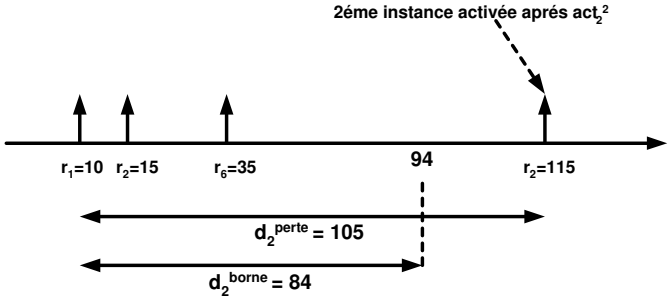

FIG. 4.5 – Le calcul de d_2^{perte} de FB_2

4.4.3 Méthode générale de calcul des échéances

Nous proposons dans cette partie une méthode calculant les échéances de toutes les actions de l'ensemble σ [Khalgui *et al.*, 2006c]. Cette méthode doit prendre en compte le calcul des échéances suivantes, pour toute action act_i :

- d_i^{borne} pour respecter des bornes sur le temps de réponse de bout en bout ;
- d_i^{perte} pour éviter tout problème de perte d'événements dans les Blocs Fonctionnels.

Contrairement à d_i^{perte}, le calcul de d_i^{borne} repose sur les échéances des successeurs de l'action. De ce fait, ces échéances doivent prendre en compte les échéances de perte d_i^{perte}.

La méthode que nous proposons décline deux étapes pour calculer l'échéance de toute action act_i de σ.

Soit tr une trace de $trace(\sigma)$, act_1 et act_n, sa première et dernière actions.

$$tr = act_1, \ldots, act_n$$

Première étape.

En se basant sur la technique calculant les échéances pour éviter toute perte d'événements, nous calculons d_i^{perte} pour chaque action act_i de la trace tr.

Deuxième étape.

Nous calculons pour chaque action act_i, de la trace tr, l'échéance réelle de la façon suivante.

- **Si** i = n, **Alors**,

$$d_n = \min\{d_n^{perte}, borne(tr)\}$$

- **Sinon**,

$$d_i = \min\{d_i^{perte}, d_i^{borne}\}$$

Une fois la méthode appliquée pour le calcul de l'échéance de chaque action, on peut savoir si l'application est non faisable, sous les hypothèses que nous avons imposées.

Proposition. Soit une application de contrôle σ localisée dans une ressource donnée d'un dispositif. S'il existe une action act_i ($act_i \in \sigma$) telle que :

$$d_i < WCET(act_i)$$

alors l'application n'est pas faisable.

Remarque : ceci ne concerne que l'étude d'une trace tr de σ ; dans le cas où $d_i \geq WCET(act_i)$ pour toute action act_i de tr, il reste néanmoins à vérifier l'ordonnançabilité de l'ensemble des traces de σ. C'est ce que nous présentons dans ce qui suit.

4.5 Analyse d'ordonnançabilité

Une fois l'application correctement transformée vers un système d'actions sous contraintes de précédence, nous appliquons une analyse d'ordonnançabilité dans l'ensemble des traces de σ [Khalgui *et al.*, 2005a].

Sous les hypothèses de périodicité des actions données précédemment, σ est donc une configuration d'actions (ou tâches) périodiques sous contraintes de temps (échéances) et de précédence. À notre connaissance, une seule contribution a été proposée pour les systèmes à base de tâches périodiques, sous contraintes de précédence et devant respecter des bornes sur les temps de réponse de bout en bout [Cucu and Sorel, 2005]. Néanmoins, le modèle de tâches proposé dans ce travail n'est pas suffisamment expressif pour modéliser tous les comportements possibles d'un réseau de Blocs Fonctionnels. En effet, le comportement conditionnel d'un bloc (envois conditionnels des événements de sortie) décrit précédemment n'est pas modélisable par le modèle de tâches décrit dans [Cucu and Sorel, 2005].

Nous proposons de réaliser cette analyse d'ordonnançabilité en appliquant, hors ligne, une politique EDF non-préemptive pour vérifier toutes les bornes sur les temps de réponse. Cette politique est optimale pour les tâches indépendantes. Or, comme nous avons tenu compte une première fois des dépendances entre actions pour le calcul des dates d'activation et d'échéances, notre problème se ramène bien à ordonnancer des groupes de tâches indépendantes. Les dépendances du problème initial ne nous servent plus qu'à définir l'ensemble des actions à exécuter ensemble.

Le résultat de ce calcul sera donc un ensemble d'ordonnancements statiques et oisifs qui seront utilisés à l'exécution par un séquenceur.

Dans certaines applications industrielles *critiques*, le respect de toutes les contraintes temporelles est une exigence à satisfaire. L'analyse de l'ordonnançabilité doit être **stricte**. Dans d'autres cas d'applications, le non respect de certaines échéances (suivant un profil donné [Hamdaoui and Ramanathan, 1995]) peut être accepté. Dans ce cas, on peut s'autoriser à *rejeter* des instances d'actions ce qui peut correspondre à une qualité dégradée et conduire à un mode dégradé de l'application qui reste acceptable sous un profil de rejets spécifié dans le cahier des charges. L'analyse d'ordonnançabilité doit, alors, intégrer de tels modes **dégradés** dans l'exécution de l'application.

Dans un souci de simplicité, nous parlerons dans le premier cas d'analyse **stricte** et dans le deuxième d'analyse **pour le mode dégradé**. Ainsi que introduit, plus haut, nous proposons de les réaliser en appliquant la politique EDF non-préemptive [Stankovic *et al.*, 1998].

Ces analyses se basent sur la construction d'un graphe d'accessibilité [Grolleau and Choquet-Geniet, 2002] sur une hyper-période fixée [Leung and Whitehead, 1982].

Le graphe d'accessibilité est un ensemble de trajectoires. Chaque trajectoire représente un ordonnancement possible d'un ensemble de traces du système d'actions. Elle représente, en fait, un comportement possible de l'application. Nous appliquons hors ligne la politique EDF non-

préemptive, durant la construction de chaque trajectoire, pour vérifier les contraintes temporelles des différentes actions de ces traces.

Soit G le graphe d'accessibilité de l'application dans l'hyper-période correspondante ($hp = [r_{min} + j_{min}; r_{max} + j_{max} + 2.lcm]$.

Chaque **noeud** de ce graphe contient une classe d'actions.

Définition. Une classe d'actions C du graphe G est définie de la façon suivante.

$$C = \{set, act, t\} \text{ avec,}$$

- set : un ensemble d'instances d'actions à exécuter.
- act : une instance d'action à exécuter parmi toutes les instances actives de set. Cette action est sélectionnée en appliquant la politique EDF non-préemptive.
- t : la date de début d'exécution de l'instance act.

Chaque **branche** de ce graphe d'accessibilité représente un ordonnancement correspondant à une exécution possible. En effet, il suffit de prendre dans chaque classe, l'action sélectionnée ainsi que sa date d'activation.

Le résultat de l'analyse est un ensemble d'ordonnancements **statiques** et **oisifs** utilisés à l'exécution par un séquenceur (selon le standard IEC 61499, ce séquenceur implémente la fonction d'ordonnancement de la ressource des Blocs Fonctionnels).

4.5.1 Ordonnançabilité stricte

Dans cette partie, nous nous intéressons à l'analyse stricte de l'ordonnançabilité. Lors de la construction du graphe d'accessibilité, on s'arrête dès la violation d'une échéance : l'application est alors non faisable.

Nous présentons d'abord l'ensemble des règles de construction du graphe puis nous présentons un algorithme réalisant la construction selon ces règles [Khalgui *et al.*, 2005a].

La génération du graphe d'accessibilité

Pour des raisons de clareté, nous définissons la fonction $echeance\_abs$ calculant l'échéance absolue d'une instance d'action. Nous rappelons que l'échéance calculée dans la section précédente est relative à la date d'activation de la trace correspondante.

$$echeance\_abs(act_i^h) \;=\; d_i + r_k + p_k * h$$
$$\text{ou } act_k = first\_action(tr) \text{ avec } tr \in traces(\sigma)/act_i \in tr$$

Nous présentons maintenant les différentes règles construisant le graphe d'accessibilité G.

* **Règle 0. Initialisation.** Nous construisons la première classe d'actions C_0 de G de la façon suivante.

$$C_0 = \{set_0, act_i^0, r_i + j_i\}$$

avec
- $set_0 = \{act_j^0 / act_j \in first(\sigma)\}$.
- $act_i^0 \in set_0 \;/\; \forall act_j^0 \in set_0,\; i \neq j \; et \; echeance\_abs(act_i^0) < echeance\_abs(act_j^0)$.

Une fois la première classe d'actions est correctement construite, nous appliquons récursivement les règles suivantes pour construire le reste du graphe G.

Soit $C_i = \{set_i, act_i^q, t_i\}$ une classe d'actions de G n'ayant pas encore de successeurs.

* **Règle 1. Condition d'arrêt avec succès.** Si nous atteignons la borne supérieur de l'hyper-période hp.

$$t_i \geq r_{max} + j_{max} + 2.lcm$$

Alors, nous arrêtons la construction de la trajectoire courante. Cette trajectoire correspond à un ordonnancement possible de l'application.

* **Règle 2. Condition d'arrêt avec échec.** Si une instance de set_i ne respecte pas son échéance.

$$\exists\ act_j^q \in set_i\ /\ echeance\_abs(act_j^q) < t_j + WCET_j$$

Alors, le système est non faisable. Nous arrêtons la construction de G et donc de l'analyse d'ordonnançabilité de l'application.

* **Règle 3.** Cette construction est réalisée récursivement. Si l'action sélectionnée act n'appartient pas à l'ensemble $last(\sigma)$ (act n'est pas le dernier de sa trace), alors nous construisons des successeurs. Supposons que,

$$succ(act_i) = \{ts_1, ..., ts_k\}$$

Dans ce cas, pour chaque $h \in [1, k]$, nous construisons une classe $C_j = \{set_j, act_j^q, t_j\}$ tel que :

** $set_j = (set_i \backslash \{act_i^q\}) \cup ts_h$
** $t_j = t_i + WCET(act_i)$
** $act_j^q \in set_j\ /\ \forall act_l^m \in set_j,\ l \neq j\ et\ echeance\_abs(act_j^q) < echeance\_abs(act_l^m)$.

Deux cas peuvent alors se présenter :

- **Si** C_j existe déjà dans G,**alors** C_j est un successeur de C_i dans G,
- **Sinon**, nous ajoutons la classe C_j à G.

* **Règle 4. Action finale** Si l'action act_i appartient à l'ensemble $last(\sigma)$ (elle est la dernière de sa trace), alors elle ne possède pas des successeurs.

Supposons que op_p l'opération contenant l'action act_i.

- **Si** $\exists C_j \in G$ tel que,

  ** $set_j = set_i \backslash \{act_i^q\} \cup \{act_p^{q+1}\}$

  ** $t_j = t_i + WCET(act_i)$

  alors C_j est un successeur de C_i dans G,
- **Sinon**, nous construisons une classe $C_j = \{set_j, act_j^h, t_j\}$ avec,

  ** $set_j = (set_i \setminus \{act_i\}) \cup act_p^{q+1}$.

  ** $t_j = t_i + WCET(act_i)$.

  ** $act_j^h \in set_j\ /\ \forall act_l^m \in set_j,\ l \neq j\ et\ echeance\_abs(act_j^q) < echeance\_abs(act_l^m)$.

Algorithme

Nous proposons l'algorithme implémentant l'analyse d'ordonnançabilité stricte [Khalgui *et al.*, 2005a]. Cet algorithme construit le graphe d'accessibilité en se basant sur une fonction récursive *generate*(*table*). La faisabilité de l'application correspond à une génération complète de toutes ses trajectoires dans l'hyper-période *hp*.

```
Bool generate(G : tasks_state, C : tasks_state,
first : tasks_list, tasks : tasks_list, time : integer)
Begin
T1 : task ; C1 : tasks_state ; result : bool ;
result ← true ;
if(C.t ≥ time)
time = 2.lcm + rmax + jmax
  then return(true) ;
for each task T1 ∈ C.S
   if deadline _ violated (T1)
      then return (false) ;
C.T ← apply _ EDF(C.S) ;
while(ts ∈ C.T → succ and result)
  if(not exist(C.S \ C.T ∪ ts, G)      create(C1) ; C1 .S ← C.S \ C.T ∪ ts ;
    C1.t ← C.t+ T.WCET ;
    result ← generate(G, C1 , first, time)
return result ;
End.
```

Par construction complète du graphe d'accessibilité G, cette analyse d'ordonnançabilité est optimale (en considérant les instructions de la norme sur la réalisation d'un ordonnancement non-préemptive dans une ressource). En effet, si une instance d'action ne respecte pas son échéance, alors il n'y a aucune autre analyse d'ordonnancement faisant respecter cette contrainte sous les hypothèses que nous avons utilisées.

Exemple. *Dans l'exemple traité dans ce chapitre, nous obtenons le graphe d'accessibilité dans l'hyper-période* $hp = [11, 231]$ *(figure 4.6). En vérifiant que toutes les échéances sont respectées, nous concluons à la faisabilité de l'application.*

Complexité

Nous analysons la complexité de l'algorithme proposé. Nous dénotons par m le nombre de toutes les opérations de σ à ordonnancer dans une ressource d'un dispositif donné. Soit p_i le nombre de traces de l'opération op_i ($i \in [1, m]$). De plus, soit q_i le nombre d'actions de la plus longue trace de op_i.

Nous dénotons par,

- $\alpha = \Pi_{j=1...m} p_j$: le nombre des trajectoires dans le graphe d'accessibilité.
- $\beta = \Sigma_{j=1}^{m} q_j$: la plus longue trajectoire du graphe.

Le nombre maximal de classes d'actions à construire dans le graphe G est $\alpha * \beta$. Il est évident que la complexité est alors en $O(\alpha * \beta)$.

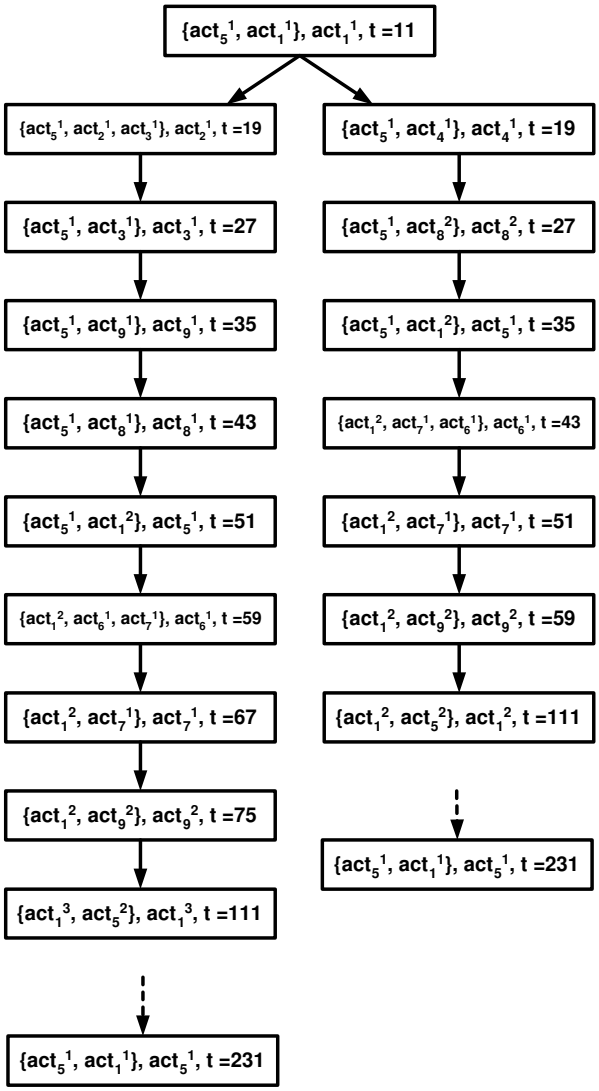

FIG. 4.6 – Graphe d'accessibilité de l'application donnée en exemple

4.5.2 Ordonnançabilité dégradée

Dans le cas où l'application n'est pas ordonnançable, il est parfois possible de supprimer (sous réserve d'un comportement acceptable) certaines traces et ainsi fonctionner en mode dégradé [Khalgui *et al.*, 2006d].

Une opération op_i implémente tous les scénarios d'exécution possibles lorsqu'une action act_i est activée. Ces scénarios ne sont pas prévisibles hors-ligne mais dépendent du contexte en-ligne. De plus, l'opération op_i doit respecter toutes les bornes sur les temps de réponse de bout en bout sur chacune de ses traces pour garantir une correction comportementale de l'application. Dans cette partie, nous considérons donc qu'un fonctionnement dégradé mais qui reste acceptable peut être obtenu en rejetant des instances spécifiques d'opérations.

Nous proposons d'exprimer des "contraintes de dégradation acceptable" sur chaque opération de l'application. Pour ceci, nous exploitons le modèle (m,k)-firm pour définir le degré de dégradation admissible [Hamdaoui and Ramanathan, 1995; 1994]. Dans un modèle de tâches temps-réel, une tâche est faisable sous une contraintes (m,k)-firm si, toujours, m parmi k instances successives respectent leurs échéances.

Dans notre cas, on dira qu'une instance op_i^j d'une opération op_i ($op_i \in oper(\sigma)$) respecte ses échéances si toutes ses traces sont ordonnançables. L'application de contrôle est alors faisable si pour chaque opération op_i ($op_i \in oper(\sigma)$), m_i parmi k_i instances respectent leurs échéances.

Nous proposons alors de généraliser la méthode proposée dans la section précédente. L'analyse construit un graphe d'accessibilité G vérifiant toutes les contraintes temporelles en tenant en compte leurs contraintes de dégradation acceptable.

De façon similaire à ce qui est fait pour une analyse stricte, la construction du graphe d'accessibilité repose sur la politique EDF non-preemptive [Stankovic *et al.*, 1998]. Si l'instance d'action $act_i^j \in op_h^j$ ne respecte pas son échéance, alors nous évaluons dans l'historique de op_h le nombre de ses instances n'ayant pas respecté leurs échéances.

- Si la contrainte (m_h, k_h) n'est pas violée, alors nous enlevons toute l'instance op_h^j du graphe d'accessibilité G.
- Sinon, nous concluons que l'application n'est pas ordonnançable en mode dégradé.

Construction du graphe d'accessibilité

Nous définissons la fonction qui compte le nombre d'instances réussites d'une opération sur les k_i dernières instances [Khalgui *et al.*, 2006d].

$$success(op_i^j, G) = cardinality(\{C \in G / C = \{set, act_i^l, t\} et j - k \leq l \leq j\})$$

Par rapport à l'ordonnancement strict, seule la **Règle 2** diffère. Par soucis de concision, nous ne présentons que celle-ci. Nous considérons de nouveau une classe $C_i = \{set_i, act_i^q, t_i\}$ du graphe G n'ayant pas encore de successeurs.

Règle 2. Si une instance de set_i ne respecte pas son échéance,

$$\exists act_j^h \in set_i \text{ tel que } echeance\_abs(act_j^h) < t_i + WCET(act_j)$$

Dans ce cas, l'instance d'opération correspondante ne peut être ordonnancée. Soit op_q^h cette instance.

$$\exists op_q^h / act_j^h \in op_q^h$$

* Si la contrainte (m, k) est violée,

$$success(op_q^h, G) \leq m_q$$

Alors l'application est non ordonnançable même en mode dégradé.

* Sinon, (i.e. la contrainte n'est pas violée)

** On passe à l'instance de l'opération suivante,

$$\forall \{set, act_q^h, t\} \in G,$$

$$set = set \setminus \{act_q^h\} \cup \{act_q^{h+1}\}$$

** On supprime l'instance de l'opération courante,

$$G = G \setminus \{\{set, act, t\} \in G / set \cap op_q^h = \varnothing\}$$

Algorithme

L'algorithme de l'analyse d'ordonnançabilité sous contraintes de dégradation acceptable repose sur une fonction récursive *generate*() à appliquer pour construire le graphe d'accessibilité. Les fonctions suivantes sont utilisées dans la fonction *generate*() [Khalgui *et al.*, 2006d].

- $Source(act_m^n)$: définit pour une instance act_m^n de op_h^n la classe $C_k = \{set_k, act_h^n, t_k\}$ du graphe G.
- $free(C)$: libère toutes les classes construites à partir de C.

```
Bool generate(C : tasks_ state, first : tasks_ list, oper :
operation_ list, time : integer, L : instances_ list)
Begin
T1 : task ; op : operation ; L1 : tasks_list ; C1 : tasks_state ;
if (C.t ≥ time)
time = 2.lcm + r_max + j_max
   then return(true) ;
for each task T1 ∈ C.S
   if deadline_violated (T1)
      then op ← get_operation(T1, oper) ;
       if (m_k_violated(op))
          then L1 ← NULL ; return false ;
       else add_instance(T1, L) ;
if (L ≠ NULL) then return false ;
C.T ← apply_EDF(C.S) ;
while (ts ∈ C.T → succ)
  if(notexist(C.S \ C.T ∪ ts, G)      create(C1) ; C1 .S ← C.S \ C.T ∪ ts ;
    C1.t ← C.t+ T.WCET ; L1 ← NULL;
   if (not generate( C1, first, operations, time, L1))
       then L ← L ∪ L1 ;        for each Tk in L and source(Tk =C
       ;
       if (Tk)
       then T1 ← root_op(Tk, C.S) ;

T1 = root(op) ; op = op(Tk) ;
          T1.r ← T1.r + T1.p ; free( C ) ;
       L ← L \ {T1} ;
          return(generate(C, first, operations,time,L)) ;
       else return false ;
return true ;
End.
```

Exemple. *Supposons que les caractéristiques de ie_1 et ie_5 soient ici :*
- $r_1 = 10, j_1 = 1, p = 100$
- $r_5 = 100, j_5 = 1, p = 200$

Et que les contraintes de dégradation acceptables pour les opérations op_1 et op_5 soient :
- $(m_1, k_1) = (1, 2)$
- $(m_5, k_5) = (1, 1)$

pour le respect des bornes suivantes sur les temps de réponses de l'application,

$$borne(ie_1, oe_8) = borne(ie_1, oe_9) = 40$$

$$borne(ie_5, oe_8) = borne(ie_5, oe_9) = borne(ie_5, oe_{10}) = 40$$

Lors de l'analyse, il ressort que la deuxième instance op_1^2 de l'opération op_1 ne respecte pas ses contraintes d'échéance. Cependant, la garantie demandée sur op_1 ((m_1, k_1)) permet de rejeter cette instance ; la construction du graphe dans l'hyper-période ($hp = [11, 501]$) se poursuit donc en supprimant cette instance (figure 4.7).

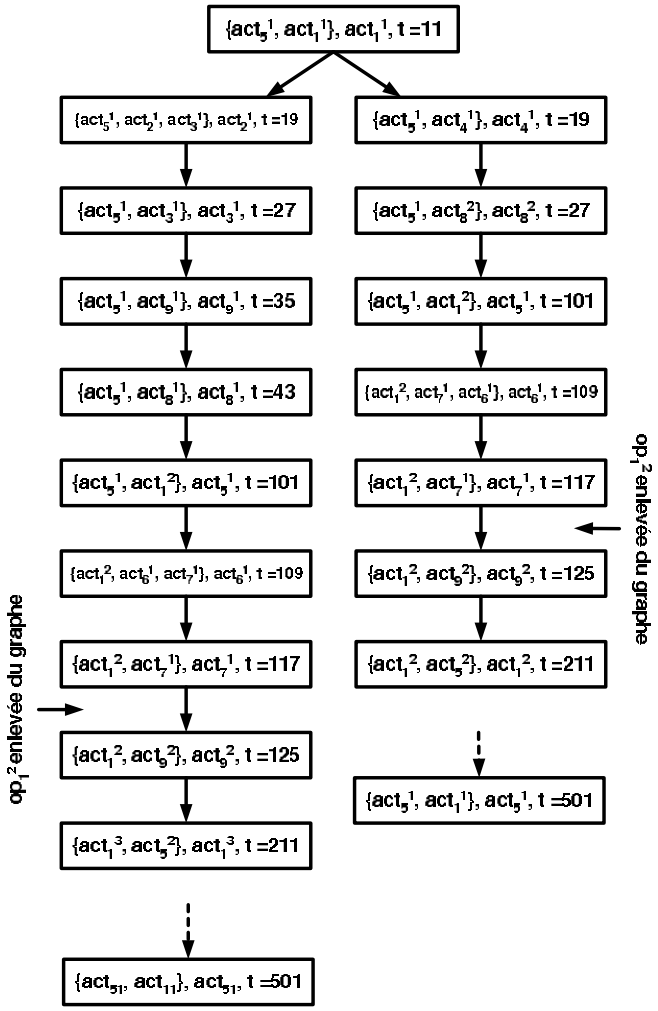

FIG. 4.7 – Un graphe d'accessibilité dégradé analysant la faisabilité d'une application.

Dans ce cas, nous concluons à la faisabilité de l'application sous des contraintes sur les temps de réponse de bout en bout, sous une garantie $(1,2)$*-firm pour* op_1 *et une garantie stricte pour* op_5.

4.5.3 Complexité

En plus des paramètres considérés dans la complexité en mode strict, nous définissons $\phi = \Sigma_{j=0}^{n-1}(k_j - m_j)$, le nombre maximal de retours arrières dans le graphe G.

Le nombre maximal de classes d'actions à construire dans le graphe est égal à $\alpha * \beta * \phi$. La complexité de l'algorithme est alors en $O(\alpha * \beta * \phi)$.

4.5.4 Génération d'un ordonnancement statique pour le séquenceur

Une fois déterminée la faisabilité d'une application dans une ressource, il est nécessaire de déduire sa fonction d'ordonnancement [Khalgui *et al.*, 2006c]. Cette fonction, exécutée par un séquenceur, a pour rôle d'allouer le processeur aux différents Blocs Fonctionnels de l'application. Nous rappelons qu'il s'agit d'un ordonnancement calculé hors-ligne, non-préemptif et oisif.

Pour calculer cette fonction, nous transformons le graphe d'accessibilité en un ordonnancement statique. Cet ordonnancement est un graphe acyclique dont chaque trajectoire est un comportement possible de l'application. A chaque noeud d'une trajectoire est associée une instance d'action à exécuter et la date du début de son exécution. Ceci revient à considérer le graphe d'accessibilité dans lequel on fait une projection de chaque noeud $(set, act, time)$ dans $(act, time)$.

Exemple. *Pour illustrer la méthode présentée, nous revenons à la configuration de l'exemple présentée au début de ce chapitre.*

En appliquant la technique proposée, nous transformons le graphe d'accessibilité généré précédemment vers un ordonnancement statique 4.8.

4.5.5 Politique de sélection d'événement

Dans la norme IEC 61499, le séquenceur n'est censé attribuer le processeur qu'à des Blocs Fonctionnels ; c'est en fait l'ECC de chaque Bloc Fonctionnel qui a la charge de sélectionner un événement parmi les occurrences en attente (ceux qui sont mémorisés dans le buffer associé aux ports).

Partant donc d'un ordonnancement statique des instances d'actions sur une ressource tel que développé précédemment, il reste à munir l'ECC d'un algorithme de sélection des événements compatible à cet ordonnancement. Notons que la politique de sélection des événements à l'intérieur d'un Bloc Fonctionnel n'a pas encore fait l'objet d'étude [Lewis, 2002].

Pour définir cette politique de sélection, nous faisons une projection du graphe d'accessibilité sur les événements d'entrée du bloc [Khalgui *et al.*, 2005a]. Cela donne donc un ordre partiel de priorité sur les instances des événements. Notons que si deux instances de deux événements sont non comparables, c'est qu'elles appartiennent à deux branches différentes du graphe d'accessibilité. Par construction, on peut dire que ces deux instances ne peuvent pas être simultanées et donc l'ordre suffit toujours à faire la sélection.

En considérant l'analyse d'ordonnançabilité dans l'hyper-période $hp = [r_{min} + j_{min}; r_{max} + j_{max} + 2.lcm]$, nous proposons de déterminer, pour chaque bloc, l'ordre des événements dans cette période.

Soit ee_i^m et ee_j^n deux instances de deux événements d'un Bloc Fonctionnel. L'ECC choisit le traitement de l'instance ee_i^m avant l'instance ee_j^n si l'ordonnancement lance l'exécution de l'instance act_i^m avant celle de act_j^n.

Exemple. *Dans l'exemple traité tout au long du chapitre, nous proposons de déduire du graphe construit précédemment, la politique de sélection des événements dans FB_2.*

Dans l'ordonnancement statique des actions obtenu, l'exécution de l'action act_2 précède celle de act_6. De ce fait, les occurrences de l'événement ie_2 sont plus prioritaires que celles de l'événement ie_6.

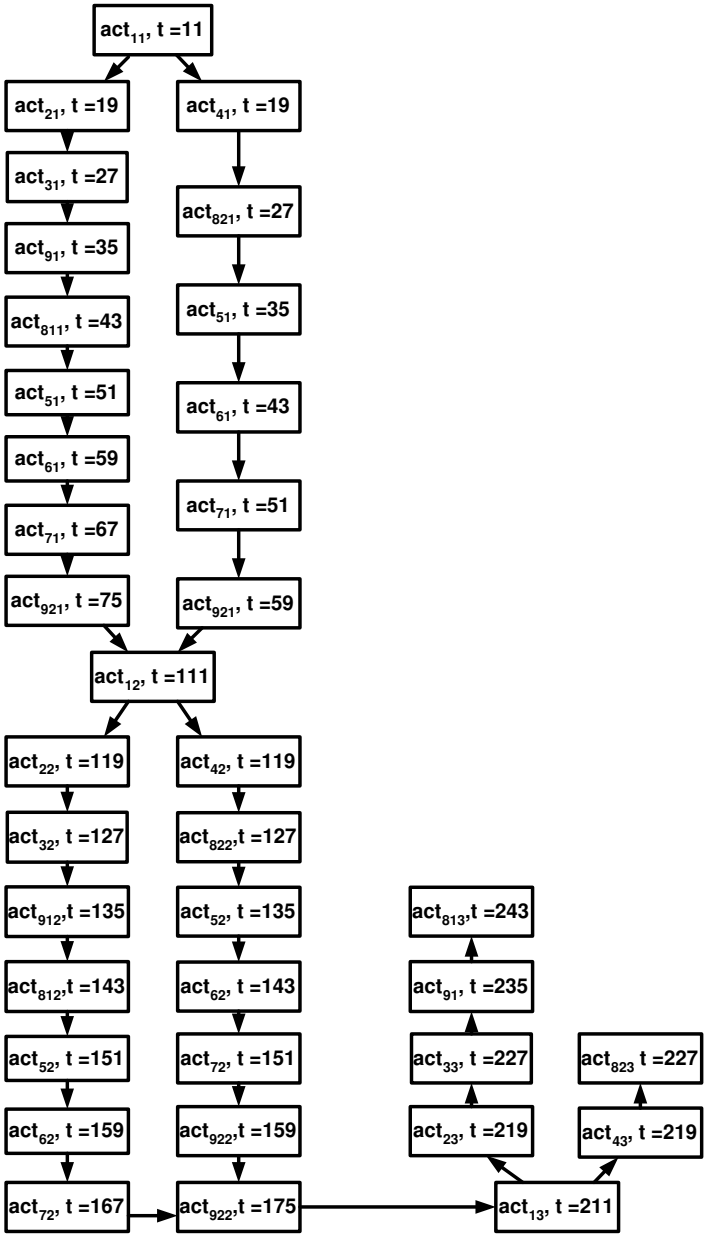

FIG. 4.8 – Un ordonnancement statique strict d'une application de contrôle.

4.6 Conclusions

Dans ce chapitre, nous avons proposé une approche validant le comportement temporel [Khalgui *et al.*, 2004b] d'une application de contrôle, spécifiée comme un réseau de Blocs Fonctionnels, dans une ressource d'un dispositif sous réserve d'arrivées périodiques des événements d'entrées de l'application et devant respecter des échéances relatives sur les temps de réponses de bout en bout.

D'aprés la norme IEC 61499, l'exécution des Blocs Fonctionnels de l'application est exclusivement non-préemptive. Afin de modéliser une telle application, nous avons proposé de transformer le réseau des Blocs Fonctionnels en un système d'actions sous contraintes de précédences. Ce système modélise tous les comportements possibles des différents blocs. Ainsi, pour vérifier la faisabilité de l'application, nous analysons l'ordonnançabilité du système d'actions correspondant. Cette analyse, basée sur la politique hors-ligne EDF non-préemptive, construit un graphe d'accessibilité dans une hyper-période bien déterminée.

Dans le cas où l'application est faisable, nous avons proposé une approche construisant l'ordonnancement statique correspondant. Cet ordonnancement définit, pour chaque action, la date possible du début d'exécution de ses instances.

Dans le cas où la violation de certaines contraintes temporelles est autorisée sous condition (définies dans le cahier des charges), l'application est exécutable dans un mode dégradé. Dans ce cas, nous proposons une approche analysant l'ordonnançabilité pour une garantie de dégradation acceptable de type (m, k)-firm dynamique. Si l'ordonnancement est faisable pour ce type de garantie, un ordonnancement statique dégradé est alors généré.

Enfin, nous avons défini une politique de sélection d'événements dans chaque bloc. Cette politique repose sur l'ordonnancement statique généré pour garantir une cohérence entre les ECC des différents blocs et la fonction d'ordonnancement (de la ressource).

Chapitre 5

Ordonnancement d'une application distribuée sur des dispositifs à multi-ressources

5.1 Introduction

Une nouvelle technologie de machines multi-tâches (Multi-tasking Programmable Logic Controlers) est de plus en plus utilisée dans l'industrie [Rockwell, 2006]. Ces machines, remplaçant les automates programmables mono-tâche, contiennent des systèmes d'exploitation supportant des politiques d'ordonnancement préemptives en-ligne [Takada and Sakamura, 1995].

Il est alors possible de déployer une application de contrôle, reposant sur la technologie des Bloc Fonctionnels, au sein de plusieurs tâches de telles machines. Cette approche est intéressante car les politiques en ligne s'avèrent plus efficaces en général que celles hors-ligne. Néanmoins, chaque exécution d'un Bloc Fonctionnel élémentaire (i.e. chaque action) représente un calcul élémentaire [Lewis, 2002]. De plus, la commutation de contexte lors de l'exécution de tâches induit généralement un surcoût temporel. De ce fait, il ne paraît pas judicieux de déployer chaque action comme une tâche. Une solution est de regrouper des actions au sein d'une même tâche.

Le problème dans ce chapitre est donc de déployer l'ensemble des actions dans des tâches préemptives. Au chapitre 4, nous avons spécifié la construction du séquencement d'un ensemble d'actions dans une ressource. Dans ce chapitre, nous exploitons cette technique pour séquencer un ensemble d'actions dans une tâche. Néanmoins, pour préserver la nom-préemption entre actions d'une même ressource, toute ressource est intégralement déployée au sein d'une seule tâche.

Nous proposons donc une méthode hybride alliant le séquencement hors-ligne oisif des actions d'une tâche d'une part, et l'ordonnancement en-ligne non oisif et préemptif entre les tâches d'un même PLC d'autre part [Khalgui *et al.*, 2006b].

Une autre évolution des systèmes industriels repose sur la distribution de l'application sur un réseau de PLCs (*Network Control System* [K.M and K.G, 1995]). Le déploiement d'une application à base de Bloc Fonctionnels revient alors à décider non seulement la distribution des actions au sein des tâches mais aussi la distribution de ces tâches dans les PLCs.

Dans la seconde partie de ce chapitre, nous considérons que l'application est déjà distribuée sur plusieurs ressources de plusieurs dispositifs [Khalgui *et al.*, 2006a]. Nous considérons un bus CAN [network, 1993] comme réseau connectant ces dispositifs. Il assure l'échange des événements

entre des Bloc Fonctionnels de l'application.

La validation temporelle de l'application repose, comme décrit précédemment, sur une méthode hybride d'ordonnancement. Cette validation vérifie le comportement temporel,

- de l'ensemble des Bloc Fonctionnels dans chaque dispositif de l'architecture,
- des messages échangés entre dispositifs,

Pour valider le trafic sur le réseau, nous proposons une caractérisation temporelle des messages échangés. Cette caractérisation sert à calculer les instants de réception des événements contenus dans ces messages pour activer des Bloc Fonctionnels. On peut donc remarquer que les validations de la messagerie et de l'ordonnancement sont intimement liés.

Pour aborder le problème de la construction des tâches, nous proposons de réutiliser des résultats existants. Ces derniers sont présentés dans la première partie de ce chapitre. Dans la deuxième partie, nous présentons l'approche hybride en considérant une application distribuée sur plusieurs ressources d'un seul dispositif. Nous étendons, dans la troisième partie, la validation à une distribution sur un réseau de dispositifs. Enfin, nous concluons sur les avantages et les inconvénients de l'approche proposée.

5.2 Tâches récurrentes temps-réel

Dans la littérature des systèmes temps-réel, plusieurs modèles de tâches dépendantes ont été proposés pour modéliser des systèmes à traitements dépendants [Cottet *et al.*, 2000; Stankovic *et al.*, 1998].

Dans [Baruah, 2003], un modèle de tâches dites "récurrentes temps réel" est proposé pour modéliser des systèmes à comportement conditionnel. Ce modèle de tâche est considéré, de nos jours, comme un des modèles le plus généraux dans la littérature des systèmes temps-réel. Une tâche récurrente T est un ensemble de sous-tâches sous contraintes de précédence. On caractérise cette tâche par,

* Un graphe $G(T)$ acyclique direct ayant un seul état source et un seul état final.

  Chaque état de ce graphe représente une sous-tâche u caractérisée par le couple $(e(u), d(u))$. les quantités $e(u)$ et $d(u)$ représentent respectivement le pire temps d'exécution et l'échéance de la sous-tâche.

  De plus, toute transition entre deux états u et v de $G(T)$ est caractérisée par une période minimale : $p(u, v)$. Cette période représente le temps minimal séparant les dates d'activation de u et de v.

* Une période $P(T)$ représentant la durée minimale qui sépare l'activation de deux instances successives de l'état source.

Dans une tâche récurrente T, lorsqu'une sous-tâche u est activée à un instant t, une quantité de travail de pire temps d'exécution $e(u)$ est activée. Cette quantité est à exécuter avant une échéance $d(u)$ relative à la date d'activation.

- Si u n'est pas un état final de $G(T)$, alors le prochain état à activer est un état v successeur de u dans $G(T)$. Notons que l'exécution de v est possible après $t + p(u, v)$.
- Si u est un état final de $G(T)$, alors le prochain état à activer est l'état source du graphe. L'activation de cet état est possible après au moins $P(T)$ de sa dernière activation.

Pour caractériser l'exécution d'une tâche récurrente, les fonctions suivantes sont proposées,

- $E(T)$: représente le temps d'exécution maximal de la tâche T sur ses trajectoires entre l'état source et l'état final.

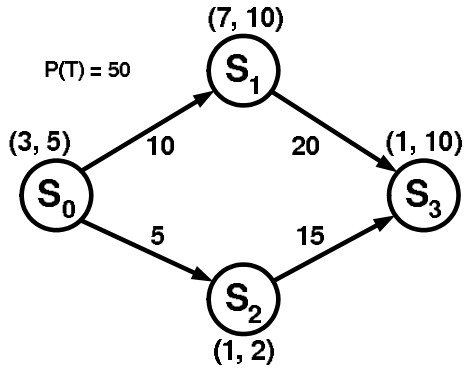

FIG. 5.1 – Un exemple d'une tâche récurrente

– $\rho_{ave}(T)$: représente la quantité $E(T)/P(T)$.

Nous présentons dans la figure 5.1 l'exemple d'une tâche récurrente. Notons que $E(T) = 11$ et $\rho_{ave}(T) = 0.22$.

Pour analyser la faisabilité de tout système de tâches récurrentes, [Baruah, 2003] définit les deux fonctions suivantes à appliquer sur chaque tâche T :

– $T.dbf(t)$: la quantité de travail maximale de T ayant la date d'activation et l'échéance dans tout intervalle de durée t.
– $T.rbf(t)$: la quantité de travail maximale de T ayant la date d'activation dans tout intervalle de durée t.

Dans le cas où les tâches récurrentes sont à priorités dynamiques, une condition d'ordonnançabilité *nécessaire* et *suffisante* a été proposée. Cette condition est à vérifier dans une hyper-période de longueur hp pour un système de tâches récurrentes S,

$$hp = [0, \frac{\sum_{\Gamma \in S} 2.E(\Gamma)}{1 - \sum_{\Gamma \in S} \rho_{ave}(\Gamma)}]$$

Précisons que [Baruah, 2003] propose une technique pour calculer cette hyper-période.

Condition d'ordonnançabilité. un ensemble de tâches récurrentes S est faisable si et seulement si,

$$\forall t \in hp\,, (\sum_{\Gamma \in S} \Gamma.dbf(t) \leq t))$$

Cette analyse de faisabilité est un problème pseudo-polynomial. Dans le cas où une tâche récurrente est non ordonnançable, il est facile par une simple analyse sur l'hyper-période de chercher l'ensemble de ses sous-tâches ne respectant pas les échéances correspondantes.

5.3 Validation hybride dans un dispositif

Selon le standard IEC 61499, les Bloc Fonctionnels d'une ressource peuvent partager des données ou la commande de procédés physiques. Le standard impose une politique non-préemptive pour ordonnancer ces blocks. Cela implémente de fait une exclusion mutuelle pour l'accès aux données partagées (ou à l'actionneur d'un procédé physique). On a donc tout intérêt à distribuer

l'ensemble de la ressource au sein d'une tâche OS afin de préserver cette caractéristique. L'enchaînement des instances d'actions au sein d'une tâche est défini statiquement [Khalgui *et al.*, 2006b].

Pour ordonnancer les Bloc Fonctionnels de l'application dans chaque ressource, nous appliquons une politique non-préemptive hors-ligne [Stankovic *et al.*, 1998] vérifiant les contraintes temporelles correspondantes [Khalgui *et al.*, 2005a]. Nous distinguons deux cas,

- Si l'ordonnancement de ces blocs n'est pas faisable, alors nous déduisons l'application dans son ensemble non ordonnançable.
- Sinon, nous générons un ordonnancement statique [Cucu and Sorel, 2005] implémentant un scénario d'exécution de ces blocs.

Dans ce manuscrit, nous considérons un ordonnancement statique d'une ressource comme un pré-ordonnancement des instances d'actions correspondantes au sein de la tâche [Mok and Feng, 2001].

Le pré-ordonnancement ainsi construit correspond à un séquencement conditionnel des actions. Nous proposons donc de ré-utiliser le modèle de tâche conditionnelle [Baruah, 2003] pour coder les différents scénarios possibles d'exécution de la tâche.

Notons qu'il faut caractériser temporellement l'activation de chaque tâche du PLC. Cette caractérisation se base sur les dates d'arrivée des occurrences d'événements en entrée de la ressource. Partant de la caractérisation temporelle des événements présentée dans le chapitre précédent, nous en déduisons les dates d'activation des tâches. Comme cette caractérisation tient compte des dépendances au sein du réseau de Bloc Fonctionnels, on peut considérer les tâches comme indépendantes bien que les ressources correspondantes peuvent être en dépendance.

Exemple. *Nous continuons dans ce chapitre le même exemple explicatif du chapitre précédant. Nous supposons dans un premier temps que l'application est distribuée sur deux ressources d'un même dispositif (figure 5.2). En particulier, nous précisons que la ressource r_2 est dépendante de la ressource r_1.*

Une fois l'architecture fonctionnelle de l'application (réseau de Bloc Fonctionnels) est transformée vers une architecture opérationnelle (ensemble de tâches indépendantes), nous proposons d'appliquer la condition d'ordonnançabilité des tâches récurrentes vérifiant la faisabilité de ces tâches dans un PLC.

Cette approche hybride reste toute à fait compatible avec le standard IEC61499. En effet, elle applique un ordonnancement non-préemptif dans chaque ressource d'un dispositif. De plus, notons que cette approche tient compte des caractéristiques du système d'exploitation comme le nombre limité des tâches à construire [Takada and Sakamura, 1995]. Nous montrons dans le chapitre suivant une heuristique réduisant un tel nombre en pré-ordonnançant les actions de plusieurs ressources à déployer dans une seule tâche.

La figure 5.3 reprend les différentes étapes de l'approche hybride. Cette approche devra être appliquée sur chaque dispositif de l'architecture distribuée.

5.3.1 Pré-ordonnancement des ressources

En se basant sur la contribution du chapitre précédent, nous validons le comportement temporel des Bloc Fonctionnels d'une ressource en appliquant une analyse d'ordonnançabilité [Khalgui *et al.*, 2005a; 2006d]. Une fois la faisabilité vérifiée, nous générons un ordonnancement statique (ou un pré-ordonnancement).

Dans toute la suite, nous dénoterons par S_r l'ordonnancement obtenu sur la resource r. La figure 5.4 présente un exemple d'ordonnancement dans le dispositif. En particulier, le pré-

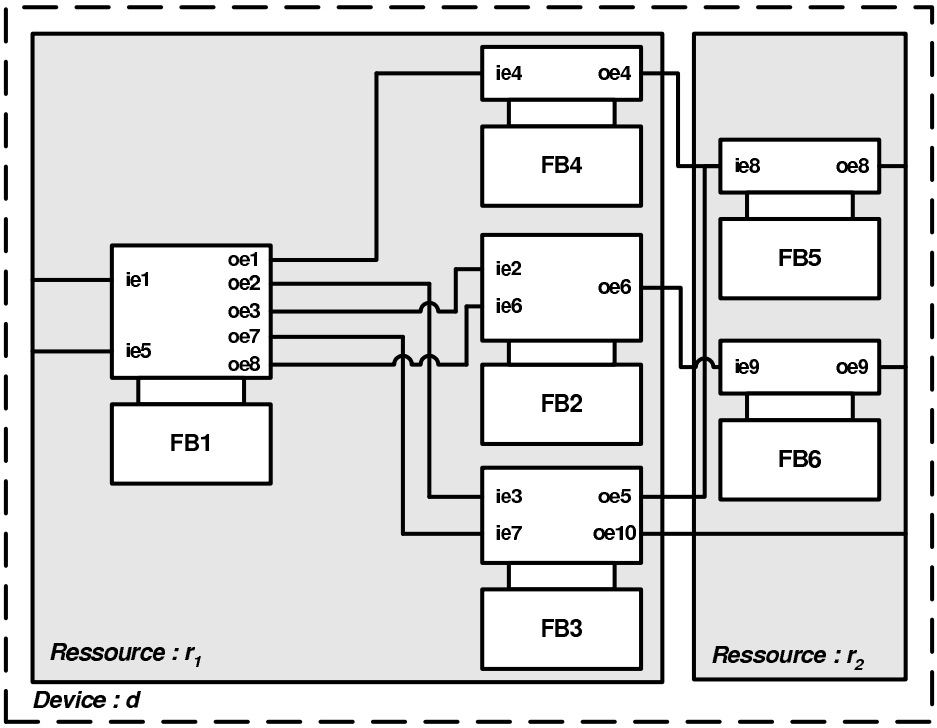

FIG. 5.2 – La distribution de l'application sur deux ressources

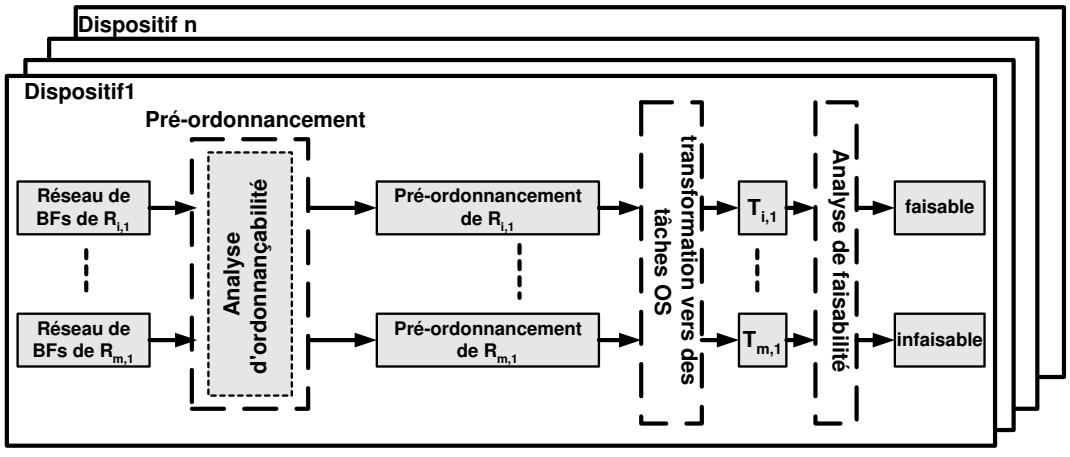

FIG. 5.3 – Les étapes de l'approche hybride d'ordonnancement

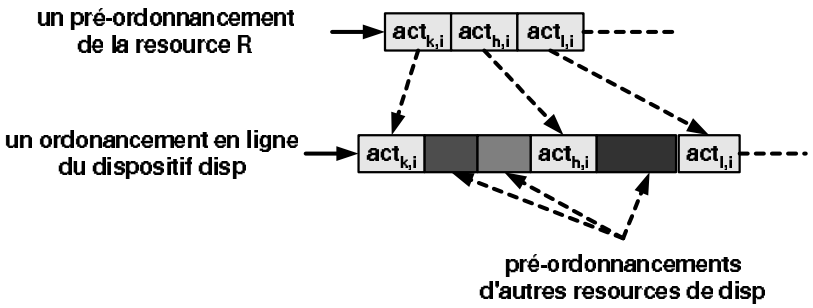

FIG. 5.4 – Un exemple d'ordonnancement en-ligne de pré-ordonnancements dans un dispositif

ordonnancement de la ressource est préempté pour exécuter des actions d'autres ressources.

Dans toute la suite, nous considérons les fonctions suivantes pour parcourir l'ordonnancement S_r. La fonction $pred\_stat(S)$ (resp, $follow\_stat(S)$) représente l'ensemble des noeuds dans S_r précédant (res, suivant) un noeud S.

Exemple. *Dans la figure 5.5, nous présentons l'ordonnancement statique de la ressource* r_1 *(voir le chapitre précédent). Cet ordonnancement implémente le comportement de la ressource. Nous utilisons cet exemple dans toute la suite du chapitre*

5.3.2 Transformation vers des tâches OS

Nous proposons une approche transformant l'architecture fonctionnelle de l'application vers une architecture opérationnelle [Khalgui *et al.*, 2006b] : une fois que toutes les ressources du dispositif sont ordonnançables, nous transformons les pré-ordonnancements correspondants vers des tâches OS.

Découpage en tâches OS

En tenant compte de la structure conditionnelle des pré-ordonnancements générés, nous proposons d'utiliser le modèle de tâches récurrentes [Baruah, 2003] pour construire les tâches OS correspondantes. Dans ce chapitre, nous considérons le déploiement d'une seule ressource au sein d'une seule tâche.

Soit R une ressource du dispositif considéré. En considérant une hyper-période d'analyse $hp = [r_{min} + j_{min}; r_{max} + j_{max} + 2.lcm]$ (voir chapitre précédent), nous distinguons deux modes de comportement dans la ressource : le mode stationnaire et le mode non-stationnaire.

- Le mode non-stationnaire correspond au comportement des Bloc Fonctionnels dans l'intervalle de temps $[r_{min} + j_{min}; r_{max} + j_{max}]$. Durant cette période, toutes les actions ne sont pas encore activées. Ce comportement n'est pas donc périodique.
- Le mode stationnaire correspond au comportement dans l'intervalle de temps $[r_{max} + j_{max}; r_{max} + j_{max} + 2.lcm]$: cette portion du graphe se répète périodiquement.

Tenant compte de cette caractérisation, nous proposons de transformer le pré-ordonnancement S_R vers deux tâches récurrentes : Γ_s et Γ_{ns}.

- La tâche Γ_s implémente le comportement stationnaire de la ressource. Ce comportement est périodique de période $2.lcm$.

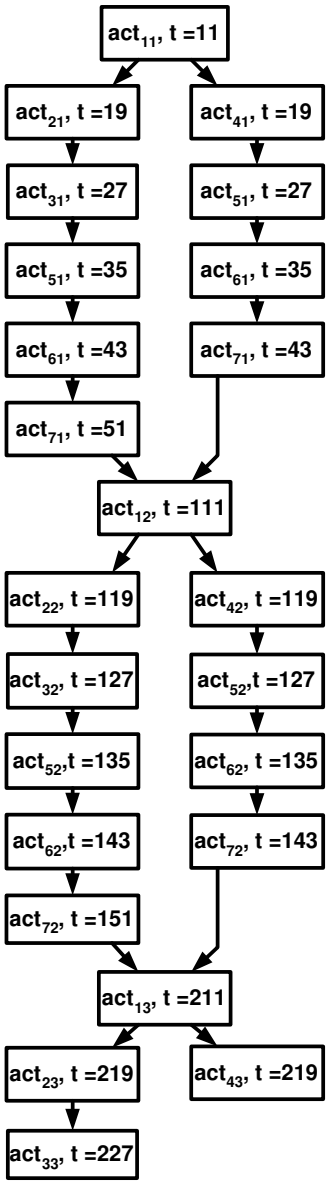

FIG. 5.5 – L'ordonnancement statique de la ressource r_1

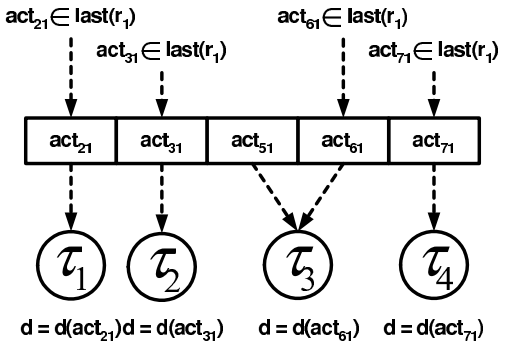

FIG. 5.6 – Une migration d'une branche d'un pré-ordonnancement vers une trace de sous-tâches.

– La tâche Γ_{ns} implémente le comportement non-stationnaire de la ressource. Dans ce manuscrit, nous supposons que sa période est ∞. Autrement dit, cette tâche est apériodique.

Création des sous-tâches

Pour transformer un pré-ordonnancement S_R vers une tâche récurrente Γ, une solution possible est de migrer chaque instance d'action de S_R vers une sous-tâche unique de Γ. Dans ce cas, le nombre de sous-tâches à construire est maximal.

Cette solution n'est pas optimale vu que l'analyse d'ordonnançabilité est fonction du nombre de sous-tâches. Nous optimisons la transformation en migrant (sous conditions) toute une séquence d'instances de S_R vers une sous-tâche de Γ (figure 5.6). De ce fait, la complexité de l'analyse d'ordonnançabilité en est réduite.

Nous précisons qu'il y a cependant des limites à l'agrégation d'actions au sein de sous-tâches. En effet, le modèle de tâche récurrente n'autorise des échéances que sur des sous-tâches. Pour tenir compte des bornes sur les temps de réponse de bout en bout, il est alors nécessaire que la dernière action de chaque trace soit la dernière action d'une sous-tâche. Dans ce cas, l'échéance de la sous-tâche correspond à celle de l'action.

Example. *Dans la figure 5.6, nous présentons la migration d'une trajectoire de l'ordonnancement statique de r_1 vers une séquence de sous-tâches.*

L'échéance de la sous-tâche τ_1 est celle de l'instance $act_{2,1}$ ($act_2 \in last(\sigma)$). De ce fait, τ_1 ne respecte pas son échéance si et seulement si la borne sur le temps de réponse correspondant à act_2 n'est pas vérifiée.

Nous proposons la méthode suivante pour construire toute tâche récurrente Γ_s d'une ressource donnée.

Nous construisons une sous-tâche τ de Γ_s de la façon suivante.

$$\tau = act_0^e, ..., act_{k-1}^f \text{ telle que,}$$

– $\forall i \in [0, k-2], follow\_stat(act_i^h) = \{act_{i+1}^l\}$,
– act_{k-1} doit être, ou bien une action finale d'une trace, ou bien une action avec plus qu'un successeur dans S_R,

$$act_{k-1} \in last(\sigma)$$

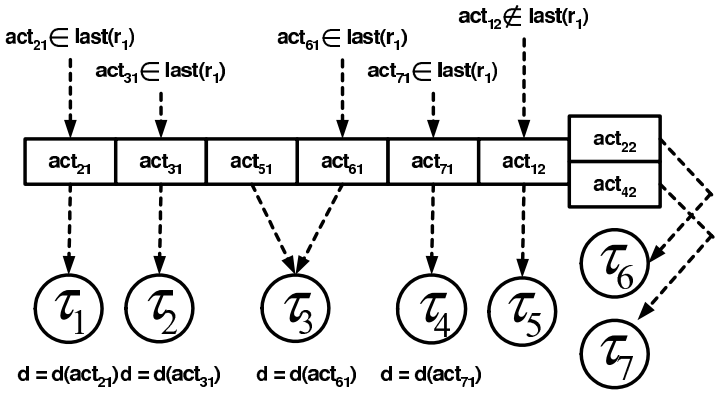

FIG. 5.7 – Un exemple de migration d'un pré-ordonnancement vers une tâche récurrente.

$$\text{ou cardinalité}(follow\_stat(act^f_{k-1}))>1$$

Exemple. *Dans la figure 5.7, l'instance* act_{12} *(*$act_{12} \notin last(\sigma)$*) a deux successeurs* act_{22} *et* act_{42}*. De ce fait, on lui construit une sous-tâche* τ_5 *ayant comme successeurs* τ_6 *et* τ_7*. Notons que l'échéance de* τ_5 *n'est que l'échéance de* act_{12}*.*

Enfin, nous dénotons dans toute la suite par $first(\tau)$ (resp $last(\tau)$) la première (resp, la dernière) instance de la sous-tâche τ. De même, nous dénotons par $first\_sub(S_R)$ l'ensemble des instances d'actions de S_R sans prédecesseurs dans le mode stationnaire.

Construction de la tâche complète

Nous proposons les règles suivantes pour construire Γ_s. En considérant le modèle proposé dans [Baruah, 2003], nous définissons les fonctions suivantes pour traiter une tâche récurrente Γ :

- $pred(\tau)$: l'ensemble des sous-tâches qui précédent la sous-tâche τ dans Γ.
- $succ(\tau)$: l'ensemble des sous-tâches qui suivent la sous-tâche τ dans Γ.

De plus, dénotons par $end(\Gamma)$ un sous ensemble de sous-tâches n'ayant pas des successeurs dans Γ

$$end(\Gamma) = \{\tau \in \Gamma / succ(\tau) = \varnothing\}$$

La première règle permet la construction de la première sous-tâche alors que la deuxième permet la construction recursive du reste des sous-tâches. Enfin la troisième construit la sous-tâche finale.

* **Règle 0.** Construction de la première sous-tâche.

Dans [Baruah, 2003], une tâche récurrente est caractérisée par un seul état source. De ce fait, il faut éventuellement adapter le pré-ordonnancement.

** **Si** cardinalité($first\_sub(\ S_R)) = 1$

Alors une seule instance est déployée dans la sous-tâche τ_0.

$$\{\tau_0\} = first\_sub(S_R)$$

** **Sinon**, il faut créer une sous-tâche *vide* pour avoir une sous-tâche unique en tête de la tâche. Nous construisons dans G(Γ) une sous-tâche virtuelle τ_0 de la façon suivante :

*** WCET(τ_0) = 0.

*** Nous relions cette sous-tâche aux suivants. Pour chaque instance $act_i^j \in first\_sub(S_R)$, nous construisons une sous-tâche τ_k telle que (τ_0, τ_k)∈ G(Γ) et p(τ_0,τ_k) = 0.

*** La date d'activation de la sous-tâche τ_0 est comme suit,

$$t(\tau_0) = min\{r_i + j * p_i, act_i^j \in first\_sub(S_R)\}$$

* **Règle 1.** Construction récursive des sous-tâches.

Soit τ_i une sous-tâche de Γ telle que $act_0^q \in follow\_stat(\text{last}(\tau_i))$. Soit $\tau_j \in \text{succ}(\tau_i)$ une sous-tâche de la façon suivante,

$$\tau_j = act_0^q, ..., act_k^p,\ act_k^p \in last(\sigma)$$

En se basant sur le modèle de tâche récurrente [Baruah, 2003], nous proposons les paramètres suivantes de la sous-tâche τ_j (tenant compte de ses prédécesseurs),

** **la date d'activation** t(τ_j) est égal à la valeur maximale entre,

*** $r_0 + q * p_0$: la date d'activation de l'instance $act_0^q = first(\tau_j)$

*** La date de fin d'exécution au plus tôt de la sous-tâche τ_j.

$$\text{t}\ (\tau_j) = \max\{r_0 + q * p_0\ ;\ max_{\tau_i \in pred(\tau_j)}\ \{\ \text{t}\ (\ \tau_i\ ) + \sum_{act_p^q \in \tau_i} \text{BCET}(act_p)\ \}\}$$

** **la période minimale entre sous-tâches** p(τ_i , τ_j) est égale à la différence entre le temps d'activation de τ_j et τ_i.

$$p_j = \text{t}(\tau_j)\text{- t}(\tau_i)$$

** **L'échéance** d_j , correspond à l'échéance d_k de l'instance act_k^p. Le respect de cette échéance garantit le respect des bornes sur le temps de réponse correspondantes à l'action act_k

** **Le temps d'exécution** WCET(τ_j) est la somme des WCET des différentes actions qui implémentent la sous-tâche τ_j.

* **Règle 2.** Génération de la sous-tâche finale.

Le modèle de tâche récurrente impose que cette sous-tâche soit unique. Nous construisons donc une sous-tâche finale unique. Cette démarche est analogue à la **Règle 0**.

Nous distinguons deux cas,

** Si $cardinality(end(\Gamma)) = 1$, alors la sous tâche de $end(\Gamma)$ est la sous-tâche finale de Γ.

** Sinon, Nous construisons une sous-tâche *vide* τ_{finale} cible des sous-tâches de $end(\Gamma)$ telle que,

*** $\forall \tau \in end(\Gamma)$,

**** $\tau = pred(\tau_{finale})$,

**** $p(\tau, \tau_{finale}) = 0$,

*** $WCET(\tau_{finale}) = 0$,

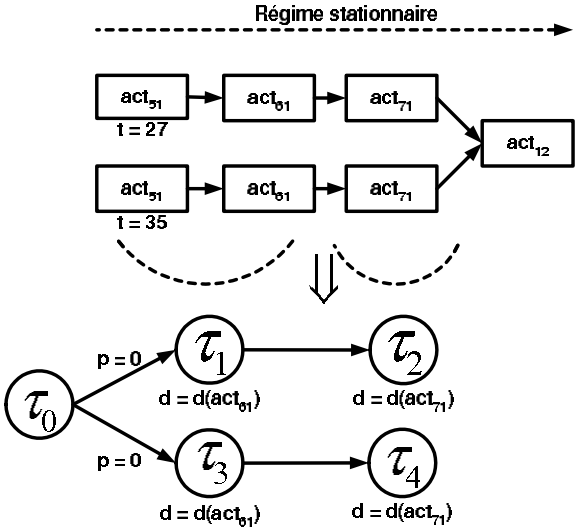

FIG. 5.8 – Un exemple de migration vers une tâche récurrente.

Exemple. *Dans la figure 5.8, nous présentons un exemple de construction d'une tâche récurrente.*

Dans l'ordonnancement statique généré, l'ensemble $first\_sub(S_{r_1}) = \{act_5^1, act_5^1\}$*, nous avons alors à construire une sous-tâche virtuelle notée* τ_0*. La date d'activation de cette sous-tâche est comme suit,*

$$t(\tau_0) = min\{r_i + j * p_i \in first\_sub(S_R)\}$$

La date d'activation de la sous-tâche τ_2 *est caractérisée comme suit,*
$t(\tau_2) = max\{r_7 + p_7; t(\tau_1) + \sum_{act_p^q \in \tau_1} BCET(act_p)\}$
Finalement, notons que nous appliquons la même méthode pour construire la tâche récurrente Γ_{ns} de toute ressource du dispositif.

Algorithme

Nous proposons un algorithme permettant la construction d'une tâche récurrente Γ à partir d'un pré-ordonnancement S_R.

Nous définissons l'enregistrement *sub_task* permettant de définir le type d'une sous-tâche de Γ. De plus, nous utilisons l'enregistrement *stat_sched* comme la définition du type d'un noeud de S_R.

La fonction $transform()$ est appliquée récursivement pour construire les différentes sous-tâches de Γ. Cette fonction récupère comme paramètres :

* τ : une sous-tâche venant d'être construire.
* S_1 : un noeud de S_R pour lequel nous cherchons à construire une nouvelle sous-tâche ($S_1 \in follow\_stat(last(\tau))$.
* Γ : première sous-tâche de Γ.
* S_R : l'ordonnancement statique généré.

Nous précisons que la fonction,

* $exist(S, \Gamma)$ recherche une sous-tâche déjà construite et ayant comme premier noeud S ($S \in S_R$),
* $frame(\tau', S, S')$ construit une nouvelle sous-tâche contenant la séquence des noeuds de S à S'.
* $temp\_char(\tau, \tau', S_R)$ caractérise temporellement la sous-tâche τ' à partir des caractéristiques de τ.

transform(τ : *sub_task*, S_1 : *stat_sched*, Γ : *sub_task*, *sched* : *stat_sched*)
Debut
  τ_1 : *sub_task*;
  S_3, S_2 : *noeud_sched*
  $S_2 \leftarrow S_1$;
  Si $card(follow\_stat(S_1) \neq \varnothing$
    Alors,
      Tant que $card(follow\_stat(S_2) = 1$ et $\{S_2.act\} \nsubseteq last$ faire
        $S_2 \leftarrow follow\_stat(S_2)$;
      Ffaire.
      $\tau' \leftarrow exist(S_1, \Gamma)$;
      Si $(\tau' = NULL)$
        Alors $frame(\tau', S_1, S_2, S_R)$;
        $add(\tau', succ(\tau))$;
        $add(\tau, pred(\tau'))$;
        $temp\_char(\tau, \tau', S_R)$;
        Pour chaque $S_3 \in follow\_stat(S_2)$ faire
          transform$(\tau', S_3, \Gamma, S_R)$;
        Ffaire.
      Fsi.
  Fsi.
Fin.

Exemple. *Nous présentons dans la figure 5.9, la tâche récurrente implémentant la ressource r_1 dans le régime périodique.*

La sous-tâche τ_1 implémente les deux instances act_{51} et act_{61}. Elle possède les caractéristiques suivantes,

- *La date d'activation : $t = 30$*
- *L'échéance est celle de l'instance act_{61} : $dd = 92$*
- *le pire temps d'exécution : $WCET(\tau_1) = WCET(act_{51}) + WCET(act_{61}) = 16$*

La sous-tâche τ_5 implémente l'instance act_{12}. Nous l'implémentons de la façon suivante,

- *La date d'activation est $t = 110$*
- *L'échéance est celle de l'instance act_{12} : $d = 168$*
- *le pire temps d'exécution $WCET(\tau_5) = WCET(act_{12}) = 8$*

5.3.3 Analyse de faisabilité en ligne

Une fois l'application transformée vers des tâches récurrentes indépendantes, nous vérifions leur faisabilité selon les modes de marche stricte [Khalgui *et al.*, 2005a] ou dégradé [Khalgui *et*

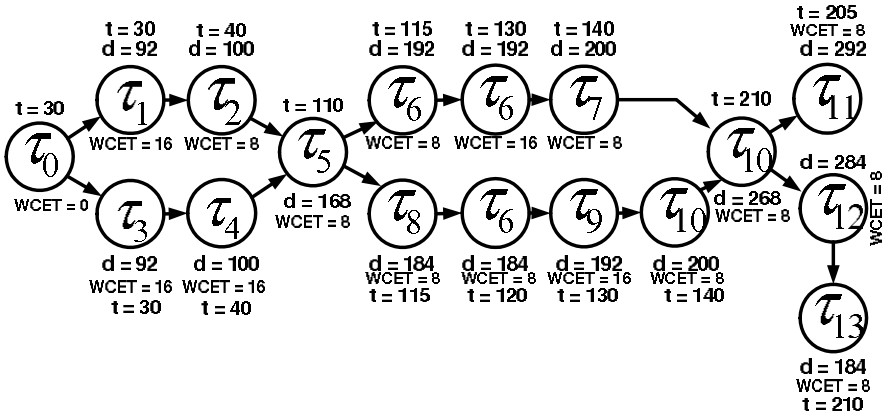

FIG. 5.9 – La tâche récurrente de la ressource r_1.

al., 2006d] (comme présenté dans le chapitre précédent). Pour cela, nous appliquons la condition de faisabilité nécessaire et suffisante proposée dans [Baruah, 2003].

Nous dénotons dans ce qui suit par $task\_set$ l'ensemble des tâches récurrentes construites dans le dispositif.

Faisabilité stricte

Ce cas représente une application directe du théorème proposé dans [Baruah, 2003].

Faisabilité dégradée

Nous proposons une approche validant la faisabilité de l'application déployée sous forme de tâches récurrentes en mode dégradé. Dans le cas où une tâche ne respecte pas ses échéances, l'idée de l'approche est de supprimer des actions du pré-ordonnancement correspondant en prenant en compte les contraintes de dégradation.

Soit op_i une opération de $oper(\Sigma)$. Nous dénotons par m_i^j le nombre de ses instances (op_i^j inclus) respectant leurs échéances parmi les k dernières.

Soit une sous-tâche $\tau \in \Gamma$ ($\Gamma \in task\_set$). Nous définissons $operation(\tau)$ l'instance d'opération contenant $last(\tau)$.

$$operation(\tau) = op_i^j \Longrightarrow last(\tau) \in op_i^j$$

En considérant un dispositif à multi-ressources, nous dénotons par $sched(op_i^j)$ les pré-ordonnancements de ressources contenant des instances de op_i^j.

$$sched(op_i^j) = \{S_R, op_i^j \in S_R\}$$

En appliquant la condition d'ordonnançabilité proposée dans [Baruah, 2003],

* **Si** toutes les tâches de $task\_set$ respectent leurs échéances, **alors** l'application est faisable.
* **Sinon,** il existe au moins une sous-tâche τ ($\tau \in \Gamma, \Gamma \in task\_set$) qui ne respecte pas son échéance. Nous supposons que $operation(\tau) = op_i^j$.
  Dans ce cas,

** **Si** $m_i^j > m_i$, **alors** une dégradation de $sched(op_i^j)$ est possible. Nous supprimons dans tous les pré-ordonnancements de $sched(op_i^j)$ les instances de op_i^j.

$$\forall S_R \in sched(op_i^j), S_R = S_R \backslash \{act_k^j, act_k \in op_i\}$$

** **Sinon**, une dégradation n'est pas possible. L'application n'est pas faisable

Nous proposons l'algorithme réalisant l'analyse de la faisabilité dégradée de l'application. Cet algorithme se base sur une fonction récursive *tester_faisabilité()* ayant comme paramètres :

- S : la liste de tâches récurrentes
- $Sched$: la liste de pré-ordonnancements.

En appliquant le théorème proposé dans [Baruah, 2003], la fonction *analyser(S)* analyse la faisabilité des tâches récurrentes et retourne *vrai* si elles sont faisables. La fonction *failed(S)* retourne la liste de sous-tâches ne respectant pas leurs échéances.

booléen tester_faisabilité($S : task\_set, Sched : Sched\_set$)
Début
  $\tau : subtask\_list$; $\tau \leftarrow NULL$;
  $res : boolen$; $res \leftarrow faux$;
  $S' : task\_set$;
  $Sched' : Sched\_set$;
  Si (analyser(S))
    Alors retourner vrai ;
  Sinon,
    Pour chaque $\tau \in failed(S)$ et $res = faux$ **faire**
      $op_p^q \leftarrow operation(\tau)$
      Si $m_p^q > m_p$
        Alors, $Sched' \leftarrow minus(Sched, op_p^q)$;
        $S' \leftarrow transformation(Sched')$;
        $res \leftarrow tester\_faisabilit(S', Sched')$;
      Fsi
    Ffaire
    retourner(res) ;
  Fsinon
Fin.

5.4 Construction de messagerie sur un réseau CAN

Dans cette section, nous nous intéressons à construire une messagerie pour le réseau considéré [Khalgui *et al.*, 2006a]. Chaque message doit transporter une occurrence d'un événement périodique ayant une échéance.

Nous proposons de nous inspirer de l'approche proposée dans [K.M and K.G, 1995] pour la validation temporelle d'un trafic de messages sur un réseau de type CAN [network, 1993]. La réception de chacun de ces messages doit respecter une échéance déduite des contraintes du cahier des charges.

Sur un réseau CAN, nous précisons que tout message m est classiquement caractérisé par un identifiant unique ID. Le dispositif, envoyant le message, utilise cet identifiant comme un degré de priorité statique. Pour tenir compte des contraintes temporelles, l'approche proposée dans

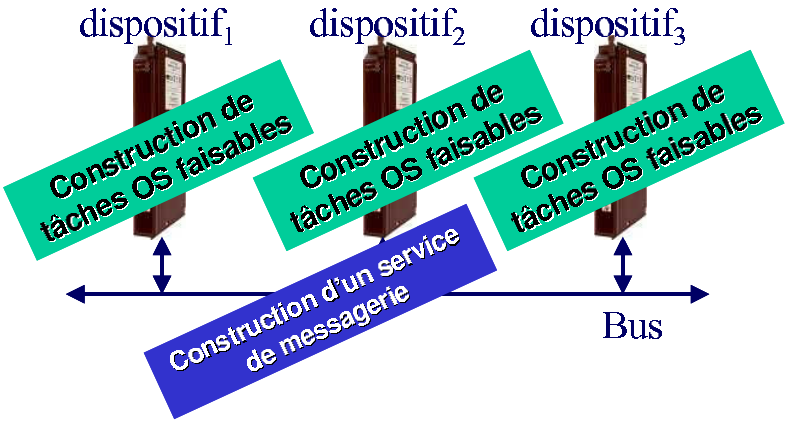

FIG. 5.10 – Principe de la validation d'une application distribuée

[K.M and K.G, 1995] construit l'identifiant de chaque message à partir de son échéance. De ce fait, le message le plus prioritaire à envoyer devient celui de plus courte échéance.

En se basant sur le modèle classique de [Liu and Layland, 1973], [K.M and K.G, 1995] considère une caractérisation temporelle des messages échangés sur le réseau. En considérant la nature du réseau CAN, la validation de l'ordonnançabilité des messages se base sur la politique EDF non-préemptive [Stankovic *et al.*, 1998]. Une condition d'ordonnançabilité, appliquant cette politique, est proposée pour vérifier le respect de toutes les échéances des messages [Cottet *et al.*, 2000].

Enfin, si le système de messagerie est temporellement faisable, [K.M and K.G, 1995] propose une technique calculant l'identifiant (i.e. priorité fixe) de chaque message à partir de son échéance.

Cette approche permet la construction d'une politique d'ordonnancement statique définissant une messagerie correcte [Khalgui *et al.*, 2006e].

5.5 Validation temporelle d'un réseau de dispositifs multi-ressources

Nous généralisons l'approche hybride en considérant une application distribuée sur un réseau de dispositifs multi-ressources [Khalgui *et al.*, 2006e]. Nous supposons que ces dispositifs sont des PLCs multi-tasking connectés par un bus CAN [Tindell *et al.*, 1994; 1995].

Pour valider le comportement global de l'application, nous devons (figure 5.10),

* Vérifier l'ordonnançabilité dans chaque ressource [Khalgui *et al.*, 2005a; 2006d],
* Construire les tâches récurrentes dans chaque dispositif [Khalgui *et al.*, 2006b],
* Vérifier l'ordonnançabilité de ces tâches,
* Construire une messagerie correcte vis à vis des bornes sur les temps de réponse [Khalgui *et al.*, 2006a]. Pour construire cette messagerie, nous devons calculer pour chaque message à échanger sur le réseau,
  ** Une caractérisation temporelle (date d'activation initiale, période, gigue),
  ** Une échéance basée sur les contraintes imposées par les dispositifs destinataires,

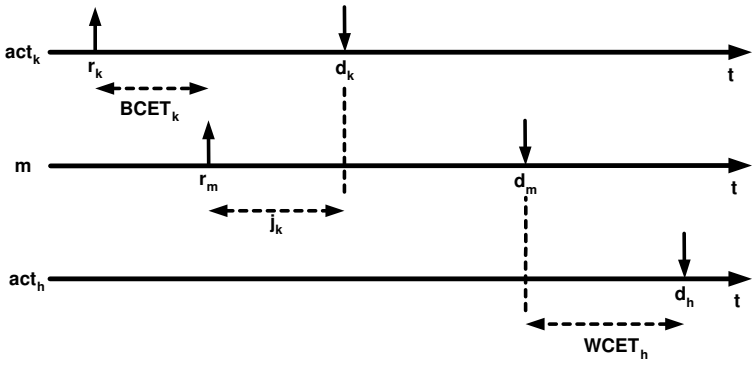

FIG. 5.11 – Caractérisation temporelle du message m.

Nous ré-utilisons les résultats que nous venons de proposer, en considérant l'application distribuée comme un seul réseau de Bloc Fonctionnels. Néanmoins, A chaque fois que deux blocs appartenant à deux dispositifs différents sont en dépendance (un événement de sortie de l'un connecté à un événement d'entrée de l'autre), il va falloir envoyer un message sur le réseau. Nous introduisons artificiellement entre les deux blocs un *bloc fonctionnel virtuel* représentant la transmission sur le réseau. Dans ce bloc, il n'y a qu'un événement d'entrée activant un algorithme *virtuel* dont le *WCET* correspond au pire temps de transmission de ce message sur le réseau non chargé.

Les messages sont alors assimilés à des actions. Nous exploitons alors les résultats du chapitre précédent pour les caractériser temporellement.

Pour valider le comportement temporel de l'application, nous avons à valider la faisabilité des Bloc Fonctionnels dans chaque dispositif ainsi que la validité de la messagerie.

Soit act_k et act_h deux actions en dépendance localisées sur deux dispositifs différents. Nous dénotons par m le message envoyé à act_h une fois l'exécution de act_k terminée.

Dans [Khalgui *et al.*, 2006a], nous caractérisons temporellement ce message de la façon suivante (figure 5.11).

$$\text{m=\{ r, j, d, p\} avec,}$$

* r : la date d'envoi au plus tôt du message. Cette date arrive lorsque l'action act_k fini son exécution au plus tôt.

$$r = r_k + BCET(act_k)$$

* j : la gigue est égale à la différence entre la date d'envoi au plus tard et celle au plus tôt. La date d'envoi au plus tard du message correspond à l'échéance de act_k.

$$j = d_k - r_k$$

* d : c'est l'échéance du message m. Elle correspond à la date au plus tard de l'activation de act_h pour qu'elle respecte son échéance.

$$d = d(act_h) - WCET(act_h)$$

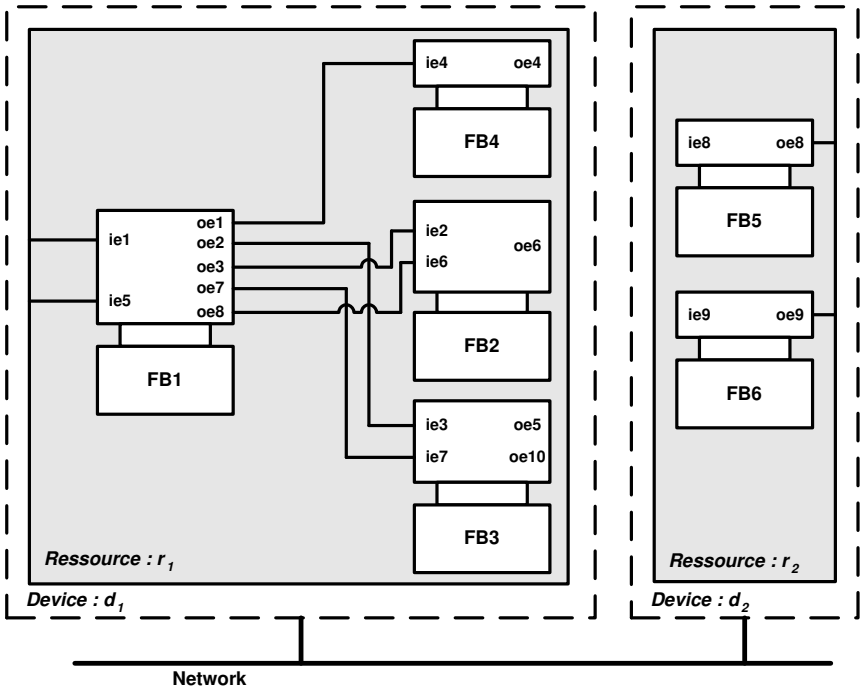

FIG. 5.12 – Une nouvelle distribution de l'application

* p : c'est la périodicité de l'envoi du message m. Sachant qu'on ne considère que des traces non-réentrantes, la période du message est égale à celle de l'action act_k.

Une fois la caractérisation de tous les messages réalisée, nous appliquons la méthode proposée dans [K.M and K.G, 1995] pour construire une messagerie correcte.

Exemple. *Supposons l'application distribuée sur deux dispositifs d_1 et d_2 (figure 5.12).*

Nous caractérisons l'envoi d'un message m entre les deux blocs FB_4 et FB_5 de la façon suivante,

- $r = r_1 + BCET(act_1) + BCET(act_4) = 20$
- $j = d(act_4) - r = 72$
- $d = d(act_8) - WCET(act_8) = 92$
- $p = p(act_1) = 100$

Grâce à cette caractérisation, nous construisons une messagerie correcte sur le réseau. Les messages échangés respectent toutes les bornes sur les temps de réponse considérées dans le cahier des charges.

5.6 Conclusion

Dans ce chapitre, nous avons présenté une approche déployant une application de contrôle, à base de Bloc Fonctionnels, dans une architecture physique distribuée. Cette application doit respecter des bornes sur les temps de réponse de bout en bout. En prenant en compte la nouvelle technologie des contrôleurs, nous avons considéré chaque dispositif à multi-ressources comme un PLC multi-tâches.

Dans un premier temps, nous supposons l'application centralisée dans un seul dispositif multi-ressources. Le problème est donc de déployer ses différents Bloc Fonctionnels dans des tâches OS du PLC correspondant. Nous proposons une approche hybride, combinant une politique préemptive en-ligne et une autre politique non-préemptive hors-ligne pour construire le déploiement. Cette approche génère une tâche récurrente à chaque ressource à partir de son pré-ordonnancement.

Dans un deuxième temps, nous avons considéré la distribution de l'application sur plusieurs dispositifs connectés par un bus CAN. Pour déployer l'application dans chaque dispositif, nous appliquons l'approche hybride construisant les tâches OS correspondantes. De plus, pour construire une messagerie correcte sur le réseau, nous proposons une caractérisation temporelle des messages échangés entre dispositifs. En validant le comportement temporel dans chaque dispositif ainsi que sur le réseau, nous déduisons la correction temporelle de toute l'application.

Chapitre 6

Allocation de Blocs Fonctionnels sur une architecture distribuée

6.1 Introduction

Dans le chapitre précédent, nous avons présenté l'approche hybride déployant une application de contrôle déjà allouée (par hypothèse) sur un support d'exécution distribué. Nous nous intéressons, dans ce chapitre, à la définition même d'une telle allocation [Khalgui and Rebeuf, 2006]. Pour cela, nous supposons donné en entrée,

* Une application de contrôle industriel décrite sous la forme d'un réseau de blocs fonctionnels (qui n'est pas encore alloué sur un support d'exécution). Notons que [Thramboulidis, 2005b] propose une approche concevant un tel réseau à partir de diagrammes UML.
* Un support d'exécution distribué. Nous continuons à considérer un bus CAN connectant les différents calculateurs considérés [network, 1993].

En considérant ces données, le problème à résoudre est de trouver une allocation possible de l'application sur un tel support d'exécution. Cette allocation doit tenir compte :

* des contraintes temporelles (bornes sur les temps de réponses de bout en bout) entre l'acquisition *périodique* de données sur les capteurs et l'activation des actionneurs correspondants.
* des contraintes fonctionnelles décrites dans le cahier des charges. Dans ce chapitre, nous nous limitons à deux types de ces contraintes :
  ** les contraintes de *localisation* fixant, pour chaque bloc fonctionnel de l'application, les dispositifs capables de l'exécuter.
  ** les contraintes *d'exclusion* définissant les groupes de blocs dont l'exécution est en exclusion mutuelle.

Nous considérons qu'une allocation de l'application est correcte si la distribution et le déploiement de ses blocs fonctionnels sur le support d'exécution satisfont les différentes contraintes fonctionnelles et temporelles. Notons que dans ce chapitre, nous nous intéressons à chercher une allocation faisable mais non optimale. L'optimalité, elle, se réfère à plusieurs critères parfois contradictoires (équilibrage de charge, optimisation de bande passante sur le réseau, etc).

Une première approche consiste à générer et à tester toutes les allocations possibles. La recherche d'une solution par cette méthode est alors NP-complet [Tindell *et al.*, 1992]. Afin de

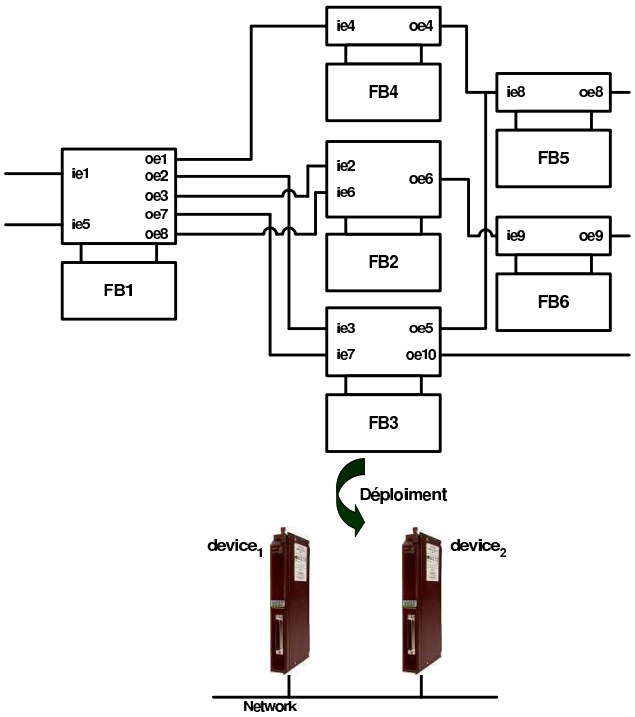

FIG. 6.1 – Exemple de réseau de blocs fonctionnels et d'un support d'exécution

réduire en moyenne le nombre d'allocations étudiées, nous proposons d'utiliser une heuristique donnant des priorités aux différents blocs fonctionnels de l'application. De ce fait, l'allocation d'un bloc particulier n'est possible que si les blocs de priorités supérieures sont *correctement* alloués. De plus, si l'allocation d'un tel bloc n'est pas possible, nous appliquons un back-tracking pour rechercher une autre allocation des blocs déjà alloués (de priorités supérieures). La technique de back-tracking permet d'envisager tous les cas possibles. Cependant, l'heuristique doit permettre de guider l'exploration pour aboutir à une solution faisable plus rapidement.

Dans la section suivante, nous présentons les travaux existants sur l'allocation de tâches temps-réel. Nous définissons dans la section d'après, les différentes contraintes à considérer sur le support d'exécution et aussi sur les blocs fonctionnels. Finalement, nous présentons la méthode que nous proposons pour l'allocation d'une application de contrôle sur un support d'exécution tout en satisfaisant les différentes contraintes temporelles et fonctionnelles.

Exemple. *Dans la figure 6.1, nous reprenons partiellement l'exemple présenté précédemment. Nous supposons que le réseau de blocs fonctionnels n'est pas déjà déployé sur le support d'exécution. De plus, nous supposons que ce support est composé de deux PLCs connectés par un bus CAN. D'autre part, nous préservons les bornes sur les temps de réponse de bout en bout considérées précédemment.*

6.2 Travaux existants sur l'allocation

L'allocation de tâches temps réel sur une architecture distribuée a été le sujet de plusieurs travaux de recherches durant les dernières années. Deux types d'allocation ont été étudiés : l'allo-

cation statique réalisée avant l'exécution, et l'allocation dynamique réalisée au cours d'exécution. Dans cette section, nous nous limitons à présenter les travaux en lien direct avec notre travail.

Dans le cadre de tâches temps réel préemptives, le problème d'allocation est NP-complet [Coffman, 1976; Garey and Johnson, 1979; Lenstra and Kan, 1978; Lawler, 1982]. Plusieurs approches ont été proposées pour réduire la complexité du problème [Baker, 1974; Chu, 1980a; Chu and Lan, 1987; French, 1982; Woolsey and Swanson, 1974; Selvakumar and Murthy, 1994; Ali and El-Rewini, 1993; Billionnet *et al.*, 1992; Bokhari, 1993]. Ces approches se basent sur des heuristiques [Ferro *et al.*, 1999; Ramamritham, 1990], des simulations (simulated annealing) [Tindell *et al.*, 1992; Birriello and Miles, 1994] ou des algorithmes génétiques [Ferro *et al.*, 1999; Sandnes, 1996].

Dans [Cambazard *et al.*, 2004], une approche est proposée pour déployer un ensemble de tâches temps réel sur une architecture distribuée. En plus des contraintes temporelles décrites dans le cahier des charges, les contraintes suivantes ont été prises en compte pour l'allocation de chaque tâche T de l'application,

- Contrainte de *résidence* : définit l'ensemble des calculateurs pouvant exécuter T.
- Contrainte de *co-résidence* : définit l'ensemble des tâches à exécuter avec T sur un même calculateur. Cela se rapproche des contraintes d'exclusion considérées dans ce manuscrit.
- Contrainte d'*exclusion* : définit l'ensemble des tâches à ne pas allouer sur le même calculateur exécutant T. Le terme exclusion dans ce travail ne désigne pas celui utilisé dans ce manuscrit.

Cette approche sépare la fonction d'allocation de celle d'analyse d'ordonnançabilité des tâches. En effet, à chaque scénario d'allocation, on vérifie la faisabilité des tâches sur les différents calculateurs de l'architecture matérielle. Les deux fonctions peuvent être appliquées plusieurs fois afin de trouver la solution d'allocation correcte respectant à la fois les contraintes temporelles ainsi que les contraintes d'allocation. Notons enfin que pour réduire le problème d'allocation, les auteurs utilisent la programmation par contraintes [Hladik *et al.*, 2006] permettant l'analyse de toute allocation échouée avant de chercher une autre.

Dans [Grandpierre *et al.*, 1999], une approche est proposée pour allouer un ensemble de tâches sous des contraintes de précédence sur une architecture matérielle. Cette approche est basée sur une heuristique de *Liste* allouant les différentes tâches selon un ordre de priorités bien déterminé [Lui and Corroyer, 1993; Ahmad and Kwok, 1996; Yang and Gerasoulis, 1993].

Dans [Kasahara and Narita, 1984], une autre approche est proposée pour l'allocation de tâches sous contraintes de précédence. Dans ce travail, les heuristiques CP/MISF (Critical Path/Most Immediate Successors First) et DF/IHS (Depth-First/Implicit Heuristic Search) sont utilisées pour trouver un placement des tâches minimisant leur temps d'ordonnancement (makespan).

[Chu, 1980b] propose d'allouer des tâches (temps réel) sous contraintes de précédence sur une architecture distribuée. Le placement de ces tâches a pour objectif la minimisation de la charge (workload) des différents calculateurs ainsi que le trafic sur le réseau de communication.

[Shukla and Agrawal, 1994] propose d'allouer un ensemble de tâches sous contraintes de précédence et soumises à des contraintes temporelles de bout en bout.

Dans [Peng *et al.*, 1997], une autre approche est proposée pour allouer des tâches périodiques sous contraintes de précédence sur une architecture hétérogène. Le problème posé est de minimiser le pire temps de réponse de chaque tâche étudiée.

Dans [Baruah, 2004], le problème d'allocation de tâches temps réel récurrentes est abordé. Ces tâches sont complètement indépendantes à placer sur une architecture hétérogène.

Enfin, dans [Xu, 1993], une approche est proposée pour allouer un ensemble de tâches sous contraintes d'exclusion et de précédence dans une architecture homogène. L'algorithme proposé permet de chercher sur chaque calculateur un ordonnancement non-préemptif des tâches allouées.

À notre connaissance, il n'y a eu aucun travail de recherche sur la *construction* et l'*allocation* simultanée de tâches temps réel dépendantes. Une telle construction doit tenir compte des contraintes fonctionnelles (gestion de l'exclusion mutuelle) et temporelles définies sur l'application. De plus, elle doit tenir compte du nombre maximal autorisé de tâches à construire dans chaque dispositif (contrainte définie par les systèmes d'exploitation).

6.3 Les contraintes d'allocation

Dans cette section, nous regroupons l'ensemble des contraintes portant sur l'allocation. Ces contraintes viennent soit du support d'exécution, soit des caractéristiques du réseau de blocs fonctionnels. Il est nécessaire de respecter ces contraintes pour le bon fonctionnement de l'application. Du point de vue de la construction d'une allocation, ces contraintes permettent de restreindre le nombre de cas possibles, réduisant par là même la combinatoire.

6.3.1 Contraintes sur le support d'exécution

Les contraintes sur le support d'exécution peuvent porter sur les caractéristiques matérielles de chaque dispositif, sur les caractéristiques des systèmes d'exploitation et enfin sur le réseau.

Les caractéristiques matériels de chaque dispositif peuvent être multiples. Nous ne citons ici que les principales :

- *La puissance CPU.* Cette caractéristique influant sur la vitesse de calcul, on peut l'exprimer sous forme d'un facteur d'accélération par rapport à un processeur de référence [Cucu and Goossens, 2006].
- *La capacité mémoire.* Cette caractéristique peut limiter, entre autre, le nombre de blocs fonctionnels pouvant être alloué sur le dispositif. Toutefois, tous les blocs ne requièrent pas les mêmes ressources mémoire. Ainsi, il faudrait donc évaluer aussi l'occupation mémoire pour chaque bloc.

Du point de vue du système d'exploitation, la contrainte principale concerne le nombre de tâches applicatives simultanées. En effet, certains systèmes imposent des limitations dans ce sens [Takada and Sakamura, 1995] (le système OSEK n'accepte que 16 tâches [Osek, 2004]). Nous noterons $max(dev)$ le nombre maximal autorisées de tâches simultanées dans un dispositif dev.

D'autre part, nous caractérisons classiquement le réseau de communication à l'aide de la paramètre $WCTT$ (Worst Case Transmission Time) définissant le pire temps de transmission d'un message échangés entre dispositifs. Enfin, notons que pour des raisons de temps, nous ne considérons que des dispositifs ayant des caractéristiques matérielles homogènes.

Exemple. *Supposons que le support d'exécution possède les caractéristiques suivantes :*

- *Les caractéristiques matérielles (vitesse CPU, mémoire,...) des dispositifs* $device_1$ *et* $device_2$ *sont identiques.*
- *Le système d'exploitation du dispositif* $device_1$ *(resp.* $device_2$*) ne peut accepter que 2 (resp. 3) tâches applicatives.*

6.3.2 Contraintes sur le réseau de blocs fonctionnels

Une analyse fine du réseau des blocs fonctionnels permet de dégager des contraintes sur l'allocation de chaque bloc.

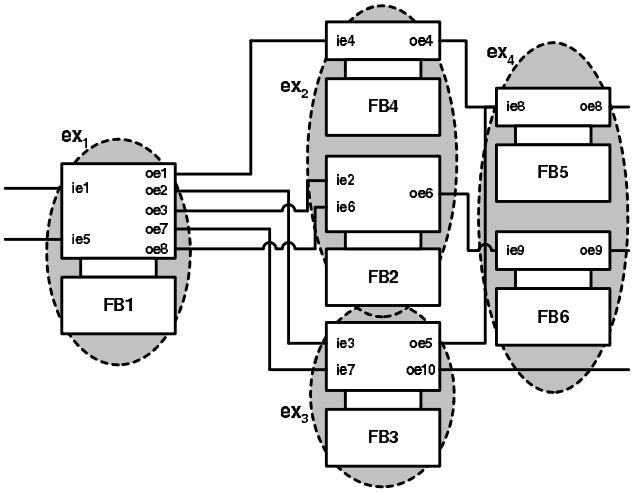

FIG. 6.2 – Exemple de définition des groupes d'exclusion.

Les contraintes d'exclusion

Deux blocs fonctionnels possèdent une contrainte d'exclusion s'ils partagent des données ou la commande de certains procédés physiques [Hladik *et al.*, 2006]. Ces deux blocs doivent donc être exécutés en exclusion mutuelle. Rappelons que du point de vue de la norme IEC61499, ce problème est géré au sein d'une ressource par un ordonnancement statique non préemptif.

Nous définissons *un groupe d'exclusion* comme un ensemble de blocs fonctionnels devant s'exécuter en exclusion mutuelle. Notons qu'un bloc peut appartenir à plusieurs groupes d'exclusion.

Soit $\sigma \in \Sigma$ un ensemble de blocs fonctionnels. Nous définissons $Excl(\sigma)$ l'ensemble des groupes d'exclusion au sein de σ. Cet ensemble $Excl(\sigma)$ est donc un ensemble d'ensembles.

Exemple. *Dans l'exemple précédent, supposons que l'analyse du réseau des blocs fonctionnels fasse apparaître les contraintes suivantes :*

Les blocs FB_2 et FB_4 (resp, FB_5 et FB_6) partagent une variable commune.

Selon les contraintes décrites ci-dessus, nous considérons les groupes d'exclusion suivants (voir figure 6.2) :

$$Excl = \{ex_1, ex_2, ex_3, ex_4\}$$

avec,

- $ex_1 = \{FB_1\}$
- $ex_2 = \{FB_2, FB_4\}$
- $ex_3 = \{FB_3\}$
- $ex_4 = \{FB_5, FB_6\}$

En considérant les contraintes de précédence entre blocs, on définit des contraintes de précédences entre les groupes d'exclusion.

Exemple. *Dans l'exemple, ex_2 et ex_3 dépendent de ex_1.*

Les contraintes de localisation

L'exécution de certains blocs fonctionnels nécessite des accès à des périphériques ou à des services particuliers. L'analyse de ces blocs permet de déduire *des contraintes de localisation* sur

des dispositifs permettant l'accès à ces services ou périphériques [Hladik *et al.*, 2006].

Comme chaque groupe d'exclusion doit être alloué sur une seule ressource, nous définissons les contraintes de localisation directement sur les groupes d'exclusion.

Soit ex un ensemble d'exclusion, notons $local(ex)$ l'ensemble des dispositifs pouvant offrir les accès aux services et périphériques nécessaires.

Exemple. *Dans le même exemple, nous considérons les contraintes suivantes de localisation :*

Les blocs FB_5 et FB_6 accèdent à un périphérique particulier. Seul le dispositif $device_2$, connecté directement à ce périphérique, peut y accéder.

Nous en déduisons la définition local :

$$local(e_1) = local(e_2) = local(e_3) = \{device_1, device_2\}$$

$$local(e_4) = \{device_2\}$$

6.4 Méthode d'allocation

Nous présentons ici la méthode d'allocation proposée [Khalgui and Rebeuf, 2006]. Elle propose de construire une allocation et de vérifier sa validité grâce à l'approche hybride présentée dans le chapitre précédent. Si l'allocation d'un groupe d'exclusion n'est pas possible, l'algorithme fait des retours arrières (back-tracking) pour re-déployer les blocs fonctionnels correspondants sur un autre dispositif.

Afin de réduire l'exploration des allocations possibles, cette méthode se base sur l'utilisation d'heuristiques.

Nous présentons d'abord l'heuristique retenue puis le principe de la méthode. Nous formalisons ensuite la méthode. Finalement, un algorithme présentant la méthode est proposé.

6.4.1 heuristique d'allocation

Comme décrit précédemment, le problème d'allocation des blocs fonctionnel se ramène à une allocation des groupes d'exclusion. Comme ce problème est NP-complet [Tindell, 1994], nous proposons d'utiliser l'heuristique de liste. Cette heuristique attribut un niveau de priorité à chaque groupe d'exclusion. Un groupe n'étant déployé que si tous les groupes de priorités supérieures le sont déjà.

Plusieurs sous-classes de cette heuristique existent. Nous proposons d'utiliser l'algorithme statique HLFET (Highest Level First with Estimated Times) appartenant à la sous-classe BNP (Bounded Number of Processors) [Kwok and Ahmad, 1999].

Cet algorithme utilise la relation de causalité (i.e. de dépendance) pour calculer une priorité à chaque groupe d'exclusion. La priorité d'un groupe correspond à la longueur du plus long chemin entre ce dernier et un groupe sans successeurs.

Dans notre cas, il faut adapter cette heuristique. En effet, tous les groupes ne nécessitent pas les mêmes ressources en temps de calcul. Nous proposons donc de pondérer ce calcul du plus long chemin en associant un poids à chaque groupe successeur correspondant à la somme des WCETs des actions de ses blocs. Notons que cette modération est une approximation car toutes les actions n'auront pas forcément des instances prises en compte dans l'ordonnancement (i.e. toutes ces actions n'appartiennent pas forcément à une opération).

Ici, nous considérons une attribution statique des priorités aux groupes d'exclusion. Dans la suite, nous dénoterons par $HLFET(ex, Excl(\sigma))$ avec $ex \in Excl(\sigma))$ la fonction attribuant une priorité au groupe d'exclusion ex.

6.4.2 Le principe de la méthode d'allocation

Nous présentons les différentes étapes de l'approche d'allocation.

Nous considérons une application de contrôle (constituée d'un ensemble de blocs fonctionnels Σ) à allouer sur un réseau de dispositifs multi-tâches.

étape préliminaire. Analyse des contraintes

L'analyse des blocs fonctionnels de Σ permet de construire l'ensemble des groupes d'exclusion $Excl(\Sigma)$ ainsi que la fonction *local* définissant les contraintes de localisation.

Nous utilisons l'heuristique HLFET pour ordonner l'ensemble des groupes d'exclusion.

$$\forall\ ex_i, ex_j\ \in [1..card(Excl(\sigma)],\ i < j \Leftrightarrow HLFET(ex_i, Excl(\sigma)) < HLFET(ex_j, Excl(\sigma))$$

Nous appliquons récursivement les étapes qui suivent pour allouer chaque groupe d'exclusion par ordre de priorité. Cela correspond à un parcours en profondeur d'abord de l'arbre de possibilités. Notons ex_i le groupe d'exclusion de plus forte priorité parmi ceux restant à allouer.

Etape 1. Choix d'une ressource cible

Nous construisons l'ensemble $AllocRes_i$ des ressources pouvant accueillir le groupe ex_i. La construction de cet ensemble se fait comme suit :

- Pour chaque ressource $R \in dev$, *avec* $dev \in local(ex_i)$, on teste la faisabilité de l'ordonnancement statique pour l'ensemble des blocs appartenant déjà à R enrichi de ceux inclus dans ex_i. Si l'ordonnancement est faisable, alors R est un candidat potentiel pour l'allocation. On rajoute R à $AllocRes_i$.
- Pour chaque dispositif $dev \in local(ex_i)$, si le nombre de ressources (i.e. de tâche OS) est inférieur à $max(dev)$, alors on créé une nouvelle ressource vide qui devient un candidat potentiel pour l'allocation (on l'ajoute à $AllocRes_i$).

Une fois construit l'ensemble des ressources candidates construit, il faut trouver une allocation de ex_i tel que l'ensemble des groupes déjà alloué plus ex_i soient faisable. Pour cela, pour chaque ressource appartenant à $AllocRes_i$, on teste la faisabilité de l'ensemble de l'allocation (cf. *étape 2*) :

- **si** l'allocation est faisable, alors
  - **si** il reste encore des groupes à placer, on rappelle l'étape1 au rang $i + 1$.
  - sinon, **on a trouvé une allocation valide**. La méthode s'arrête.
- Sinon, on teste l'allocation sur la ressource suivante dans $AllocRes_i$.

Si l'allocation est infaisable, alors il faut remettre en cause les choix d'allocation précèdent. On effectue donc un retour arrière (cf. *étape 3*).

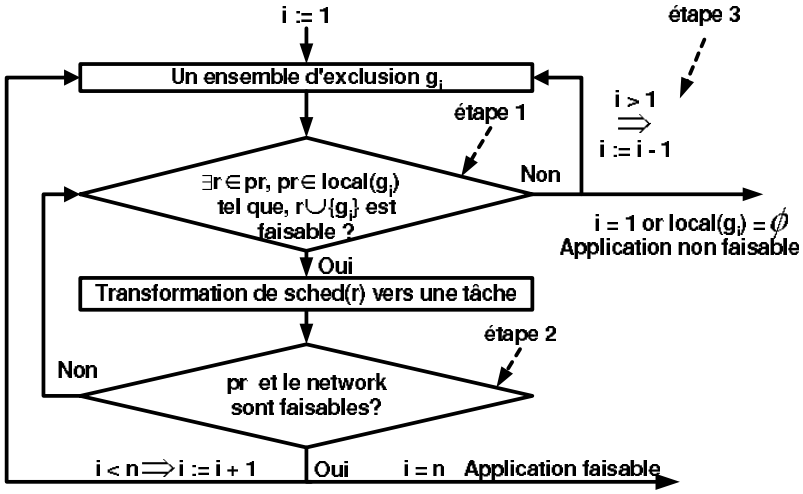

FIG. 6.3 – Les différentes étapes d'allocation d'une application de contrôle

Etape 2. Évaluation de la faisabilité

Cette étape reprend la méthode présentée dans le chapitre 5. Partant de l'allocation des groupes d'exclusions sur les ressources, nous effectuons les étapes suivantes :

- Nous transformons le pré-ordonnancement (incluant des instances d'actions de ex_i) de res vers une tâche récurrente.
- Nous construisons la messagerie du réseau.
- Sur chaque dispositif, nous validons l'ordonnancement dynamique des tâches récurrentes.

Le résultat de cette étape est donc un verdict.

Etape 3. Le retour arrière

Cette étape consiste à remettre en cause l'allocation des groupes de priorités supérieures.

Si $i = 1$ alors on ne peut remettre en cause des allocations précédentes. La méthode s'arrête et **n'a pas trouvé d'allocation faisable**.

Sinon, on retire ex_{i-1} de l'allocation et on rappelle *l'étape 1* au rang $i - 1$.

Notons que l'on peut optimiser cette étape. En effet, si $local(ex_{i-1}) \cap local(ex_i) = \emptyset$, alors la ré-allocation de ex_{i-1} ne permettra pas de résoudre le problème pour ex_i. Dans ce cas, on peut directement retirer aussi ex_{i-2} de l'allocation et ainsi de suite jusqu'à trouver un ex_j tel que $local(ex_i) \cap local(ex_j) \neq \emptyset$

La figure 6.3 présente les différentes étapes à suivre pour allouer les n groupes d'exclusion de l'ensemble $Excl(\sigma)$.

6.4.3 Formalisation

Nous proposons une formalisation de l'approche de l'allocation d'une application de contrôle.

Soit la fonction $alloc(ex_i)$ allouant un groupe d'exclusion ex_i dans un dispositif $dev \in local(ex_i)$. Si une telle allocation est possible, alors la fonction retourne une $vrai$.

Nous supposons les fonctions suivantes déjà définies :

- $stat\_feasible(r \cup \{ex_i\})$. Cette fonction analyse l'ordonnançabilité statique de la ressource r en prenant en compte les blocs fonctionnels du groupe ex_i.
- $add(r, ex_i)$ (resp. $remove(r, ex_i)$). Cette fonction ajoute (resp. supprime) les BFs du groupe ex_i dans (resp. de) la ressource r. cette fonction ajoute aussi les BFs d'interface assurant la communication avec l'extérieur de r.
- $card(dev)$. Cette fonction fournit le nombre de ressources déjà construites dans le dispositif dev.
- $new(r, dev)$. Cette fonction crée une nouvelle ressource r dans le dispositif dev.
- $feasible\_dev(dev)$. Cette fonction analyse l'ordonnançabilité des tâches OS déjà construites dans le dispositif dev.
- $feasible\_net(net)$. Cette fonction analyse la faisabilité des messages échangés sur le réseau entre les groupes déjà alloués.

Nous détaillons la fonction $alloc(ex_i)$ comme suit.

Étape 1.

Pour chaque $dev \in local(ex_i)$

Si $stat\_feasible(ex_i) = faux$ dans dev

Alors $local(ex_i) = local(ex_i) \setminus \{dev\}$;

Commentaire. Si les blocs fonctionnels du groupe ex_i ne sont pas ordonnançables dans dev, alors ce dispositif ne fait plus partie de l'ensemble $local(ex_i)$.

Sinon Si $\exists r \in Ressource(dev) telque stat\_feasible(r \cup \{ex_i\}) = vrai)$

Alors $add(r, ex_i)$; Appliquer(**Étape2**) ;

Commentaire. Si nous trouvons une ressource pouvant contenir les blocs fonctionnels du groupe ex_i, alors nous devons à appliquer l'étape 2 de l'approche.

Sinon Si $card(dev) < max(dev)$

Alors $new(r, dev); add(r, ex_i)$; Appliquer(**Étape2**) ;

Commentaire. Nous n'avons pas pu trouver une ressource pouvant contenir les BFs de ex_i, mais il est possible d'en construire une nouvelle.

Si $\nexists dev \in local(ex_i)$

Commentaire. Il n'existe pas de dispositif dans $local(ex_i)$ pour allouer ex_i.

Alors aucun placement n'est possible pour les différents groupes d'exclusion. L'application ne peut être allouée.

Sinon si $i > 0$

Alors $alloc(ex_{i-1})$;

Sinon, l'application n'est pas déployable.

Commentaire. Nous appliquons un retour arrière si l'allocation de ex_i n'est pas possible.

Étape 2.

Si $feasible\_dev(dev)$ et $feasible\_net(net)$

Alors Appliquer(**Étape 3**) ;

Sinon Appliquer(**Étape 1**) ;

Commentaire. Nous vérifions la faisabilité en ligne des différentes tâches OS construites et des messages échangés sur le réseau. Si l'allocation de ex_i n'est pas faisable, alors nous appliquons l'étape 1 pour le ré-allouer dans une autre ressource.

Étape 3.

Si $i = n - 1$

Alors l'application est correctement allouée.

Sinon $alloc(ex_{i+1})$

Si $\exists ex_j \in Excltelquej > ietalloc(ex_j) = faux$

Alors $remove(ex_i, r)$

Si $dev \notin local(ex_j)$

Alors $alloc(ex_{i-1})$

Commentaire. La ré-allocation de ex_i n'a aucun effet sur celle de ex_j. Nous lançons alors le re-allocation de ex_{i-1}.

Sinon Appliquer(Étape 1) ;

Commentaire. Nous re-déployons le groupe ex_i en excluant la ressource r de tout test de allocation.

End $alloc(ex_i)$.

6.4.4 Algorithme

Nous présentons l'algorithme effectuant les allocations d'une application de contrôle selon le standard IEC 61499 [Khalgui and Rebeuf, 2006]. Cet algorithme est basé sur deux fonctions récursives allouant tous les groupes d'exclusion de l'ensemble $Excl$.

La fonction $alloc()$ permet l'allocation d'un groupe d'exclusion de $Excl$. Les paramètres d'entrée de la fonction sont les suivants :

- ex : un nouveau groupe à allouer dans un des dispositifs de $local(ex)$.
- $failed$: un groupe de priorité inférieure à ex et qu'on échoue à le déployer.

Pour chaque dispositif dev de $local(ex)$, la fonction teste l'allocation de ce groupe en lançant la fonction $integration()$.

Si l'allocation de ex n'est pas possible, alors un retour arrière doit être appliquer pour ré-allouer des groupes déjà alloués. Sinon, l'allocation du groupe de priorité juste inférieure est lancée.

Enfin, si l'allocation d'un groupe $failed$ de priorité inférieure n'est pas possible, alors nous essayons de ré-allouer le groupe ex en excluant la ressource res de tout test d'allocation dans la suite (ajouter res à un ensemble $excluded$).

D'autre part, la fonction $integration()$ recherche une ressource res capable de contenir les BFs d'un groupe d'exclusion ex. Si une telle ressource existe, alors une analyse d'ordonnançabilité est appliquée pour vérifier les contraintes temporelles dans le dispositif dev et aussi sur le réseau.

En se basant sur le calcul de complexité dans [Kwok and Ahmad, 1999], nous notons que la complexité de cette approche est en $O(n^2)$ où n est le nombre de groupes d'exclusion à déployer. En ce qui concerne la construction des tâches OS, la complexité a été calculée dans les chapitres 4.

```
booléen alloc(ex : exclu_set, failed : exclu_set)
début
place : bool ;
res : ressource ; excluded : ressources_set ;
dev : device ;
   place ← false ;
   excluded ← NULL ;
   res ← NULL ;
   while dev ∈ local(ex)
      if(integration(ex, dev, res, excluded)
      //res a ressource able to contain ex.
         then if(ex.suiv)
            then failed ← NULL ;
            place ← alloc(ex.suiv, failed) ;
            if(place = vrai)
               then return(place) ;
            else remove(res, ex) ;
               if(dev ∉ local(failed))
               then return(place) ;
               else add(res, excluded) ;
         else return(true) ;
   failed ← ex ; return(place) ;
fin.
booléen integration(ex : exclu_set, dev : device, res : ressource, excluded : ressources_set)
début
   cond : bool ;
   if(feasible(ex,dev))
   //verify the feasibility of ex in dev.
      then cond ← false ;
      while(res ∈ ressource(dev) \ excluded and not(cond))
         if(feasible(ex, res))
            then cond ← true ; add(ex,res) ;
      if(cond = false and nbre_res(dev) < max(dev))
         then new(res) ; add(ex, res) ; cond ← true ;
      if(online_feas(dev) and network_feas(net, ex) and cond)
         then return(true) ;
      else return(false) ;
   else local(ex) ← local(ex) \ dev ; return(false) ;
fin.
```

Exemple. *Dans l'exemple considéré, nous appliquons l'algorithme proposé pour allouer les différents groupes d'exclusion sur l'architecture matérielle.*

En appliquant l'approche hybride, nous analysons la faisabilité des groupes d'exclusion {ex_1, ex_2, ex_3} dans $device_1$. Les BFs FB_1, ..., FB_4 sont faisables dans ce contrôleur. De ce fait, le allocation de ces groupes est correcte.

De plus, nous analysons la faisabilité des BFs FB_5 et FB_6 dans le contrôleur PLC_2. Nous concluons le respect des différentes bornes sur les temps de réponses et de ce fait nous concluons la correction du allocation.

Enfin, nous analysons la faisabilité du trafic sur le réseau ce qui valide complètement le

allocation de toute l'application sur l'architecture matérielle.

6.5 Conclusion

Dans ce chapitre, Nous avons présenté une méthode d'allocation des bloc fonctionnels d'une application sur un réseau de dispositifs. Cette allocation tient en compte des contraintes temporelles, fonctionnelles et opérationnelles relatives soit à l'application elle-même, soit au support d'exécution.

Ces contraintes permettent de réduire l'exploration des allocations possibles. Toutefois, le problème restant NP-complet, nous avons proposé d'utiliser une heuristique pour restreindre l'espace des solutions possibles. Notons qu'en suivant cette heuristique, nous risquons de ne pas trouver d'allocation faisable alors qu'il peut en exister une (mais qui n'est pas compatible avec l'heuristique).

La méthode construit donc une allocation en distribuant les groupes d'exclusion par ordre de priorité. En utilisant la méthode du chapitre précédent, nous construisons les tâches OS, la messagerie et évaluons la faisabilité de l'ensemble. En cas de non-faisabilité, nous opérons un retour arrière (back tracking) afin de remettre en cause les choix d'allocation des groupes de priorité supérieure.

Chapitre 7

Étude de cas : Une application de contrôle d'une chaîne de production

7.1 Introduction

Dans ce chapitre, nous considérons comme étude de cas une application de contrôle industriel à base de blocs fonctionnels. Cette application, contrôlant une chaîne de production de pièces en plastique, est extraite de [Lewis, 2002]. Plusieurs laboratoires de recherches, sur les systèmes de contrôle, utilisent cette application pour évaluer leurs contributions [Thramboulidis, 2005b].

Dans cette étude, nous prenons en compte l'interprétation de la norme proposée dans le chapitre 3. Nous étendons en particulier l'étude proposée dans [Lewis, 2002], en enrichissant la distribution de l'application à l'aide du concept de ressource. De plus, nous donnons à certains blocs fonctionnels un comportement conditionnel pour rendre le problème plus complexe. L'application doit être déployée sur plusieurs ressources de dispositifs qui sont ici des automates programmables multi-tâches connectés sur un bus CAN.

Nous présentons dans la section suivante le principe général de l'application. Nous introduisons, en particulier, les différents blocs fonctionnels la composant. Pour alléger les calculs, nous nous limitons à étudier les blocs fonctionnels composés sans détailler les blocs élémentaires. Pour analyser la faisabilité d'un déploiement, nous présentons le comportement de l'application en prenant en compte son architecture fonctionnelle. Enfin, nous présentons dans cette section les différentes contraintes décrites dans le cahier des charges.

Dans la section 3, nous supposons que l'application est déjà distribuée sur des dispositifs multi-ressources. Nous appliquons les contributions des chapitres 4 et 5 pour construire un déploiement faisable de ses différents blocs fonctionnels. Nous définissons en particulier les tâches OS implantant l'application sur chaque automate programmable.

Nous supposons, dans la section 4, que l'application n'est pas déjà allouée sur un support d'exécution. En considérant les différentes contraintes du cahier des charges, nous appliquons l'approche par heuristique pour allouer les différents blocs fonctionnels aux contrôleurs.

7.2 Principe de l'application

Considérons une chaîne de production de pièces en plastique de différentes formes géométriques (figure 7.1). Cette chaîne, contenant un tapis roulant, transporte les pièces entre différents

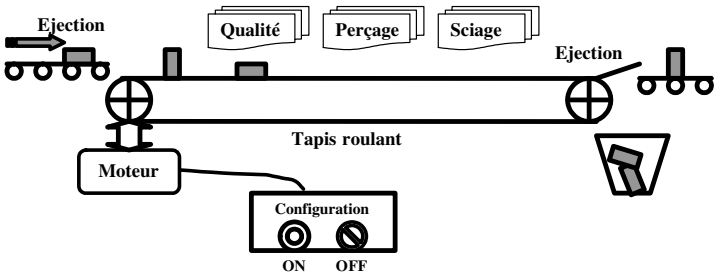

FIG. 7.1 – La chaîne de production

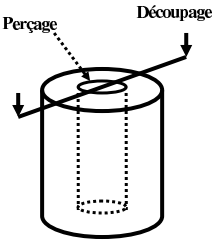

FIG. 7.2 – Traitement d'une pièce cylindrique.

postes de production. Dans cet exemple, nous nous limitons aux postes de contrôle de qualité, de perçage et de sciage. Selon les spécifications décrites dans le cahier des charges, toute pièce à produire est soumise à des contrôles de qualité, à des opérations de perçage et/ou de sciage.

Dans cette étude de cas, l'application contrôlant tout le cycle de production, est à base de blocs fonctionnels. Ces blocs contrôlent la production des pièces pour en assurer la bonne qualité. Initialement, les pièces considérées sont ou bien cylindriques ou bien cubiques. Nous supposons qu'elles ont subies précédemment à des opérations de surfaçage.

Le tapis roulant de la chaîne est commandé par un panneau de contrôle marche/arrêt. Pour éviter tout accident, nous supposons qu'une pièce, au plus, doit être sur le tapis pendant le fonctionnement. Une fois déposée sur le tapis, toute pièce subit deux contrôles successifs de qualité. Le premier contrôle valide les dimensions de la pièce en prenant en compte son cahier des charges. Le deuxième valide l'opération de surfaçage. Le résultat de cette opération caractérise la qualité de la pièce comme excellente, acceptable ou insuffisante.

Pour chaque pièce cylindrique, une fois la phase de contrôle de qualité finie, deux cas se posent :

- **si** la qualité est excellente, alors la pièce subit une opération de perçage suivie d'une opération de découpage en deux (figure 7.2).
- **Sinon**, la pièce est rejetée.

Pour chaque pièce cubique, une fois la phase de contrôle de qualité finie, trois cas se posent :

- **Si** la qualité est excellente, **alors** la pièce subit 4 opérations de perçage (figure 7.3).
- **Sinon si** la qualité est acceptable, **alors** elle subit une opération de découpage (figure 7.3).
- **Sinon**, la pièce est rejetée.

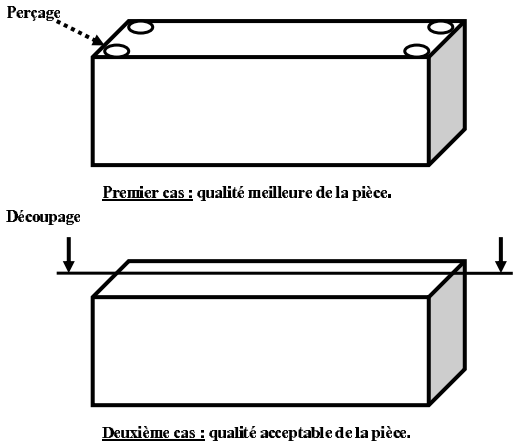

FIG. 7.3 – Les traitements possibles d'une pièce cubique

7.2.1 Présentation des blocs fonctionnels

Dans cette section, nous présentons le réseau des blocs fonctionnels implémentant l'application (figure 7.4). Avant d'analyser son comportement en détail, nous nous basons sur [Lewis, 2002] pour présenter les différents blocs fonctionnels utilisés.

FB_Exec

Ce bloc fonctionnel est une instance du type de bloc *Exec*. Il met en marche tout le système de contrôle en envoyant un événement *Init* et il l'arrête en envoyant *Stop*.

FB_DriveCntl

Ce bloc fonctionnel composé est une instance du type *DriveCntl*. En se basant sur l'état du panneau de configuration fixé par l'utilisateur, il contrôle le fonctionnement du moteur entraînant le tapis roulant.

Lorsque le moteur change de mode de marche (de marche vers arrêt ou inversement), le bloc envoie un événement EXO aux blocs FB_Feed1 et FB_Feed2. Notons que la variable *Running* associée à EXO est un booléen égal à *vrai* lorsque le moteur est en marche. Si *Running* est *vrai*, l'envoi d'un événement EXO lance l'exécution des blocs FB_Feed1 et FB_Feed2. Nous présentons dans la figure 7.5 le comportement interne de ce bloc.

Le bloc fonctionnel FB_Cycle est une instance du type E_Cycle générant un événement EO périodique de période 100 ms. Cet événement permet à l'application de consulter périodiquement le panneau de configuration pour lire l'état des boutons de commande du moteur. Cet accès se fait grâce au bloc fonctionnel d'interface FB_Rd_Panel.

Trois cas peuvent se présenter :

– L'utilisateur décide de mettre en marche le moteur. Dans ce cas, la valeur de la variable booléenne RD_1 est *vrai*. Le bloc fonctionnel $FB\_DemandRun$ informe le bloc fonctionnel $FB\_EnableRun$ que le moteur doit se mettre en marche.

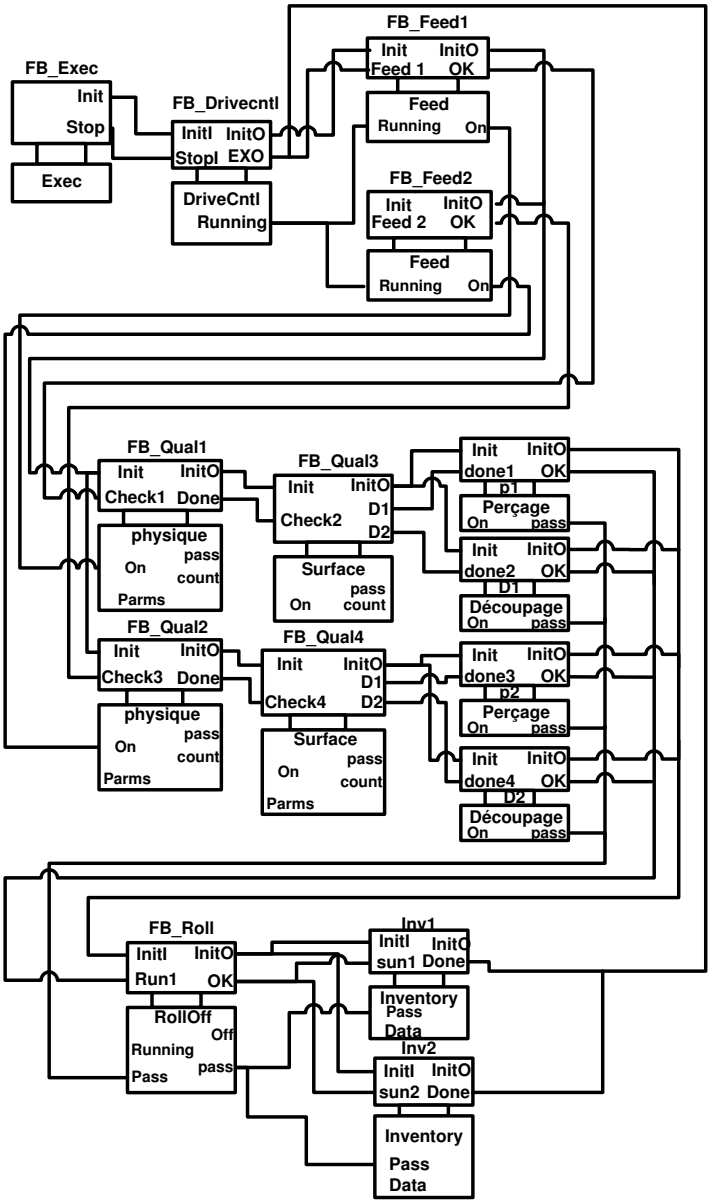

FIG. 7.4 – L'application de contrôle de la chaîne de production

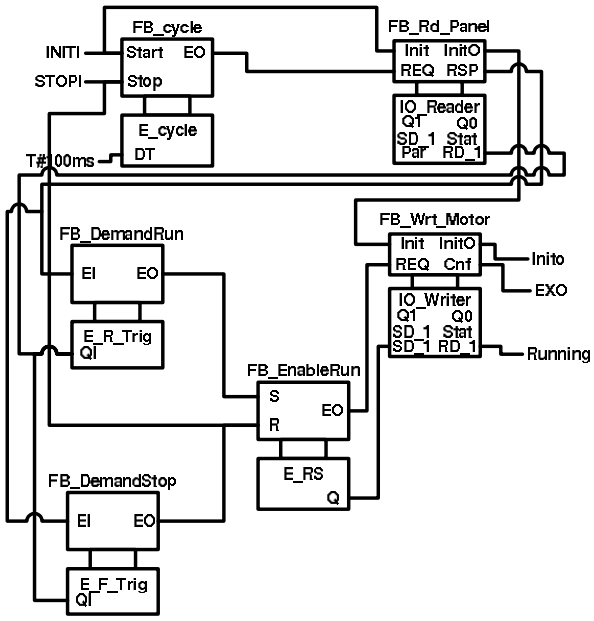

FIG. 7.5 – Le bloc fonctionnel composé Drivecntl

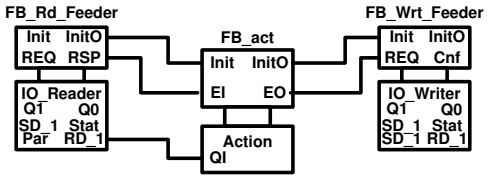

FIG. 7.6 – Le bloc fonctionnel composé FB_feed

- L'utilisateur décide d'arrêter le moteur. Dans ce cas, la variable RD_1 prend la valeur $faux$. Le bloc fonctionnel $FB\_DemandStop$ signale cette information au bloc fonctionnel $FB\_EnableRun$.
- Le bloc fonctionnel FB_Exec envoie à $FB\_EnableRun$ un événement $Stop$ pour arrêter tout le système.

En recevant le nouveau mode désiré, le bloc $FB\_EnableRun$ agit sur le moteur grâce au bloc fonctionnel d'interface FB_Wrt_Motor. Ce bloc change le mode du moteur et confirme ce changement en envoyant l'événement de sortie EXO.

FB_Feed

Ce bloc composé commande un éjecteur de pièce sur le tapis. Une fois activé par $FB\_DriveCntl1$, il vérifie l'existence d'une pièce brute à éjecter à l'aide du bloc fonctionnel d'interface FB_Rd_Feeder. Si cette pièce est présente, il actionne l'éjecteur en utilisant FB_Wrt_Feeder (figure 7.6).

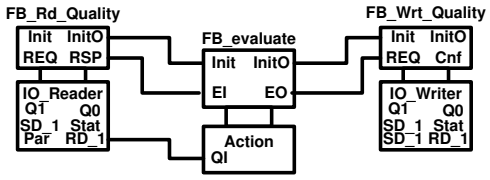

FIG. 7.7 – Le bloc fonctionnel composé *FB_Quality*

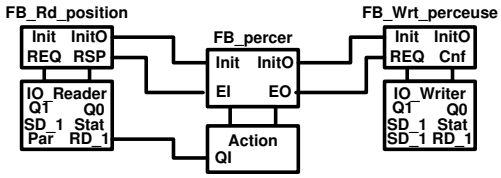

FIG. 7.8 – Le bloc fonctionnel composé *FB_perçage*

FB_Qualphysique

Ce bloc composé valide les dimensions d'une pièce en considérant les spécifications décrites dans le cahier des charges. Il se base sur les blocs fonctionnels d'interface *FB_Rd_Quality* et *FB_Wrt_Quality* pour commander le poste de contrôle de qualité. À la fin du test, il envoie le résultat : excellente, acceptable ou insuffisante selon le contrôle effectué (figure 7.7).

FB_Perçage

Selon le type de la pièce (cylindrique ou cubique), ce bloc composé lance une opération de perçage en commandant la perceuse à l'aide des blocs fonctionnels d'interface *FB_Rd_position* et *FB_Wrt_perceuse* (figure 7.8).

FB_Découpage

De même que *FB_Perçage*, ce bloc composé lance une opération de découpage d'une pièce en commandant la scie à l'aide des blocs fonctionnels d'interface *FB_Rd_position* et *FB_Wrt_scie* (figure 7.9).

FB_Roll

Ce bloc composé est activé à la fin du cycle de production d'une pièce. En fonction des tests de qualité, il se charge de passer la pièce à un autre cycle de production ou de la rejeter. Pour commander l'éjecteur final des pièces, ce bloc se sert des blocs fonctionnels d'interface *FB_Rd_capteur* et *FB_Wrt_éjecteur* (figure 7.10).

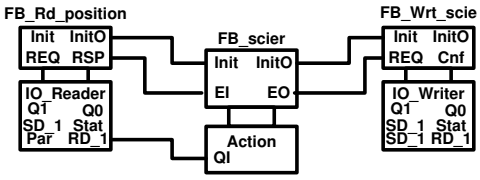

FIG. 7.9 – Le bloc fonctionnel composé $FB\_Découpage$

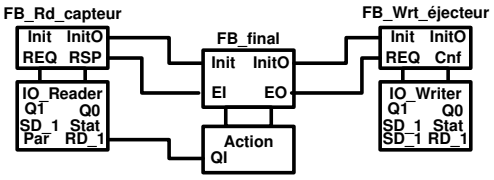

FIG. 7.10 – Le bloc fonctionnel composé $FB\_RollOff$

Inv

Ce bloc fonctionnel représente une interface à une base de données contenant des statistiques sur les pièces produites. Nous précisons que [Lewis, 2002] n'a pas défini en détail ce bloc.

7.2.2 Étude de comportement

Nous détaillons le comportement de l'application considérée. Dès que l'utilisateur met en marche le système de contrôle, tous les blocs de l'application sont initialisés. De plus, le bloc E_cycle de $Drivecntl$ commence à envoyer chaque 100ms un signal EO pour lire l'état du panneau de configuration.

Au moment où l'utilisateur agit sur le panneau de configuration, le bloc $FB\_EnableRun$ met en marche le moteur grâce au bloc d'interface FB_Wrt_Motor. De plus, il signale aux deux blocs FB_Feed1 et FB_Feed2 le lancement de la production des pièces. Le bloc FB_Feed1 (resp. FB_Feed2) agit alors sur l'éjecteur des pièces correspondantes. De plus, il envoie un événement au bloc FB_Qual1 (resp. FB_Qual2) pour commencer le contrôle de qualité. Nous précisons que le contrôle de pièces cylindriques (resp. cubiques), est assuré par les blocs FB_Qual1 et FB_Qual3 (resp. FB_Qual2 et FB_Qual4).

À la fin de l'exécution de FB_Qual3, deux cas se présentent.

- **si** la qualité est excellente, **alors** les blocs $P1$ **ET** $D1$ sont à exécuter pour effectuer le perçage et le découpage de la pièce (cylindrique).
- **Sinon**, Le bloc FB_Roll est appelé pour jeter la pièce.

À la fin de l'exécution de FB_Qual4, trois cas se posent :

- **Si** la qualité est excellente, **alors** le bloc $P2$ est à exécuter pour effectuer le perçage.
- **Sinon si** la qualité est acceptable, **alors** le bloc $D2$ est à exécuter pour effectuer le découpage.
- **Sinon**, Le bloc FB_Roll est appelé pour jeter la pièce.

Une fois les opérations de perçage et de découpage effectuées, le bloc FB_Roll est activé.

En prenant en compte la phase de contrôle de qualité, ce bloc se charge de jeter les pièces ou de les faire passer à un autre cycle de production. De plus, il lance l'exécution des blocs d'interfaces $Inv1$ et $Inv2$.

Une fois l'accès à la BD effectué par le bloc $Inv1$ (resp. $Inv2$), la production d'une nouvelle pièce est lancée.

7.3 Hypothèses et Restrictions

Pour réduire la complexité du problème, nous supposons que tous les blocs composés sont des boîtes noires. De ce fait, nous les traitons sans tenir en compte les blocs les composant.

D'autre part, nous ne nous intéressons pas aux événements d'initialisation ni au cas de rejet de pièces. De plus, nous négligeons les exécutions des Blocs Fonctionnels d'interface. Enfin, nous supposons que les buffers des blocs sont de tailles larges de pour qu'il n'y a pas de perte d'événements.

Nous présentons, dans les paragraphes suivants, les différentes contraintes que devra respecter l'application.

7.3.1 Contraintes du support d'exécution

Nous supposons que l'architecture matérielle est composée de deux calculateurs multi-tâches D_1 et D_2. Ces calculateurs sont connectés par un bus CAN.

7.3.2 Contraintes fonctionnelles

En se basant sur la contribution du chapitre 6, nous distinguons deux types de contraintes fonctionnelles : les contraintes d'exclusion et les contraintes de localisation.

Contraintes d'exclusion

Nous proposons dans la figure 7.11 les différents groupes d'exclusion formant l'application. Ces groupes contiennent des blocs fonctionnels dont l'exécution est en exclusion mutuelle.

À titre d'exemple, dans le groupe ex_4, l'exécution des blocs P_1, D_1, P_2 et D_2 doit être en exclusion mutuelle. Les opérations de perçage et de découpage doivent être faites en exclusion.

Dans la figure 7.12, nous présentons les relations de causalité entre ces groupes. En appliquant l'heuristique de liste (l'algorithme HLFET), nous calculons leurs degrés de priorités.

Les contraintes de localisation

Dans la figure 7.13, nous définissons pour chaque groupe d'exclusion les calculateurs pouvant exécuter ses blocs fonctionnels. Parmi les deux calculateurs disponibles, D_2 est le seul pouvant exécuter les blocs du groupe ex_4. Il offre à ses blocs l'accès à la perceuse et la scie.

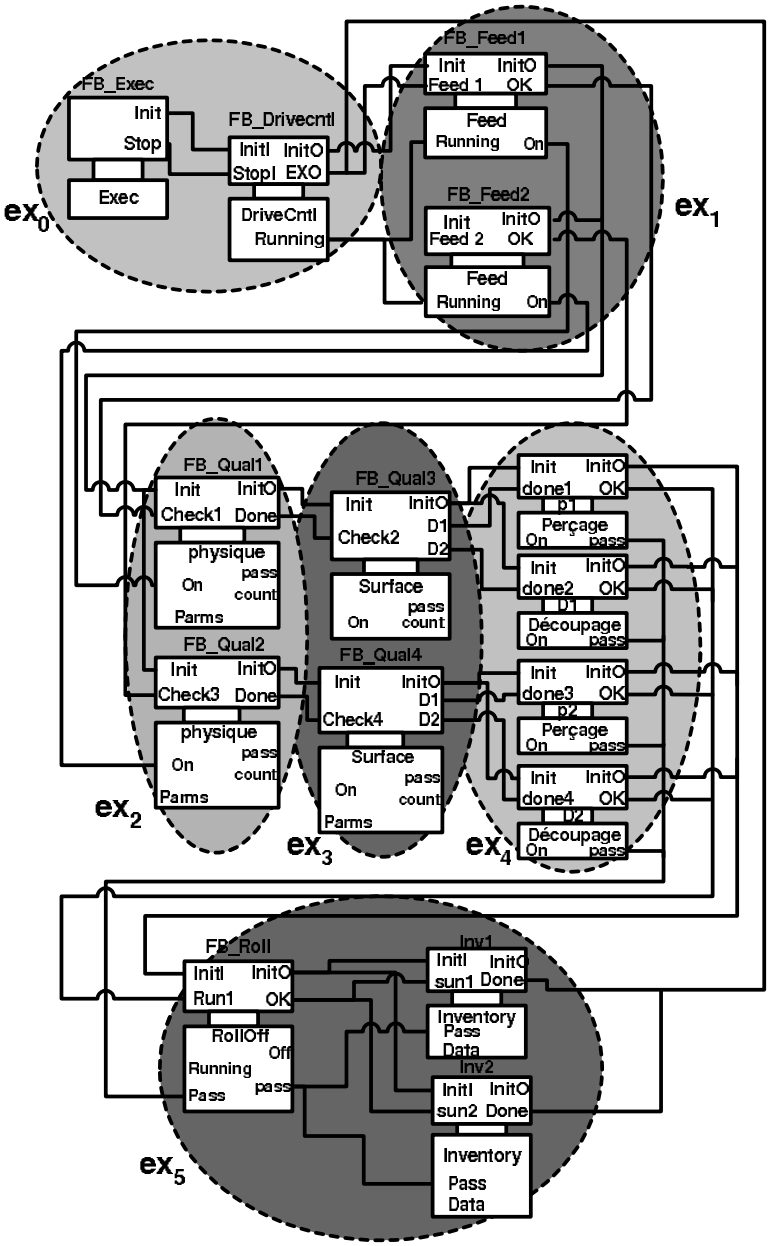

FIG. 7.11 – Les groupes d'exclusion de l'application

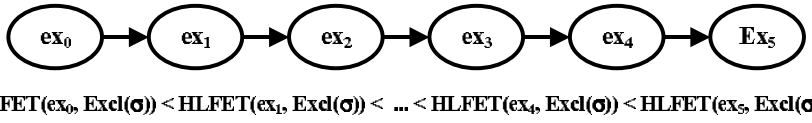

FIG. 7.12 – La relation de causalité entre les différents groupes d'exclusion

Groupe ex	ex_0	ex_1	ex_2	ex_3	ex_4	ex_5
local(ex)	$\{D_1\}$	$\{D_1\}$	$\{D_2\}$	$\{D_2\}$	$\{D_2\}$	$\{D_2\}$

FIG. 7.13 – La localisation des groupes d'exclusion sur l'architecture matérielle

7.3.3 Contraintes temporelles

Dans cette étude de cas, nous considérons seulement un traitement périodique des pièces cylindriques et cubiques. La première pièce cylindrique (resp. cubique) à traiter est à $t = 2$ (resp. $t = 20$). La périodicité de traitement de chaque type de pièce est $p = 100$ (i.e. la durée entre l'éjection de deux pièces consécutives de même type est 100).

D'autre part, nous simplifions le problème en supposant que la borne à respecter pour traiter chaque pièce est $borne = 80$ (Cette contrainte exprime la durée maximale entre l'éjection d'une pièce sur le tapis et sa fin de production).

7.4 Validation temporelle de l'application

Dans un premier temps, nous supposons que l'application est déjà distribuée sur des ressources du support d'exécution (figure 7.14). Nous assurons les communications entre ces ressources à l'aide des blocs de communication Pub_i et Sub_j ($i \in [1, 5], j \in [1, 4]$).

Pour construire les différentes tâches OS, nous analysons dans la section suivante le comportement conditionnel de l'application avant de la transformer vers un système d'actions.

7.4.1 Analyse du comportement

Nous analysons le comportement conditionnel de l'application de la façon suivante :

- Une fois le bloc d'interface $Sub2$ exécuté, l'application traite les deux événements $Feed1$ et $Feed2$. Ces événements peuvent donc être simultanés.
- Une fois le traitement de l'événement $Check_2$ de FB_Qual3 fini, l'application peut traiter les deux événements $done1$ **et** $done2$ des blocs $P1$ et $D1$.
- Une fois le traitement de l'événement $Check_4$ de FB_Qual4 fini, l'application peut traiter les deux événements $done3$ **ou** $done4$ des blocs $P2$ et $D2$.
- Une fois le traitement de l'événement Run_1 de FB_Roll fini, l'application peut traiter les deux événements $sun1$ **et** $sun2$ des blocs $Inv1$ et $Inv2$.

7.4.2 Transformation vers un système d'actions

Pour analyser la faisabilité de l'application dans chaque ressource, nous transformons son réseau de blocs fonctionnels vers un système d'actions sous contraintes de précédence. Dans cette section, nous nous limitons à la caractérisation de certaines actions de ce système.

Dans le bloc $DriveCntl$, l'action act_STOPI est utilisée pour la production des pièces cubiques et aussi cylindriques. De ce fait, nous proposons à la dupliquer afin d'analyser la faisabilité de l'application.

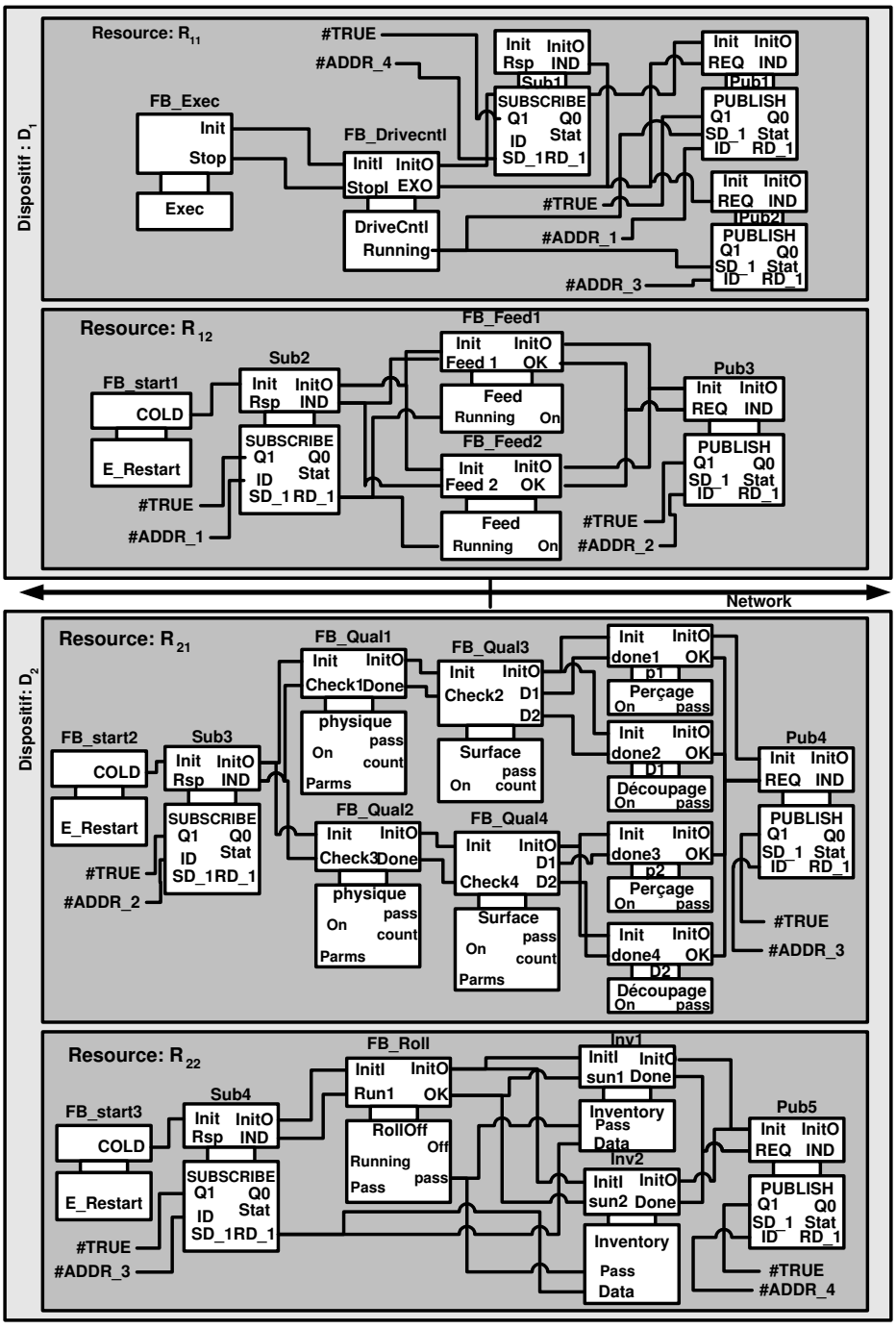

FIG. 7.14 – Une distribution de l'application

Dans le bloc FB_Qual3 (resp. FB_Qual4), l'action act_Check2 (resp. act_Check4) représente l'exécution des algorithmes lorsqu'une occurrence de $Check2$ (resp.$Check4$) arrive :

* Nous caractérisons le prédécesseur et les successeurs de act_Check2 de la façon suivante :
  ** Une fois l'exécution de act_Check1 terminée, l'action est alors activée.

$$pred(act\_Check2) = act\_Check1$$

  ** Une fois l'exécution terminée, les deux actions act_done1 **et** act_done2 sont à activer.

$$succ(act\_Check2) = \{\{act\_done1, act\_done2\}\}$$

* De même, nous caractérisons le prédécesseur et les successeurs de act_Check4 de la façon suivante :
  ** L'action act_Check3 est le prédécesseur de act_Check4.

$$pred(act\_Check4) = act\_Check3$$

  ** Une fois l'exécution de act_Check4 terminée, l'action act_done3 **ou** act_done4 est à activer.

$$succ(act\_Check4) = \{\{act\_done3\}, \{act\_done4\}\}$$

D'autre part, nous définissons les traces d'actions suivantes de l'application,

* Dans la ressource R_{11}, une seule trace d'actions existe. Cette trace contient l'action act_STOPI du bloc $FB\_Drivecntl$.

$$tr_1 = act\_STOPI$$

* Dans la ressource R_{12}, nous distinguons deux traces d'actions,

$$tr_2 = act\_Feed1$$

$$tr_3 = act\_Feed2$$

* Dans la ressource R_{21}, quatre traces d'actions existent,

$$tr_4 : act\_Check1, act\_Check2, act\_done1$$

$$tr_5 : act\_Check1, act\_Check2, act\_done2$$

$$tr_6 : act\_Check3, act\_Check4, act\_done3$$

$$tr_7 : act\_Check3, act\_Check4, act\_done4$$

* Dans la ressource R_{12}, nous distinguons deux traces d'actions,

$$tr_8 : act\_Run1, act\_sun1$$

$$tr_9 : act\_Run1, act\_sun2$$

Enfin, nous nous limitons à caractériser les opérations de l'application dans la ressource R_{21}.

$$op\_Check1 = \{tr_4, tr_5\}$$

$$op\_Check3 = \{tr_6, tr_7\}$$

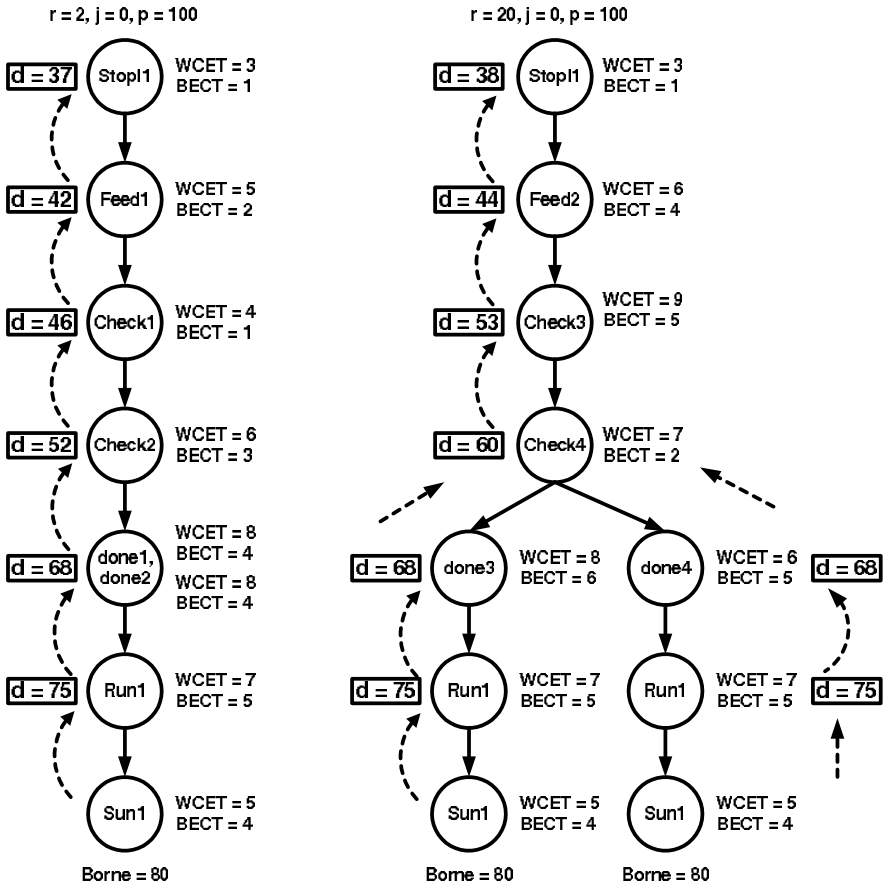

FIG. 7.15 – Le calcul des échéances

7.4.3 Validation hybride

Nous appliquons l'approche hybride pour la validation temporelle de l'application. Dans chaque calculateur, nous calculons les pré-ordonnancements de ses différentes ressources. Ensuite, nous construisons ses tâches OS à partir de ces pré-ordonnancements.

Vu qu'à chaque instant, il y a au plus un seul message à échanger entre calculateurs (de $Pub3$ vers $Sub3$ ou de $Pub5$ vers $Sub1$), nous n'étudions pas donc la messagerie du réseau.

Calcul des échéances

Pour respecter les différentes contraintes temporelles, nous calculons pour chaque action une échéance en prenant en compte celles de ses successeurs (figure 7.15).

Notons que,

$$d(check2) = d(done2) - WCET(done1) - WCET(done2) = 52$$

$$d(check4) = min\{d(done4) - WCET(done4); d(done3) - WCET(done3)\} = 60$$

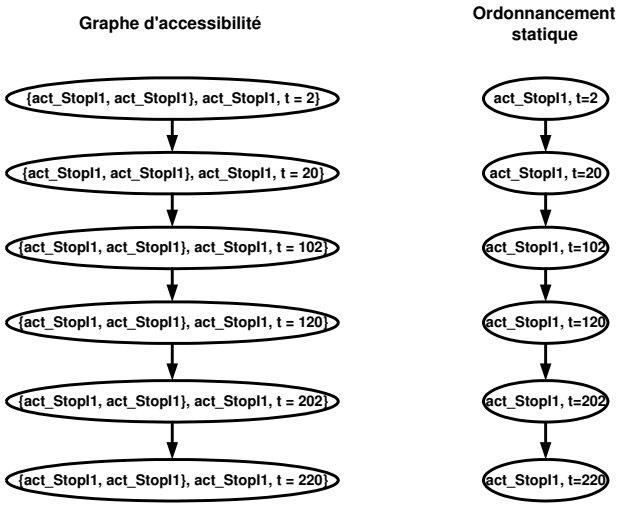

FIG. 7.16 – le graphe d'accessibilité de la ressource R_{11}

Génération de pré-ordonnancements

Pour valider le comportement temporel des blocs fonctionnels dans chaque resource d'un dispositif, nous appliquons l'algorithme de calcul de l'ordonnancement statique en utilisant la construction d'un graphe d'accessibilité. Cette construction est réalisée dans une hyper-période calculée. Dans la ressource R_{11}, cette analyse est effectuée dans l'intervalle [2, 220] (le ppcm est égal à $lcm = 100$).

Nous présentons dans la figure 7.16 (resp. figure 7.17) le graphe d'accessibilité ainsi que le pré-ordonnancement de la ressource R_{11} (resp. R_{12}) du dispositif D_1. Dans la figure 7.18 (resp. figure 7.19), nous présentons le graphe d'accessibilité validant le comportement temporel dans la ressource R_{21} (resp. R_{22}). En particulier, nous créons un état virtuel dans la ressource R_{22} pour séparer le traitement d'une pièce cylindrique de celui d'une pièce cubique. Enfin, nous présentons dans la figure 7.20 (resp. figure 7.21) le pré-ordonnancement de la ressource R_{21} (resp. R_{22}). Dans cet exemple, les calculs conduisent à des ordonnancements statiques faisables pour toutes les ressources.

Construction des tâches OS

Une fois ces ordonnancements statiques calculés, nous construisons la tâche OS correspondant à chaque ressource. Ces tâches sont des séquences ordonnant l'exécution d'instances d'actions.

La figure 7.22 présente les tâches OS implémentant le comportement de l'application dans les ressources R_{11} et R_{12} pendant le régime stationnaire.

La figure 7.23 (resp. figure 7.24) présente la tâche OS implémentant le comportement de l'application dans la ressource R_{21} (resp. R_{22}) pendant le régime stationnaire.

Enfin, nous appliquons la condition d'ordonnançabilité proposée dans [Baruah, 2003] sur une hyper-période calculée. Nous concluons la validation de l'application dans chaque dispositif.

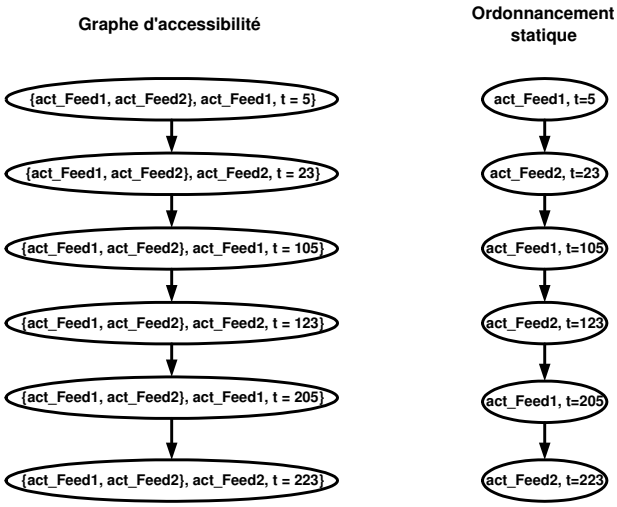

FIG. 7.17 – le graphe d'accessibilité de la ressource R_{12}

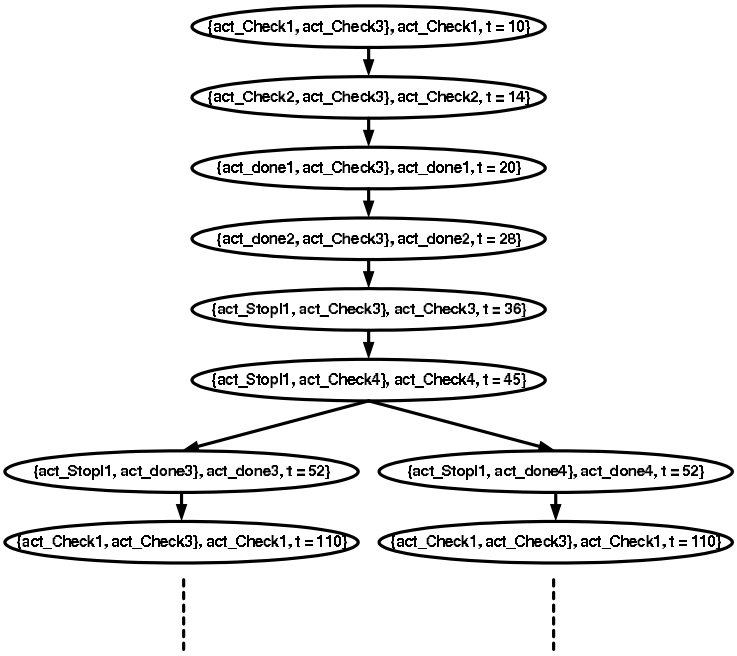

FIG. 7.18 – le graphe d'accessibilité de la ressource R_{21}

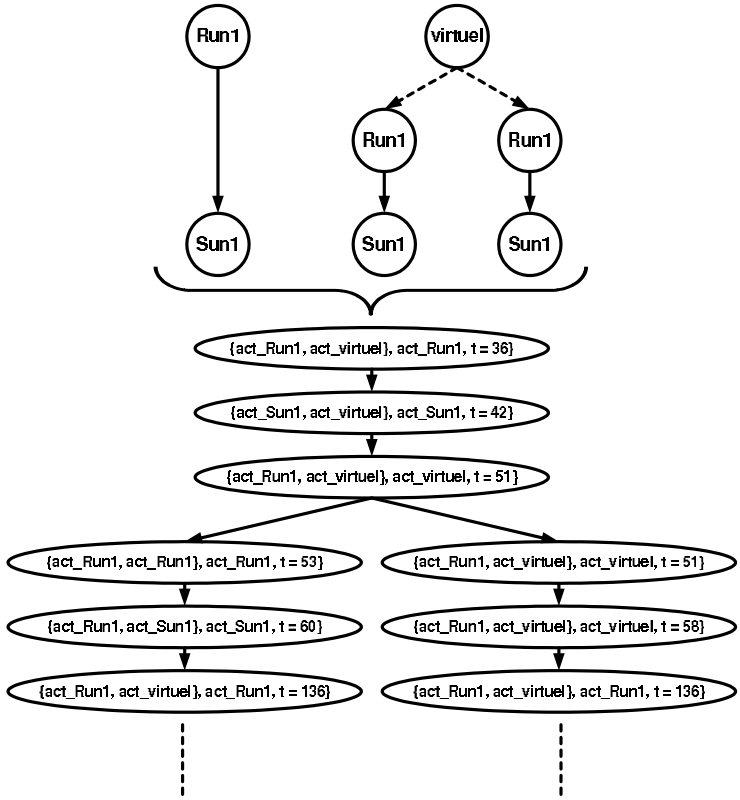

FIG. 7.19 – le graphe d'accessibilité de la ressource R_{22}

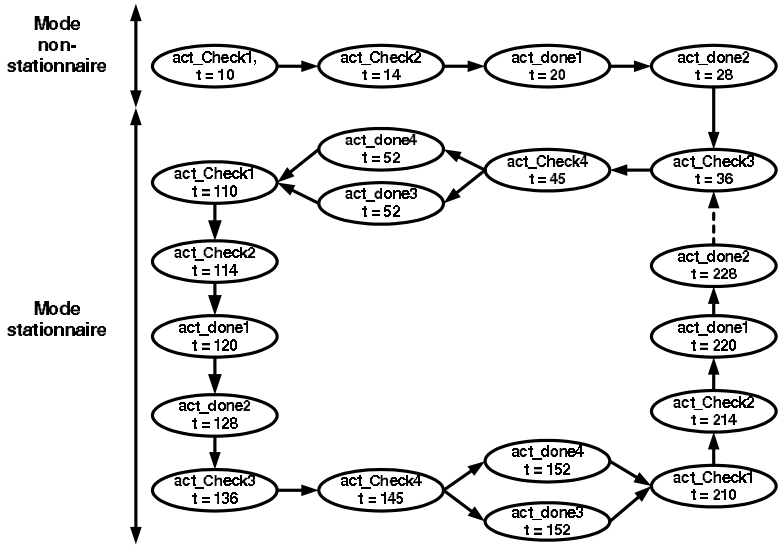

FIG. 7.20 – L'ordonnancement statique de la ressource R_{21}

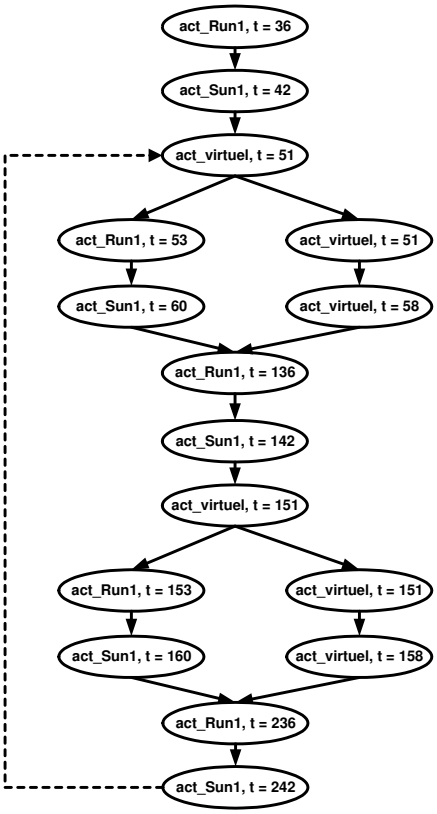

FIG. 7.21 – L'ordonnancement statique de la ressource R_{22}

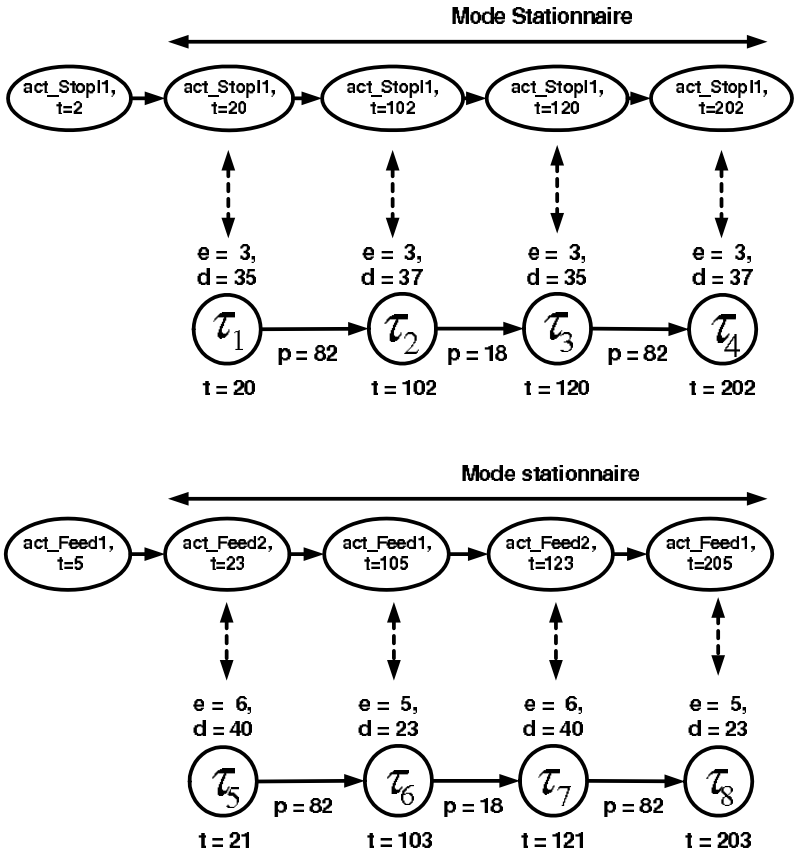

FIG. 7.22 – La construction des tâches OS des ressources R_{11} et R_{12}

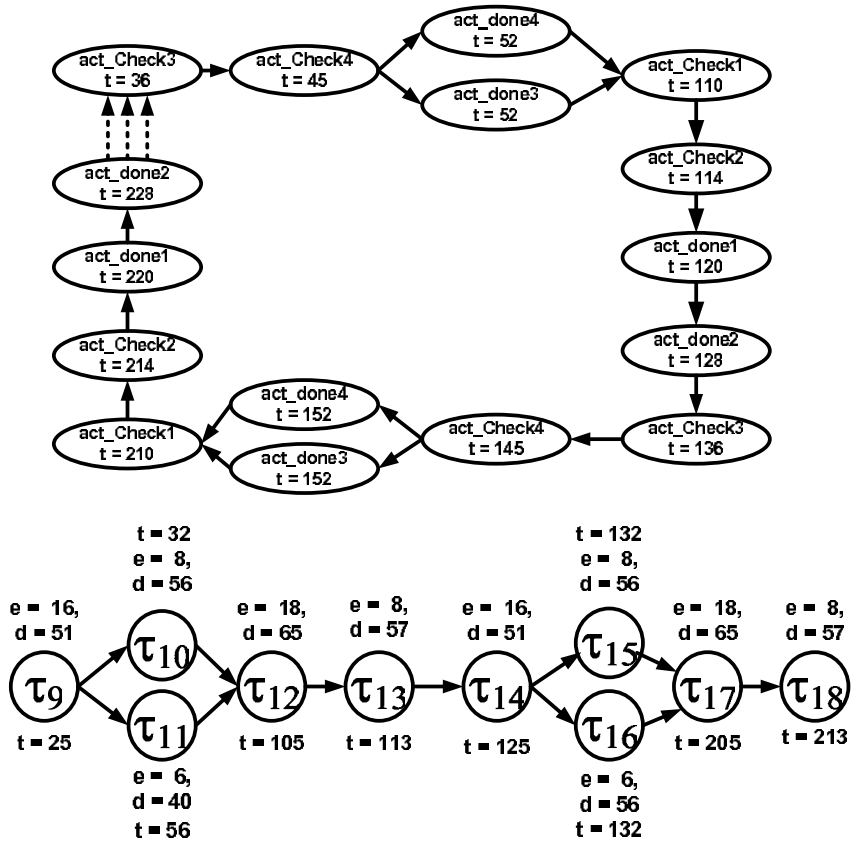

FIG. 7.23 – La construction de la tâche OS de la ressource R_{21}

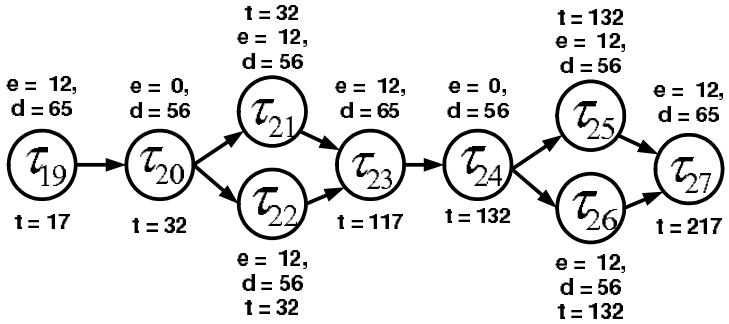

FIG. 7.24 – La construction de la tâche OS de la ressource R_{22}

7.5 Calcul automatique d'une allocation

Dans cette section, nous supposons que l'application n'est pas encore allouée sur les ressources et les dispositifs. Nous appliquons l'heuristique proposée dans ce manuscrit pour allouer ses différents blocs dans l'architecture matérielle.

Nous commençons tout d'abord par l'allocation du groupe d'exclusion ex_0 dans le dispositif D_1 ($D_1 \in local(ex_0)$). Pour cela, nous construisons une nouvelle ressource R_1 pour contenir ses différents blocs fonctionnels. En se basant sur l'approche hybride, nous validons la faisabilité temporelle des blocs de ce groupe dans cette ressource.

Une fois ex_0 alloué, nous commençons l'allocation du groupe ex_1. Nous essayons, tout d'abord, d'ajouter ses différents blocs dans la ressource R_1. En appliquant l'approche hybride, nous concluons la faisabilité des blocs de ex_0 et ex_1 dans cette ressource.

Une fois, ces deux groupes alloués, nous commençons l'allocation du reste des groupes dans le dispositif D_2. En suivant la même technique, nous vérifions leur faisabilité dans une seule ressource dénotée R_2. En tenant en compte cette vérification, la distribution de l'application devient alors sur deux ressources de dispositifs (figure 7.25).

Nous présentons, dans la figure 7.26, le pré-ordonnancement de la ressource R_1 ainsi que la tâche récurrente correspondante. Dans la figure 7.27, nous présentons le pré-ordonnancement de la ressource $R2$. Nous construisons à partir de cet ordonnancement la tâche OS correspondante (figure 7.28).

7.6 Analyses et Discussions

Nous partageons l'analyse de cette étude en deux parties. La première partie met en évidence l'apport de l'approche hybride, alors que la deuxième discute l'utilité de l'heuristique d'allocation.

7.6.1 Apport de l'approche hybride

Dans la section 7.4, le problème à résoudre est de construire un ordonnancement faisable dans les différentes ressources des dispositifs D_1 et D_2.

Une première solution évidente est de calculer un seul pré-ordonnancement (pour les deux dispositifs) de tous les blocs fonctionnels de l'application. Dans ce cas, nous avons à générer un séquencement de 44 instances d'actions (en considérant l'hyper-période fixée). Cette solution n'est pas intéressante car nous appliquons une politique oisive qui n'optimise pas les temps de réponses des blocs fonctionnels [Stankovic *et al.*, 1998].

La deuxième solution, appliquée dans cette étude, est l'approche hybride. Elle se base sur une politique EDF préemptive tout en étant conforme à la norme. Dans ce cas, nous construisons pour chaque ressource une tâche OS récurrente.

Cette solution hybride réduit considérablement le nombre de tâches à construire dans le dispositif :

* D_1 de 4 actions à 2 tâches récurrentes
* D_2 de 14 actions à 2 tâches récurrentes

D'autre part, pour construire les sous-tâches des tâches récurrentes, deux solutions distinctes sont possibles :

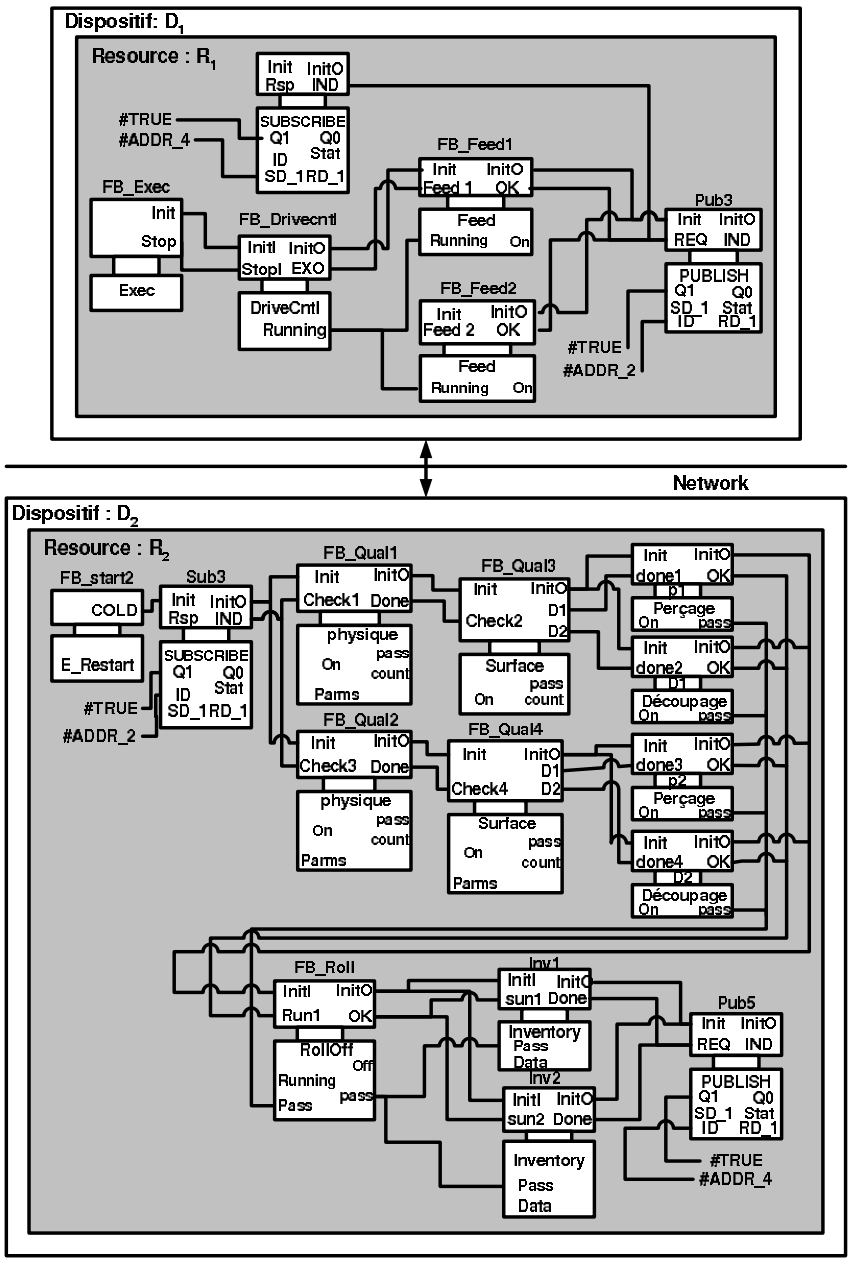

FIG. 7.25 – La nouvelle distribution de l'application

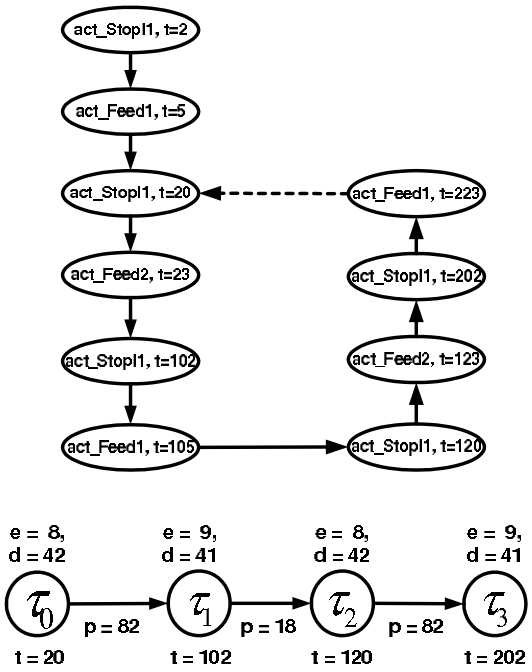

FIG. 7.26 – Le pré-ordonnancement et la tâche OS de la ressource R_1

* Nous construisons pour chaque instance d'action une sous-tâche. Dans ce cas, le nombre total de sous-tâches est égal à 44.

* En appliquant la technique proposée dans le chapitre 5, nous construisons (sous conditions) une seule sous-tâche pour une séquence d'instances. Dans ce cas, le nombre de sous-tâches construites est 27.

7.6.2 Le Gain de l'allocation

Dans la section 1.5, nous appliquons l'heuristique, proposée dans le chapitre 6, pour allouer l'application dans les deux dispositifs D_1 et D_2.

Grâce à cette approche, nous arrivons à construire dans chaque dispositif une seule tâche OS au lieu de deux. De plus, le nombre total des sous-tâches construites est égal à 11 au lieu de 27. Cette réduction en nombre de sous-tâches réduit considérablement la complexité de l'analyse d'ordonnançabilité proposée dans [Baruah, 2003].

D'autre part, en considérant cette minimisation du nombre de ressources, le nombre de blocs fonctionnels d'interfaces à construire est réduit. Ce nombre est devenu égal à 4 au lieu de 9 ce qui réduit la complexité de toute l'architecture fonctionnelle.

Enfin, si nous ajoutons d'autres blocs fonctionnels (de priorités inférieures) à l'application, nous risquons que les différentes tâches OS construites ne soient plus faisables. Dans ce cas, nous avons à appliquer des retours arrières pour remettre en cause le déploiement.

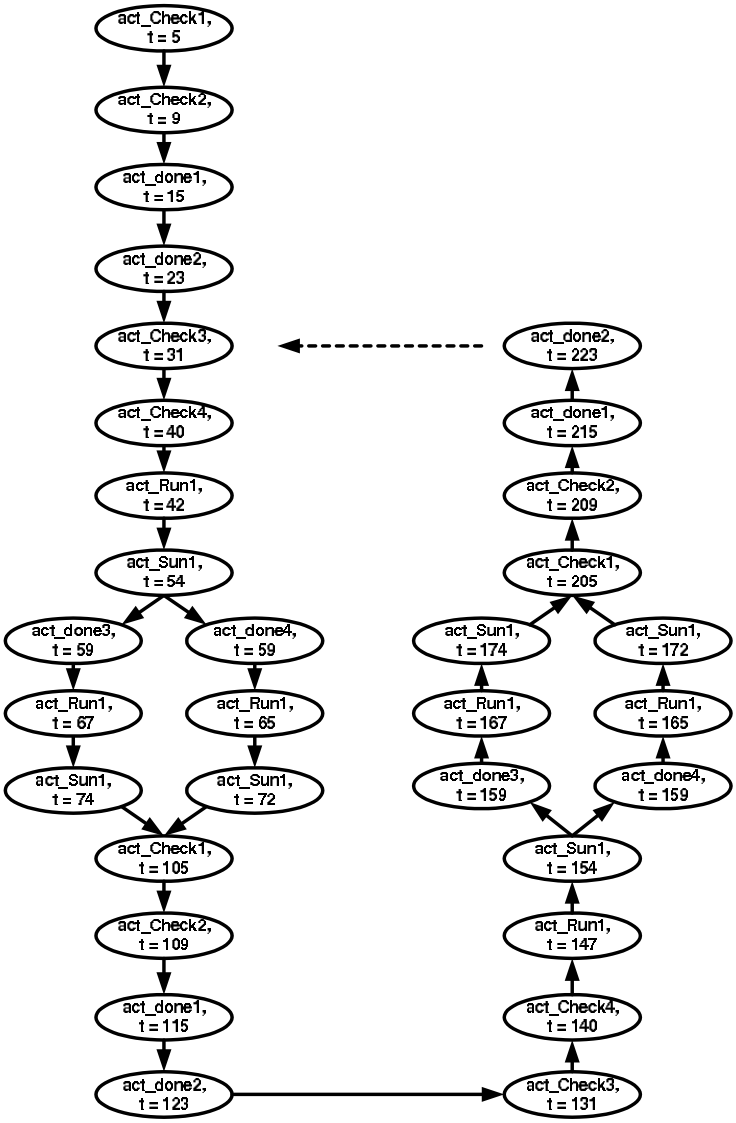

FIG. 7.27 – Le pré-ordonnancement de la ressource R_2

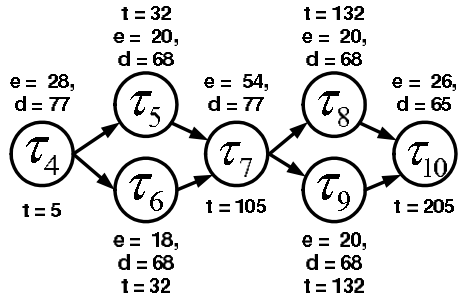

FIG. 7.28 – La tâche OS de la ressource R_2

7.7 Conclusion

Dans ce chapitre, nous avons étudié une application, à base de blocs fonctionnels, contrôlant une chaîne de production. Nous avons enrichi l'architecture fonctionnelle de l'application en tenant en compte le concept de ressource proposé dans la norme. De plus, nous avons considéré des contraintes fonctionnelles, temporelles et de support d'exécution.

Dans un premier temps, nous avons considéré une distribution quelconque de l'application sur l'architecture matérielle. Pour vérifier les contraintes temporelles, nous avons appliqué l'approche hybride proposée dans le chapitre 5. Dans un deuxième temps, nous avons abordé le problème d'allocation étudié dans le chapitre 6. Nous avons appliqué l'heuristique proposée pour allouer les différents blocs tout en considérant les contraintes du cahier des charges.

Enfin, en se basant sur cette étude, nous avons mis en évidence l'apport des différentes approches proposées dans ce manuscrit. Le premier apport est l'exploitation d'une politique EDF préemptive pour ordonnancer les différents blocs fonctionnels de l'application. Le deuxième est la réduction, par construction de l'allocation, du nombre de tâches OS et de leurs sous-tâches dans chaque calculateur. Cela réduit par conséquence la complexité de l'analyse de faisabilité.

Chapitre 8

Conclusion Générale et Perspectives

Dans ce manuscrit, nous nous intéressons à l'approche par composants pour le développement d'applications de contrôle industriel. L'objectif est de prendre avantage de cette approche, en terme de la modularité de conception, ainsi que la ré-utilisabilité de bibiliothèques de composants.

En considérant des contraintes fonctionnelles et temporelles décrites dans le cahier des charges [Khalgui *et al.*, 2004b], la validation *a priori* d'applications de contrôle à comportement critique est nécessaire. De ce fait, la composition de ses composants doit être prévisible hors-ligne. De nos jours, plusieurs technologies de composants sont exploitées dans différents cadres d'applications. Dans ce manuscrit, nous choisissons la technologie des blocs fonctionnels supportant la composition hors-ligne. Nous nous basons, en particulier, sur le standard IEC 61499 permettant le développement d'applications de contrôle industriel, à base de Bloc Fonctionnels et distribuées.

Un Bloc Fonctionnel est défini, dans le standard, comme un composant réactif supportant des fonctionnalités. Une application est définie comme un réseau de Bloc Fonctionnels communicants et localisés dans un ou plusieurs calculateurs. Cette norme, très répandue dans l'industrie, permet une description statique de l'application ainsi que de son support d'exécution.

8.1 Contributions

En l'état actuel, la norme IEC 61499 présente des imprécisions qui ne permettent pas de considérer un comportement déterministe d'un bloc fonctionnel soumis à des contraintes de temps. Ce problème majeur concerne la gestion des occurrences d'événements en entrée du bloc. De plus, la définition du concept de *ressource*, comme présenté dans la norme, reste trop pauvre pour évaluer les performances d'une application décrite sous forme de réseaux de Bloc Fonctionnels et déployée sur une plate-forme d'exécution.

Dans le chapitre 3, nous identifions soigneusement les problèmes de déterminisme du comportement d'un bloc fonctionnel. Nous évoquons, en particulier, le problème de perte d'événements dû à la surcharge du buffer [Khalgui *et al.*, 2005b]. De plus, nous proposons une modélisation du comportement d'un bloc fonctionnel à l'aide des automates temporisés [Khalgui *et al.*, 2004a]. Nous avons attribué, en particulier, aux différents événements d'entrée des degrés de priorité pour garantir un comportement déterministe du bloc.

Enfin, nous enrichissons le standard en proposant deux définitions fonctionnelles et opérationnelles du concept de ressource [Khalgui *et al.*, 2006a]. Cette contribution permet, dans la suite, un déploiement de l'architecture fonctionnelle de l'application dans une architecture opérationnelle.

Dans le chapitre 4, nous considérons la distribution de l'application sur une seule ressource d'un seul calculateur. Cette application reçoit des stimuli, événements périodiques de son environnement. De plus, nous considérons des bornes sur les temps de réponse de bout en bout que doit respecter une telle application. Ces bornes peuvent représenter des durées maximales à respecter entre des lectures périodiques sur des capteurs et l'activation des actionneurs correspondant. Afin d'exploiter les travaux précédents sur l'ordonnancement, nous proposons une transformation d'une telle application vers un système d'actions sous contraintes de précédence [Khalgui *et al.*, 2006c]. Une action de ce système est une fonctionnalité d'un bloc fonctionnel correspondant à un événement d'entrée particulier.

Pour respecter toutes les bornes sur les temps de réponse, nous proposons une approche calculant pour chaque action de l'application, une échéance. Nous étendons de plus cette approche pour résoudre le problème de perte d'événement. En prenant en compte la taille du buffer, le calcul de l'échéance d'une action permet d'éviter tout problème de perte tout en assurant le respect des bornes [Khalgui *et al.*, 2005b].

Pour valider le comportement temporel d'une application à base de Bloc Fonctionnels, nous proposons une analyse d'ordonnançabilité basée sur la construction d'un graphe d'accessibilité [Khalgui *et al.*, 2005a]. Dans le cas où certaines contraintes temporelles ne sont pas vérifiées sous conditions, nous étendons cette analyse en autorisant l'ordonnancement de l'application en mode dégradé [Khalgui *et al.*, 2006d]. En se basant sur un modèle de dégradation particulier, nous proposons une approche construisant un graphe d'accessibilité dégradé.

Enfin, si les Bloc Fonctionnels sont faisables (en mode strict ou dégradé), nous générons un ordonnancement statique implémentant tous les scénarios possibles de l'application [Khalgui *et al.*, 2006c]. Cet ordonnancement, caractérisé comme un graphe acyclique direct, est de structure conditionnelle.

Dans le chapitre 5, nous considérons une distribution de l'application sur plusieurs ressources de plusieurs dispositifs [Khalgui *et al.*, 2006a]. Nous proposons une approche hybride déployant les Bloc Fonctionnels localisés dans chaque ressource vers une tâche OS. L'application devient alors un ensemble de tâches indépendantes distribuées sur plusieurs calculateurs [Khalgui *et al.*, 2006b]. Nous appliquons, dans chaque calculateur, une condition d'ordonnançabilité en-ligne validant la faisabilité de ces tâches. En combinant deux politiques d'ordonnancement une préemptive et l'autre non-préemptive, cette approche optimise l'analyse de la faisabilité de l'application dans chaque calculateur.

Pour terminer la validation de l'application, nous avons à valider le trafic sur le réseau. Dans ce manuscrit, nous considérons un réseau de type CAN connectant les différents calculateurs. Nous construisons statiquement sa messagerie en caractérisant temporellement les différents messages échangés. Cette caractérisation permet la construction des priorités statiques de ces messages. Le message le plus prioritaire devient alors celui de la plus courte échéance [Khalgui *et al.*, 2006a].

Dans le chapitre 6, nous supposons que l'application n'est pas allouée sur les noeuds de l'architecture matérielle. Le problème est alors d'allouer ses blocs tout en prenant en compte des contraintes fonctionnelles et temporelles [Khalgui and Rebeuf, 2006]. Nous considérons de plus, des contraintes sur le support d'exécution en termes de nombre de tâches maximal autorisé par le système d'exploitation. Vu que le problème est classiquement NP-complet, nous proposons une heuristique particulière déployant les différents Bloc Fonctionnels dans des tâches OS [Khalgui and Rebeuf, 2006]. Cette heuristique, basée sur l'approche hybride, vérifie toutes les contraintes temporelles considérées. En particulier, dans le cas où le déploiement d'un bloc est infaisable, un

retour arrière "backtracking" est possible pour chercher une autre solution de déploiement.

Enfin, dans le chapitre 7, nous considérons une étude de cas appliquant les contributions de ce manuscrit. L'exemple considéré est une chaîne de production de pièces en plastique. Cet exemple extrait d'un document de référence est enrichi pour prendre en compte le concept de la ressource. De plus, nous avons introduit un comportement conditionnel à l'application pour enrichir la problématique.

Dans une première étape, nous supposons que l'application est déjà distribuée sur une architecture matérielle. Pour valider son comportement temporel, nous appliquons l'approche hybride construisant les différentes tâches OS dans les calculateurs. Cette approche construit aussi la messagerie du réseau considéré.

Dans une deuxième étape, nous supposons que l'application n'est pas déjà déployée. En considérant des contraintes fonctionnelles, temporelles et de support d'exécution, nous appliquons l'heuristique de déploiement proposée. Nous montrons l'utilité de cette heuristique en termes de minimisation du nombre de tâches à construire.

8.2 Perspectives

Malgré les travaux intensifs réalisés de nos jours autour du standard IEC 61499, plusieurs points restent à étudier sur la technologie des Bloc Fonctionnels.

Dans nos travaux futurs, nous planifions l'étude des Bloc Fonctionnels d'interface dans une ressource. Vu que ces blocs supportent l'accès à des procédés physiques partagés, nous avons à trouver une approche optimisant la construction de ces blocs dans une ressource. De plus, nous avons à étudier en détail les intergiciels (*"Process interface"* et *"Communication interface"*) utilisés dans un dispositif IEC 61499 pour supporter les communications avec les procédés physiques ainsi qu'avec d'autres dispositifs.

D'autre part, nous planifions l'optimisation de l'heuristique de déploiement selon des critères bien déterminés. Un de ces critères est la minimisation d'énergie qui permet, en garantissant un comportement correct de l'application, de réduire l'énergie dépensée. Un autre critère à prendre en compte est la minimisation de la mémoire réduisant le coût de l'application à développer. La considération de ces critères permettra le développement d'applications de contrôle répondant aux besoins actuels du marché.

Dans nos travaux futurs, nous planifions aussi l'étude de la reconfiguration d'une application à base de Bloc Fonctionnels. Le terme reconfiguration exprime toute modification de l'application (ajouts, modifications et suppressions de Bloc Fonctionnels) pour répondre à un changement dans le comportement de l'environnement. En se basant sur les travaux effectués sur "le contrôle adaptatif", nous planifions la définition d'un *"objet de contrôle"* gérant la reconfiguration de l'application.

Cet objet définit tous *les scénarios* possibles de reconfiguration. Chacun de ces scénarios représente une distribution particulière de l'application sur l'architecture matérielle. On caractérise classiquement un tel scénario par un ensemble de *stratégies de contrôle* où chacune définit un comportement particulier de l'application. Le problème est alors de définir soigneusement chaque stratégie de chaque scénario de l'objet de contrôle.

Dans un premier temps, nous avons à modéliser l'objet de contrôle ainsi que ses conditions de commutation de scénarios. Ces conditions sont nécessaires à spécifier pour éviter toute perturbation de l'environnement. De plus, nous avons à modéliser soigneusement le concept de scénario ainsi que ses conditions de changement de stratégies de contrôle.

Une fois la modélisation correctement réalisée, nous avons à proposer une méthode validant tous les scénarios de reconfiguration. Notons, que chaque stratégie de contrôle de chaque scénario doit être soigneusement validée. Nous nous baserons , en particulier, sur l'heuristique de déploiement pour analyser la faisabilité temporelle de l'application.

D'autre part, nous planifions une approche générant le code d'une application re-configurable à base de Bloc Fonctionnels. Cette approche doit prendre en compte tous les scénarios définis dans l'objet de contrôle. Une fois implémentée, la réalisation d'un jeu de test *a posteriori* est nécessaire pour renforcer la validation fonctionnelle et temporelle de toute l'application.

Bibliographie

[Abi-Antoun *et al.*, 2005] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng. *Modeling and Implementing Software Architecture with Acme and ArchJava (Research Demonstration).* Proceedings of the 27th International Conference on Software Engineering, St. Louis, 2005.

[AES, 2006] *http ://www.cs.cmu.edu.* AESOP, 2006.

[Ahmad and Kwok, 1996] I. Ahmad and Y. K. Kwok. *Analysis, evaluation and comparison of algorithms for scheduling task graphs on parallel processors.* In Int. Symp. on Parallel Architecture, Algorithms and Networks. Pages : 207-213. China., 1996.

[Ali and El-Rewini, 1993] H. H. Ali and H. El-Rewini. *Task Allocation in Distributed Systems : A Split Graph Model.* Journal. Computer Math. Combination Computing, Vol 14, No 1, pp : 15-32, 1993.

[Allen, 1997] R. Allen. *A formal Approach to Software Architecture.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[Alur and Dill, 1994] R. Alur and D. Dill. *A theory of Timed Automata.* Theoretical Computer Science. Vol. 126(2) : 183-235, 1994.

[Amnell *et al.*, 2001] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. *Uppaal - Now, Next, and Future.* In Proceedings of Modelling and Verification of Parallel Processes (MOVEP'2k), France. LNCS Tutorial 2067, pages 100-125, F. Cassez, C. Jard, B. Rozoy, and M. Ryan (Eds.), 2001.

[André *et al.*, 2005] C. André, F. Mallet, and M.A. Peraldi-Frati. Real-time architecture description and quantitative analysis using uml. In *OMER3. Paderborn. Germany*, 2005.

[André, 2003] C. André. Modélisation de systèmes réactifs par une approche synchrone : Syncharts. In *In Z. Mammeri, editor, Real Time Summer school, France*, 2003.

[Aoyama, 1998] M. Aoyama. *New Age of Software Development : How Component-Based Software Engineering Changes the Way of Software Development.* Proc. 1st Workshop on Component Based Software Engineering, 1998.

[Archimedes-System, 2006] Archimedes-System. *Archimedes System Platform web page.* 2006.

[Articus, 1996] Articus. *Rubus OS Reference Manual.* Articus Systems, 1996.

[Baker, 1974] K. R. Baker. *Introduction to sequencing Scheduling.* John Wiley and Sons, 1974.

[Barbacci *et al.*, 1991] M. R. Barbacci, C. B. Weinstock, D. L. Boubledey, and R. W. Lichota. *Durra : an integrated Approach to Software Specification, Modeling and Rapid Prototyping.* Technical report, Software Engineering Institute n° CMU/SEI-91 TR-21, 1991.

[Barbacci *et al.*, 1993] M. R. Barbacci, C. B. Weinstock, D. L. Boubledey, M. J. Gardner, and R. W. Lichota. *Durra : a structure description language for developing distributed applications.* Software Engineering Journal, vol(8) : 83-94, 1993.

[Baruah *et al.*, 1999] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. *Generelized multiframe tasks.* Real-Time Systems : The international Journal Of Time Critical Computing 17(1) : 5-22, 1999.

[Baruah, 2003] S. K. Baruah. *Dynamic and Static Priority Scheduling of Recurring Real-Time Tasks.* Real-Time Systems. Kluwer Academic Publishers. Manufactured in the Netherlands, 2003.

[Baruah, 2004] Sanjoy Baruah. *Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms.* Proceedings of the IEEE International Real-Time Systems Symposium, Portugal. IEEE Computer Society Press., 2004.

[Bellissard and Riveill, 1995] L. Bellissard and M. Riveill. *Olan : A Language and Runtime Support for Distributed Application Configuration.* Journée du GDR de Programmation. France, 1995.

[Benveniste *et al.*, 1991] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. *Synchronous programming with events and relations : the SIGNAL language and its semantics.* Science of Computer Programming, 16 : 103-149, 1991.

[Berthomieu *et al.*, 2004] B. Berthomieu, J. P. Bodeveix, S. Devulder, J. M. Farines, M. Filali, P. Gaufillet, J. L. Lambert, P. Michel, O. Nasr, G. Padiou, P. O. Ribet, and F. Vernadat. *Cotre : prospective pour la vérification.* Rapport LAAS No04372 Contrat RNTL COTRE, 2004.

[Beugnard *et al.*, 1999] A. Beugnard, J. M. Jézéquel, and N. Plouzeau. *Making components contract aware.* IEEE Computer, 32(7) : 38-45, 1999.

[Billionnet *et al.*, 1992] A. Billionnet, M. C. Costa, and A. Sutter. *An efficient algorithm for a task allocation problem.* Journal. ACM, Vol.39, No 3, pp : 502-518, 1992.

[Birriello and Miles, 1994] G. Birriello and D. Miles. *Task Scheduling for Real-Time Multiprocessor Simulations.* 11th Workshop on RTOSS, pages : 70-73, 1994.

[Bokhari, 1993] S. H. Bokhari. *A Network Flow Model for Load Balancing in Circuit-Switched Multicomputers.* IEEE Transaction. Parallel and Distributed Systems, Vol 4, No 6, pp : 649-657, 1993.

[Box, 2000] D. Box. *House of COM : Is COM dead? MSDN magazine.* Microsoft, 2000.

[Brennan *et al.*, 2002] R. Brennan, S. Olsen, F. Fletcher, and D. Norrie. *Comparing two Approaches to Modelling Decentralized Manufacturing Control Systems with UML Capsules.* 13éme IEEE International Workshop on Database and Expert Systems Applications. France., 2002.

[Cambazard *et al.*, 2004] H. Cambazard, P. E. Hladik, A. M. Déplanche, N. Jussien, and Y. Trinquet. *Decomposition and learning for a real time task allocation problem.* Principles and Practice of Constraint Programming (CP 2004), Lecture Notes in Computer Science, pp. 153-167, Springer-Verlag, 2004.

[Cengic *et al.*, 2005] G. Cengic, K. Akesson, B. Lennartson, Y. Chengyin, and Ferreira P. *Implementation of full synchronous composition using IEC 61499 function blocks.* Automation Science and Engineering 2005. IEEE International Conference on Aug 1 pages : 267-272, 2005.

[Christensen, 2000a] J. H. Christensen. *Design patterns for systems engineering in IEC 61499.* Otto-Von-Guericke-Universitat Magdeburg, 2000.

[Christensen, 2000b] J.H. Christensen. Basic concepts of iec 61499. In *Verteile Automatiseiriung (Distributed Automation)*, pages 55–62. Magdeburg, Germany, 2000.

[Chu and Lan, 1987] W. W. Chu and L. M. Lan. *Task Allocation and Precedence Relations for Distributed Real-Time System.* IEEE Transaction. Computers, Vol 36, No 6, pp : 667-679, 1987.

[Chu, 1980a] W. W. Chu. *Task Allocation in Distributed Data Processing.* Computer, Vol. 13, pp : 57-69, 1980.

[Chu, 1980b] W. W. Chu. *Task Allocation in Distributed Data Processing.* Computer, Vol 13, pp : 57-69, 1980.

[Coffman, 1976] E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* New York : John Wiley and Sons, 1976.

[CORBA, 1997] CORBA. *CORBA Services : Common Object Services Specification.* Object Management Group, 1997.

[CORBA, 1998] CORBA. *CORBA Components.* Report ORBOS/99-02-01, Object Management Group, 1998.

[CORFU-FBDK, 2006] CORFU-FBDK. *http ://seg.ee.upatras.gr/corfu/dev/index.htm.* Patras University, 2006.

[Cottet *et al.*, 2000] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Ordonnancement temps réel.* Hermes Science Publications. ISBN : 2-7462-0099-6, 2000.

[Crnkovic and Larsson, 2002] I. Crnkovic and M. Larsson. *Building reliable component-based software systems.* Artech House. London, 2002.

[Cucu and Goossens, 2006] L. Cucu and J. Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In *the 11th IEEE International Conference on Emerging Technologies and Factory Automation, (ETFA'06), Prague*, 2006.

[Cucu and Sorel, 2004] L. Cucu and Y. Sorel. *Ordonnancement non-preemptif pour systemes temps reel a contraintes de precedences et de latences.* Quatriemes Journees Francophones de Recherche Operationnelle (FRANCORO'04), Germany, 2004.

[Cucu and Sorel, 2005] L. Cucu and Y. Sorel. *Condition d'ordonnancabilite pour systemes temps reel non-preemptif a contraintes de precedences, de periodicites et de latences.* 6eme congres de la Societe Francaise de Recherche Operationnelle et d'Aide a la Decision (ROADEF'05), France, 2005.

[DACHSview, 2006] DACHSview. *Visual programming of Real-Time Applications.* Steinhoff Automation, 2006.

[David *et al.*, 2003] A. David, G. Behrmann, K.G. Larsen, and W. Yi. *A Tool Architecture for the Next Generation of Uppaal.* Technical report Uppsala University, 2003.

[Debruyne *et al.*, 2004] V. Debruyne, F. simonot Lion, and Y. Trinquet. *EAST-ADL an architecture description language - validation and verification aspects.* IFIP Workshop on Architecture Description Language, WADL04, 2004.

[Defour *et al.*, 2004] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Extra-functional contract support in components. In *In Proc. of International Symposium on Component-based Software Engineering (CBSE7)*, 2004.

[Deplanche and Durand, 1999] A.M. Deplanche and E. Durand. *CLARA : un langage de configuration dedie a la description d'architectures d'applications temps reel.* RTS'99, Conference Real-Time and Embedded Systems,Paris, France., 1999.

[Dissaux *et al.*, 2004] P. Dissaux, M. F. Amine, and P. Michel. *Architecture Description Languages.* Workshop On Architecture Description Languages (wadl), World Computer Congress. France, 2004.

[D'Souza and Wills, 1998] D. D'Souza and A. C. Wills. *Objects, Components and Frameworks : The Catalysis Approach.* Reading, MA : Addison-Wesley, 1998.

[Durand, 1998] E. Durand. *Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de petri temporels.* PhD thesis, université de Nantes, Ecole Centrale de Nantes, 1998.

[Ehrig *et al.*, 2002] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. *A transformation-based component framework for a generic integrated modeling technique.* In Proc of Integrated Design and Process Technology 2002., 2002.

[Faure *et al.*, 2002] J.M. Faure, C. Schnackenbourg, and J.J. Lesage. *Toward IEC 61499 Function Blocks diagrams verification.* IEEE International Conference on Systems Man and Cybernetics. SMC'2002. Tunisia, 2002.

[FBDK, 2006] FBDK. *Function Block Development Kit.* http ://www.holobloc.com, 2006.

[Ferrarini and Veber, 2004] L. Ferrarini and C. Veber. *Implementation approaches for the execution model of IEC 61499 applications.* 2éme IEEE International Conference on Industrial Informatics. Germany. INDIN'04, 2004.

[Ferro *et al.*, 1999] E. Ferro, R. Cayssials, and J. Orozco. *Tuning the Cost Function in a Genetic/Heuristic Approach to the Hard Real-Time Multitask-Multiprocessor Assignment Problem.* Proceeding of the Third World Multiconference on Systemics Cybernetics and Informatics, 1999.

[Filipe, 2000] Juliana Kuster Filipe. *Fundamentals of a Module Logic for Distributed Object Systems.* Journal of Functioning and Logic Programming, 2000(3), 2000.

[Filipe, 2002a] J. K. Filipe. *A Logic-Based Formalisation for Component Specification.* Journal of Object Technology, Special Issue : TOOLS USA 2002 Proceedings, 1(3) : 231-248, 2002.

[Filipe, 2002b] Juliana Kuster Filipe. *A Logic-Based Formalization for Component Specification.* 2002.

[Fioukov *et al.*, 2002] A. V. Fioukov, E. M. Eskenazi, D. K. Hammer, and M.R. V. Chaudron. *Evaluation of static properties for component-based architectures.* In Proceedings 28th Euromicro Conf. IEEE Computer Society Press. Germany, 2002.

[French, 1982] S. French. *Sequencing and Scheduling.* Halsted Press, 1982.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman Co, 1979.

[Goessler and Sifakis, 2002a] G. Goessler and J. Sifakis. *Composition for Component-Based Modeling.* FMCO02, 2002.

[Goessler and Sifakis, 2002b] G. Goessler and J. Sifakis. *Composition for Component-Based Modeling.* FMCO02, 2002.

[Goessler and Sifakis, 2003] G. Goessler and J. Sifakis. *Component-based construction of deadlock-free systems.* FSTTCS03, 2003.

[Grandpierre *et al.*, 1999] T. Grandpierre, C. Lavarenne, and Y. Sorel. *Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors.* CODES'99 7th International Workshop on Hardware/Software Co-Design, Italy, 1999.

[Grolleau and Choquet-Geniet, 2002] Emmanuel Grolleau and Annie Choquet-Geniet. *Off-line Computation of Real-Time Schedules using Petri nets.* Discrete Event Dynamic Systems, DEDS, vol. 12 (3), Kluwer Academic Publishers, pp. 311-333, 2002.

[Hagge and Wagner, 2005] N. Hagge and B. Wagner. *A New Function Block Modeling Language Based on Petri Nets for Automatic Code Generation.* IEEE transaction on Industrial Informatics, Vol. 1, No 4, pp. 226-237, 2005.

[Hamdaoui and Ramanathan, 1994] M. Hamdaoui and P. Ramanathan. *A Service Policy for Real-Time Customers with (m,k)-Firm Deadlines.* Proceedings of the Fault-Tolerant Computing Symposium, pp. 196-205., 1994.

[Hamdaoui and Ramanathan, 1995] M. Hamdaoui and P. Ramanathan. A dynamic priority assignement technique for streams with (m,k)-firm deadlines. In *IEEE Transactions on Computers, Vol. 44, No. 4*, 1995.

[Heinman and Council, 2001] G.T. Heinman and W.T. Council. *Component based Software Engineering : Putting the pieces together.* Addison-Wesley, 2001.

[Hladik *et al.*, 2006] P. E. Hladik, H. Cambazard, A. M. Déplanche, and N. Jussien. *Solving allocation problems of hard real-time systems with dynamic constraint programming.* 14th International Conference on Real-time and Network Systems (RTNS06), France, 2006.

[Hussain and Frey, 2004] T. Hussain and G. Frey. *Developing IEC 61499 Complinat Distributed Systems with Network Enabled Controllers.* IEEE Conference on Robotics Automation and Mechatronics. Singapore, 2004.

[ICS-Triplex-ISaGRAF, 2006] ICS-Triplex-ISaGRAF. *Commercially Available IEC 61499 Software.* http ://www.icstriplex.com, 2006.

[IEC61131, 1993] IEC61131. *Programmable Controllers Part 3.* International Standard IEC 1131-3. Bureau Central de la commission Electrotechnique Internationale. Switzerland., 1993.

[IEC61499-1, 2003] IEC61499-1. *Function Blocks for Industrial Process Measurements and Control Systems.* International Standard IEC-TC65-WG6, 2003.

[IEC61499-2, 2004] IEC61499-2. *Industrial Process Measurements and Control Systems.* International Standard IEC-TC65-WG6, 2004.

[IEC61804-1, 2006] IEC61804-1. *IEC 61804 Part 1 General Requirements Committee Draft (SC65CWG7(PT1CD)2).* IEC61804-1, 2006.

[IEC61804-2, 2006] IEC61804-2. *IEC 61804 Part 2 Specification Committee Draft (SC65CWG7(PT2CD)1).* IEC61804-2, 2006.

[IEC61804, 2006] IEC61804. *http ://www.ifak-md.de/wg7/Vesteras/IEC61804OverviewISA.pdf.* Ifak, 2006.

[IMS-systems, 2006] IMS-systems. *Intelligent Manufacturing Systems Reasearch and Development (R and D) program.* IMS, http ://www.ims.org, 2006.

[Jifeng *et al.*, 2003] H. Jifeng, Z. Liu, and L. Xiaoshan. *Contract-Oriented Component Software Development.* Technical Report 276, UNU-IIST, P.O.Box 3058, Macau, April 2003., 2003.

[Jézéquel, 2006] Jean-Marc Jézéquel. Reifying the semantic domains of component contracts. In *In 5th IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'06, Braga, Portugal. Springer SBM*, 2006.

[Kasahara and Narita, 1984] H. Kasahara and S. Narita. *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing.* IEEE Trans. Computers, Vol 33, No 11, pp : 1, 023 - 1, 029, 1984.

[Khalgui and Rebeuf, 2006] M. Khalgui and X. Rebeuf. A heuristic based method for automatic deployment of distributed component based applications. In *IES2006. Nice. France*, 2006.

[Khalgui *et al.*, 2004a] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A behavior model for iec 61499 function blocks. In *MOCA04. Denmark*, 2004.

[Khalgui *et al.*, 2004b] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A contribution to the validation of complex real-time systems. In *Setit04. Tunisie*, 2004.

[Khalgui *et al.*, 2005a] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A schedulability analysis of an iec 61499 control application. In *FET 05. Mexico*, 2005.

[Khalgui *et al.*, 2005b] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A schedulability condition of an iec 61499 control application with limited buffers. In *OMER3. Germany*, 2005.

[Khalgui *et al.*, 2006a] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. Component-based deployment of industrial control systems : an hybrid scheduling approach. In *ETFA06, Czech*, 2006.

[Khalgui *et al.*, 2006b] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A hybrid scheduling approach of centralized component based applications. In *WFCS2006. Italy*, 2006.

[Khalgui *et al.*, 2006c] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A static scheduling generator for the deployment of a component based application. In *Book Chapter. Heinz-Nixdorf Institute publisher. ISBN3-939350-10-9. Germany*, 2006.

[Khalgui *et al.*, 2006d] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A tolerant temporal validation of components based applications. In *INCOM 06. France*, 2006.

[Khalgui *et al.*, 2006e] M. Khalgui, X. Rebeuf, and F. ZAMPOGNARO. Adaptable opc-xml contracts taking into account network traffic. In *ETFA05, Italy*, 2006.

[Klein *et al.*, 1993] M H. Klein, T. Ralya, B. Pollack, R. Obenza, and MG. Harbour. *A practioner's handbook for Real-Time analysis. Guide to Rate monotonic Analysis for Real-Time Systems.* Kluwer Academic Publisher, 1993.

[K.M and K.G, 1995] Zuberi. K.M and Shin. K.G. Non-preemptive scheduling of messages on controller area network for real-time control applications. In *Real-Time Technology and Applications Symposium, Page(s) :240 - 249.*, 1995.

[Kramer *et al.*, 1989] J. Kramer, J. MAGEE, and M. Sloman. *Constructing Distributed Systems in CONIC.* IEEE TSE, vol. 15(6) : 663-675, 1989.

[Krutchen, 1995] P. B. Krutchen. *The 4+1 View Model of Architecture.* IEEE Software, 42-50, 1995.

[Kwok and Ahmad, 1999] Y. K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. In *ACM Computing Surveys, vol. 31, No. 4.*, 1999.

[Lawler, 1982] E. L. Lawler. *Deterministic and Stochastic Scheduling.* Recent Developments in Deterministic Sequencing and Scheduling : A Survey, p. 35-74. The Netherlands : Reidel, Dordrecht, 1982.

[Lenstra and Kan, 1978] J. K. Lenstra and A. H. G. R. Kan. *Complexity of Scheduling Under Precedence Constraints.* Operations Research, Vol 26, No 1, pp : 23-35, 1978.

[Leung and Whitehead, 1982] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. In *performance Evaluation 2(1982). 237250*, 1982.

[Lewis, 2002] R. Lewis. *Modelling Control Systems using IEC61499.* The institution of Electrical Engineers, 2002.

[Liu and Layland, 1973] C. Liu and J. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment.* Journal of the ACM, 20(1) :46–61, 1973.

[Liu, 2000] JWS. Liu. Real-time systems. In *Prentice Hall*, 2000.

[Lobov *et al.*, 2003] A. Lobov, J. L. Martinez Lastra, R. Tuokko, and V. Vyatkin. *Methodology for Modeling Visual Flowchart Control Programs using Net Condition/Event Systems Formalism in Distributed Environments.* 9th IEEE Conference on Emerging Technologies in Factory Automation (ETFA'03), Proceedings ETFA'03, Portugal, 2003.

[Luckham and Vera, 1995] D. Luckham and J. Vera. *An Event-based Architecture Definition Language.* IEEE TSE, Vol 21(9) : 717-734, 1995.

[Luckham *et al.*, 1995] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. *Specification and Analysis of System Architecture Using Rapide.* IEEE TSE, Vol 21(4) : 336-355, 1995.

[Luders and Lau, 2002] F. Luders and K. K. Lau. *Specification of Software Components.* In I. Crnkovic and M. Larsson, editors : Building Reliable Component-Based Software Systems, p : 23-38, Artech House, 2002.

[Lui and Corroyer, 1993] Z. Lui and C. Corroyer. *Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. An empirical comparison.* Proc. of PARLE'93, 5th Int. PARLE conference, Germany, pages : 452-463, 1993.

[M. Herlihy, 1992] J. E. B. Moss M. Herlihy. Transactional memory : Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1992.

[M. Richard and Cottet, 2002] E. Grolleau M. Richard, P. Richard and F. Cottet. *Contraintes de précédences et ordonnancement mono-processeur.* Real Time and Embedded Systems, edited by Teknea, pp. 121-138, 2002.

[MAGEE *et al.*, 1995] J. MAGEE, N. DULAY, S. Eisenbach, and J. Kramer. *Specifying Distributed Software Architectures.* In proceedings of the 5th European Software Engineering Conference (ESEC 95). Espagne, 1995.

[Medvidovic *et al.*, 1999] N. Medvidovic, D. Rosenblum, and T. Richard N. *A language and environment for architecture-based software development and evolution.* In Proceedings of the 21st international conference on Software engineering. USA, 1999.

[Microsoft-Corporation, 1995] Microsoft-Corporation. *The component object model specification.* Microsoft, 1995.

[Microsoft, 1996] Microsoft. *The Component Object Model Specification.* Report Vol 99. Microsoft Standard. WA : Microsoft., 1996.

[Mok and Feng, 2001] A. K. Mok and X. Feng. *Towards Compositionality in Real-Time Ressource Partitioning Based on Regularity Bounds.* Real-Time Systems Symposium, pp. 129-138, 2001.

[Mok *et al.*, 1996] A.K. Mok, D. A. Stuart, and F. Jahanian. *Specification and Analysis of Real-Time Systems : Modechart Language and Toolset.* in Formal Methods For Real-Time Computing, John Wiley, 1996.

[Moriconi and Reimenschneider, 1997] M. Moriconi and R.A Reimenschneider. *Introduction to SADL 1.0 : A Language for Specifying Software Architecture Hierarchies.* Technical Report SRI-CSL-97-01, SRI International, 1997.

[Moschoyiannis *et al.*, 2003] S. Moschoyiannis, M.W. Shields, and J. Küster-Filipe. *Formalising Well-Behaved Components.* In Hung Dang Van and Zhiming Liu, eds, Proceedings of Formal Aspects of Component Software FACS'03, Satellite Workshop of FME 2003, 8-9 September 2003, Pisa, Italy, pp. 121-142, 2003.

[.net technology, 2006] .net technology. *http ://www.microsoft.com/net/.* Microsoft, 2006.

[network, 1993] CAN network. Road vehicles. interchange of digital information. controller area network (can) for high speed communication. In *ISO 11898.1st edition*, 1993.

[Ommering *et al.*, 2000] R. Van Ommering, F. van der Linden, and J. Kramer. *The KOALA component model for consumer electronics software.* IEEE Computer, 2000.

[Ommering, 2002] R. Van Ommering. *Building product populations with software components.* In Proceedings 24th International Conference on Software Engineering. ACM Press, 2002.

[Open-Group, 2000] Open-Group. *Architecture Description Markup Language (ADML).* Version 1. Technical Report I901, Reading, UK, 2000.

[Osek, 2004] Osek. Osek consortium. oil :osek implementation language. In *version 2.5, http ://www.osek-vdx.org*, 2004.

[Panjaitan *et al.*, 2005] S. Panjaitan, T. Hussain, and G. Frey. *Development of re-configurable Distributed Controllers in IEC 61499 based on Task Schedules described by UML diagrams or Gant charts.* 3éme IEEE International Conference on Industrial Informatcs. Autralia, 2005.

[Pattison, 2000] T. Pattison. *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0, 2nd Edition.* Microsoft, 2000.

[Pecos-project, 2006] Pecos-project. *www.pecos-project.org/publications.html.* 2006.

[Peng *et al.*, 1997] D. Peng, K. G. Shin, and T. F. Abdelzaher. *Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems.* IEEE Transactions on Software Engineering. Vol 23. No 12, 1997.

[Peterson, 1981] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall. ISBN 0136619835, 1981.

[Projet-Artist, 2003] Projet-Artist. *Roadmap : Component-based Design and Integration Platforms.* www.artist-embedded.org, 2003.

[Ramamritham, 1990] K. Ramamritham. *Allocation and Scheduling of Complex Periodic Tasks.* 10th International Conference on Distributed Computing Systems, 1990.

[Rockwell, 2006] Rockwell. *http ://www.holobloc.com.* Rockwell automation, 2006.

[Sandnes, 1996] F. E. Sandnes. *A hybrid genetic algorithm applied to automatic parallel controller code generation.* 8th IEEE Euromicro Workshop on Real-Time Systems, 1996.

[Selvakumar and Murthy, 1994] S. Selvakumar and C. S. R. Murthy. *Static Task Allocation of Concurrent Programs for Distributed Computing Systems with Processor and Ressource Heterogeneity.* Journal. Parallel Computing, Vol 20, No 6, pp. 835-851, 1994.

[Shukla and Agrawal, 1994] S. B. Shukla and D. P. Agrawal. *A framework for mapping periodic Real-Time Applications on Multicomputers.* IEEE Trans. Parallel and Distributed Systems. Vol 5, No 7, pp : 788-784, 1994.

[Sifakis, 2005] J. Sifakis. *A Framework for Component-based Construction.* 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM05), 2005.

[Stanica and Guéguen, 2004] M. Stanica and H. Guéguen. *Using timed automata for the verification of IEC 61499 applications.* Workshop on Discrete Event Systems, WODES04, France, 2004.

[Stankovic *et al.*, 1998] J. Stankovic, M. Spuri, and K. Ramamritham. Deadline scheduling for real-time systems. In *Kluoner Academic Publishers*, 1998.

[Stewart *et al.*, 1997] D. B. Stewart, R. A. Volpe, and P. K. Khosla. *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects.* IEEE Transaction on Software Engineering, 1997.

[Sun-Microsystems, 1997] Sun-Microsystems. *JavaBeans.* http ://www.java.sun.com, 1997.

[Sun-Microsystems, 2002] Sun-Microsystems. *Enterprise JavaBeans specification.* http ://java.sun.com/products/ejb/index.html, 2002.

[Szyperski, 1998] C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley. New York, 1998.

[Takada and Sakamura, 1995] H. Takada and K. Sakamura. mu-itron for small-scale embedded systems. In *IEEE MICRO, vol.15, no.6, pp.46-54*, 1995.

[Thramboulidis *et al.*, 2004] K. Thramboulidis, G. Doukas, and A. Frantzis. *Towards an implementation Model for FB-based Reconfigurable Distributed Control Applications.* 7th IEEE Int Symposium on Object-oriented Real-Time Distributed Computing. Autria, 2004.

[Thramboulidis, 2005a] K. Thramboulidis. *Model Integrated Mechatronics - Towards a new Paradigm in the Development of Manufacturing Systems.* IEEE Trans on Industrial Informatics. Vol1, No 1, 2005.

[Thramboulidis, 2005b] Kleanthis Thramboulidis. *IEC 61499 in Factory Automation.* Int. Conf. on Industrial Electronics, Technology and Automation (CISSE-IETA 05), 2005.

[Tindell *et al.*, 1992] K. Tindell, A. Burns, and A. Wellings. *Allocation Hard Real-Time Tasks : an NP-Hard Problem Made Easy.* The Journal of Real-Time Systems, 4(2) : 145-165, 1992.

[Tindell *et al.*, 1994] K. W. Tindell, H. Hansson, and A. J. Wellings. Analyzing real-time communications : Controller area network (can). In *in Proc. Real-Time Systems Symposium, pp. 259-263*, 1994.

[Tindell *et al.*, 1995] K. Tindell, A Burns, and A. J. Wellings. Calculating controller area network (can) message response times. In *Control Engineering Practice, Vol 3, no 8, pp. 1163-1169*, 1995.

[Tindell, 1994] K. Tindell. *Allocating Real-Time Tasks.* Journal of Real-Time Systems ; Vol 4, No. 2(1994), 1994.

[Tracz, 1993] W. Tracz. *lileanna : a parameterized programming language.* In Proceedings, Second International Workshop on Software Reuse. Italy, 1993.

[Unicon-homepage, 2006] Unicon-homepage. *http ://unicon.sourceforge.net/.* UNICON, 2006.

[Vestel, 1993] S. Vestel. *Scheduling and Communicating in MetaH.* Real-Time Systems Symposium, p194-200, Raleigh-Durham, 1993.

[Vestel, 1995] S. Vestel. *MetaH Programmer's Manual.* Technical report, Honeywell Technology Center, n° Version 1.01, 1995.

[Vyatkin and Hanisch, 2001] V. Vyatkin and H. Hanisch. *Formal-modelling and Verification in the Software Engineering Framework of IEC 61499 : a way to self-verifying systems.* ETFA01. Nice, 2001.

[Vyatkin *et al.*, 2005] V. Vyatkin, J. Christensen, and J. Lastra. *OOONEIDA : An Open, Object-Oriented Knowledge Economy for Intelligent Distributed Automation.* IEEE Trans on Industrial Informatics. Vol1, No 1, 2005.

[Western-Reserve-Controls, 2006] Western-Reserve-Controls. *W2 series IEC 61499 Development Kit*. http ://www.wrcakron.com/IEC61499.html, 2006.

[Woolsey and Swanson, 1974] R. E. D. Woolsey and H. S. Swanson. *Operations Research for Immediate Applications : A Quick and Dirty Manual*. Harper and Row, 1974.

[XIA *et al.*, 2004] F. XIA, Z. Wang, and Y. Sun. *Allocating IEC function blocks for parallel real-time distributed control system*. IEEE International Conference on Control Applications, Vol. 1, pp254-259, 2004.

[Xu, 1993] J. Xu. *Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence and Exclusion Relations*. IEEE Trans. Software Eng. Vol 19. No 2. pp : 139-154, 1993.

[Yang and Gerasoulis, 1993] T. Yang and A. Gerasoulis. *List scheduling with and without communication delays*. Parallel Computing Journal, 19 : 1321-1344, 1993.

[Zoilt *et al.*, 2005] A. Zoilt, G. Grabmair, F. Auinger, and C. Sunder. *Executing real-time constraint Control Applications modeled in IEC 61499 with respect to Dynamic Reconfiguration*. 3éme IEEE International Conference On Industrial Informatics. Australia. INDIN'05, 2005.

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur Charles ANDRE, Professeur, Université de Nice Sophia Antipolis, Nice

Monsieur Jean-Marc FAURE, Professeur, SUPMECA, LURPA, Cachan

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Monsieur KHALGUI Mohamed

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Validation temporelle et déploiement d'une application de contrôle industrielle à base de composants"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : « **Informatique** »

Fait à Vandoeuvre, le 18 janvier 2007
Le Président de l'I.N.P.L.,
F. LAURENT

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE - NANCY -

TEL. 33/03.83.59.59.59
FAX 33/03.83.59.59.55