RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Conception d'un système de reconfiguration de logiciels multitâches embarqués sur satellite †

Damien Cailliau[1], Arthur Leger[2], Olivier Marin[3], Bertil Folliot[2,4]

[1]Observatoire de Meudon
Département d'Etudes Spatiales / CNRS
5 Place Jules Janssen
92195 Meudon Cedex

[2]Laboratoire d'Informatique de Paris 6 / CNRS
Université Pierre et Marie Curie
4 Place Jussieu
F-75252 Paris Cedex 05

[3]Laboratoire d'Informatique du Havre
25 rue Philippe Lebon
BP 540 76058 Le Havre cedex

[4]Institut National de la Recherche en Informatique et Automatique
Projet Système à Objets Répartis
Domaine de Voluceau, Rocquencourt
78153 Le Chenay Cedex

Couriel : damien.cailliau@obspm.fr, arthur.leger@lip6.fr, olivier.marin@univ-lehavre.fr, bertil.folliot@lip6.fr

---

Résumé
Dans le cadre des études prospectives à la mission française Corot nous proposons une approche permettant la reconfiguration de logiciels multitâches embarqués sur satellites. La conception repose sur une chaîne de contrôle et de planification de reconfiguration englobant des outils sols basés sur un langage de configuration, un interpréteur de bytecode spécifique à bord du satellite et une application rendue reconfigurable par l'usage d'un OS générique implémenté par un intergiciel. Le prototype réalisé montre d'excellentes performances et paraît extensible à d'autres domaines d'application.

Mots-clés : Système reconfigurable, logiciel embarqué, commande-contrôle, évolutivité

---

1. Introduction

Les systèmes spatiaux comportent énormément de logiciel spécifiquement conçus pour la mission et intensivement testés et validés : contrôle de l'attitude du satellite ou de la sonde, contrôle des ressources (énergie, cryogénie), des communications avec le segment sol, en plus des logiciels dédiés aux instruments embarqués et à la mission scientifique. L'ensemble de ces logiciels s'interface, créant une chaîne comuniquante jusqu'au segment sol, permettant ainsi aux opérateurs du satellite de suivre et contrôler le système en orbite. On identifie traditionnellement trois familles de fautes, corrigées par modification de ces logiciels : 1) les bogues dans le logiciel (très rares) ; 2) les erreurs de commande qui peuvent placer le satellite dans un état incorrect (quelque fois irrécupérable par télécommandes correctives comme ce fût le cas pour le satellite Soho [13]) ; 3) les cas de pannes matérielles (telles le non déploiement de l'antenne principale de la mission américaine Galileo corrigée en 1996 [18]).

†Ce travail est partiellement financé par le projet LIP6 PLERS

Ces fautes nécéssitent de modifier le logiciel embarqué. La technique traditionellement utilisée est la technique du "patch". Elle consiste au remplacement par téléchargement d'instructions machine directement dans la mémoire de l'ordinateur de bord. D'autres méthodes plus évoluées consistent en l'insertion d'un certain nombre de "crochets" dans le code initial lors de la conception, crochets qui peuvent être utilisés pour détourner le fil d'exécution, voir de référencer les procédures critiques que l'on veut pouvoir rendre modifiables par des tables d'indirections [2]. A notre connaissance, seul le SMF (Software Maintenance Facility), développé par Alénia Aerospazio [10], présente un système bord/sol documenté facilitant la maintenance du logiciel de vol, et ce dès la conception de celui-ci. Il met à disposition des opérateurs des moyens de test et de simulation importants permettant de reproduire l'état du logiciel bord et les divers scénari de mise à jour du logiciel.
Or les systèmes numériques embarqués et les traitements qu'ils réalisent prennent une part de plus en plus importante dans les systèmes spatiaux, systèmes dont la durée de vie, la complexité et la portée ne cessent d'augmenter. Ceci fait apparaitre de nouveaux besoins en terme d'évolutivité et de flexibilité des logiciels embarqués, besoins nécessitant une évolution dans la manière dont ceux-ci sont conçus et exploités. Dans le cadre des études prospectives et de faisabilité du satellite astrophysique européen COROT [1], le projet PLERS [7] – Plateforme Logicielle Embarquée Reconfigurable pour Satellite – étudie depuis un an une plate-forme logicielle répondant à cette approche.
Après avoir rappelé les besoins (section 2), nous identifions section 3 l'ensemble des contraintes auxquelles doit se soumettre un système flexible en contexte spatial. La section 4 présente notre analyse d'un tel dispositif offrant la reconfigurabilité de logiciels embarqués, proposant une approche combinant un intergiciel à bord et un langage de configuration comme base des outils sol. La section 5 détaille la chaine de reconfiguration que nous proposons, articulée autour de l'intergiciel BUSARD – BUs de Support d'Applications Reconfigurable Dédié–, du langage CURL – Corot Unified Reconfiguration Language –, d'un interpréteur et d'un simulateur. Enfin nous comparons cette approche avec d'autres travaux menés sur l'évolutivité des logiciels embarqués en section 6 avant de conclure (section 7).

2. La flexibilité des logiciels embarqués sur systèmes spatiaux

Cette section présente les besoins émergents en reconfigurabilité des logiciels embarqués sur satellites. Ces besoins vont au delà des besoins de modification à l'exécution par suite d'erreurs, présentés en introduction.

2.1. Evolution et réutilisation

Il existe un besoin croissant de pouvoir faire évoluer le logiciel au cours de la vie du satellite ; ceci soit pour prendre en compte de nouveaux objectifs dans le cadre de missions scientifiques, soit pour intégrer de nouvelles technologies qui pouvaient ne pas être matures au moment du lancement du système spatial. Par ailleurs les missions interplanétaires actuelles imposent une très longue durée de transfert avant l'arrivée sur site (Cassini mettra 7 ans pour rejoindre Saturne, une mission vers Pluton prendrait de 8 à 9 ans de voyage, et la sonde Européenne Rosetta mettra 8 ans pour arriver en orbite de la comète Wirtanen). Il serait ainsi possible de ne pas avoir à développer l'ensemble des logiciels avant le lancement. En plus de réduire le nombre de causes possibles de retard de lancement, ceci permet d'allonger le temps de réflexion sur les fonctionnalités logicielles à implémenter et donc d'utiliser les technologies les plus achevées : on ne téléchargerait les fonctionnalités de la mission scientifique que lorsque l'on en a besoin [1].
Ceci offre l'avantage d'un gain substantiel en terme de mémoire de stockage. Cette approche a été initiée par la sonde américaine Cassini, courant 2000, lorsque de nouveaux éléments logiciels ont été téléchargés dans les ordinateurs de bord après que les scientifiques et la NASA aient décidé de profiter du passage à proximité de Jupiter pour étendre son programme d'étude.
Les sondes planétaires et les satellites ont souvent une durée de vie qui excède la durée prévue de la mission. Ainsi il pourrait être extrêmement intéressant de réutiliser le matériel développé pour accomplir de nouvelles missions scientifiques, limitant de ce fait les coûts impliqués dans les missions spatiales. Ceci n'est possible qu'en modifiant les fonctionnalités logicielles embarquées.

[1] Les fonctionnalités nécessaires pour maintenir la sonde et piloter les expériences spécifiques lors de sa phase de transfert son différentes de celles requise lors de la phase de rendez-vous.

2.2. Adaptation

Traditionnellement, les données brutes acquises par les instruments scientifiques embarqués sont transmisent au sol en ayant subi une intervention minime, souvent sous forme de compression sans ou avec faible perte. En effet la bande passante de transmission au sol allouée à l'instrument et le volume d'information acquise sont généralement du même ordre de grandeur. Or ces bandes passantes restent constantes alors que les taux de production de données des instruments augmentent de plus en plus. Il devient alors nécessaire de déporter à bord une partie croissante des traitements jusque là effectués au sol. C'est le cas de la mission scientifique Corot. En outre, le Remote Exploration et Experiment project du JPL (REE) [9] étudie la possibilité de spatialiser les techniques de *super-clustering*. Leur but est de permettre des capacités accrues de traitement à bord et donc de faciliter la croissance des performances des instruments scientifiques embarqués.

Ces traitements modifient radicalement les données brutes acquises de manière à en extraire au maximum l'information scientifiquement significative. Or les algorithmes utilisés sont généralement basés sur des modèles qui n'ont souvent pas été testés sur des cas réels. Lorsque ces modèles sont trop loins de la réalité l'information reçue au sol est dégradée et perd de sa valeur scientifique. Il est donc primordial de pouvoir adapter ces traitements au cours de la mission au fur et à mesure que la connaissance de l'environnement et que les modèles s'affinent. Ceci n'est possible qu'en modifiant simplement et de manière sûre les fonctionnalités logicielles implémentées à bord.

3. Contraintes

Après avoir identifié les contraintes générales de la reconfiguration à bord de satellites, nous présentons dans cette section les contraintes issues du contexte particulier de l'étude : la mission européenne Corot.

3.1. Contraintes générales

Les besoins présentés ci-dessus imposent de concevoir le logiciel bord ainsi que sa chaîne de conception et de contrôle-commande de manière à ce qu'il soit évolutif. Un tel système doit respecter un certain nombre de contraintes.

Tout d'abord les logiciels embarqués sur satellites sont très limités en terme de ressources, que ce soit au niveau de la puissance CPU, de la mémoire disponible ou de l'importance du facteur de charge du processeur.

Ces logiciels étant de la classe des logiciels réactifs, il ont également des contraintes temps réel fortes. Ainsi lorsque le logiciel s'exécute sur une plate-forme dont l'attitude est asservie par traitement, il est souvent primordial que les modifications du logiciel n'affectent pas le déroulement des calculs de pointage et de stabilisation. Par ailleurs les satellites de télécommunication ont des contraintes de qualité de service telles qu'ils ne peuvent souffrir d'interruption de fonctionnement. Ceci implique que toute opération de modification doit se dérouler avec une intrusion bornée (autorisant à continuer certains traitements durant la reconfiguration). On retrouve cette contrainte de continuité de fonctionnement (que l'on appelle *dynamisme*) à divers degrès dans de nombreux systèmes spatiaux.

En outre les logiciels embarqués en environnement spatial doivent respecter des contraintes de fiabilité, de tolérance aux fautes et de sûreté de fonctionnement. En effet l'éloignement des systèmes sur lesquels ils opérent, leur coût, leur unicité et leur importance (scientifique, commerciale, stratégique, politique) font qu'il est hautement préjudiciable de les perdre. Ainsi de manière à ce que les opérations de reconfiguration ne soient ni destructives pour la mission ni ne mènent à un état bloquant, elles doivent être vérifiées avant dêtre activées. De même les erreurs lors des modifications doivent être détectées et relayées vers le segment sol, le logiciel prenant alors un état deterministe et connu. Toutes ces opérations doivent être réversibles. Par ailleurs, de manière à garantir que les services essentiels assurant la survie de l'instrument sont toujours présent à bord et intègres, ceux-ci ne doivent pas être modifiables et ne doivent en aucun cas être affectés par les mises à jour. Pour conserver également un cohérence et une prédictibilité dans ces modifications, il est préférable de les limiter à des modules logiciels clairement identifiés sans agir sur des instructions individuelles.

Enfin, la solution apportant la flexibilité doit éviter de complexifier le développement du logiciel bord et capitaliser sur les pratiques et méthodologies en vigueur dans le secteur spatial. De la même manière, notre système doit proposer une succession d'outils d'aide à la mise en oeuvre fiable de modifications à

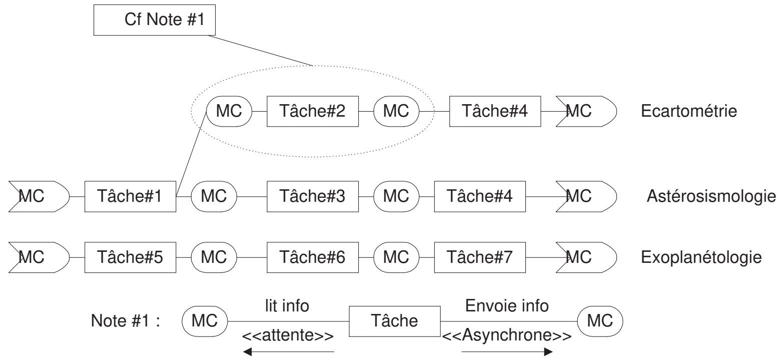

Fig. 1 – Structure typique d'une application Corot

distance du logiciel de vol du satellite, d'un usage aisé et réduisant les possibilités d'erreurs humaines.

3.2. Contraintes d'implémentation : le cas de Corot

Nous intégrons nos travaux dans une reflexion globale sur des architectures logicielles flexibles pour petits satellites. De manière à garantir une certaine pérennité et portabilité aux principes que nous étudions, ceux-ci doivent être suffisament génériques pour ne dépendre qu'au minimum du noyau multitâche utilisé, du processeur cible ou de la suite d'outils de développement. La mission Corot nous sert de domaine applicatif concret pour la réalisation d'un démonstrateur. Celui-ci doit donc en respecter les contraintes.

Le logiciel bord de Corot est un logiciel multitâche, avec contraintes temps réel. Il se compose essentiellement d'un système d'exploitation (abrégé *OS* dans la suite) pilotant une application et prenant en charge l'ordonnancement du logiciel et l'interface avec le matériel. Les premières études de faisabilité de Corot ont défini un OS bord optimum s'articulant autour du noyau multitâche ThreadX d'Express Logic, Inc. L'application regroupe l'ensemble des traitements des données acquises, et s'organise en chaînes de tâches concurrentes (figure 1). Ces chaînes s'interfacent avec l'OS et entre elles par l'intermédiare de média de communication (queues de messages ou éléments de synchronisation de type événements ou sémaphores). Cette architecture logicielle est classique dans les systèmes embarqués, notamment les systèmes spatiaux et donc ne dégrade pas la portabilité de notre approche. Par ailleurs le satellite Corot étant un système non autonome, entièrement piloté par le sol, il doit en être de même pour son logiciel bord et toute opération de modification de son application.

Le calculateur de bord de Corot s'appuie sur un DSP 21020 d'Analog Device, cadencé à 20MHz spécialement conçu pour résister aux radiations, 384 kmots de mémoire de travail à répartir entre code et données, et seulement 500 kmots d'EEPROM utilisés pour stocker du code et des paramètres de fonctionnement. Dix pourcent du temps CPU est affecté à l'OS.

Nous considérons ces contraintes d'implémentation comme suffisamment fortes et représentatives du contexte d'instrumentation spatiale pour être généralisables.

4. Critères de conception de logiciels embarqués reconfigurables

Cette section décrit notre analyse de la flexibilité logicielle et décrit les éléments de conception et les choix d'architecture qui en découlent. Les éléments réalisés sont détaillés dans les sections suivantes.

4.1. Grains de reconfiguration

Dans [4] nous présentons le niveau de granularité des opérations que l'on considère : ajout, suppression de tâches et de timers logiciels, modification de procédures, ajout, suppression de média de communication au sein des chaînes de traitement. Ces modifications ciblant des portions de la configuration du logiciel embarqué, nous parlons de *reconfiguration*. En nous limitant à cette granularité on atteint la souplesse et la cohérence désirée en se restreignant à des objets logiciels connus : une application est construite à partir de cet ensemble d'abstractions simples.

Le plus petit grain de reconfiguration que nous avons choisi est donc la procédure, sans imposer aucune

contrainte sur la taille du code de celle-ci. Il est aussi possible qu'une procédure reconfigurable soit découpée en autant de procédures séquentielles (non reconfigurables) que nécessaire, permettant même de grouper des traitements complètement dépendants en une unité de reconfiguration.
La reconfigurabilité du logiciel est limitée à l'application seule en rendant les autres éléments du logiciel, dont l'OS, inaccessibles par conception à toute modification, assurant ainsi que les services essentiels sont toujours intègres et présents à bord.

4.2. Dynamisme

L'ensemble des éléments modifiés ne doivent pas être exécutés lors de la reconfiguration. L'application doit être synchronisée dans un état connu à partir duquel on pourra la reprendre après reconfiguration. Le domaine de l'informatique répartie ayant largement étudié cette problématique (voir par exemple [5], [17], [11]), nous nous en sommes inspirés pour définir la notion de point de configuration. Ce sont des points de synchronisation fixes définis dans le déroulement de chaque tâche (l'entité exécutable insécable dans les logiciels multitâches). De toutes celles étudiées, une telle approche offre la souplesse et le déterminisme nécessaires en ne requérant que le minimum de ressources en temps de calcul et en usage mémoire.
Or le dynamisme de toute opération (telle la reconfiguration) menée sur une application, quelle qu'elle soit, n'est possible que si sa durée est inférieure à la marge laissée par les traitements. Lorsqu'on ne peut s'assurer que cette règle sera systématiquement respectée il faut arrêter un certain nombre de ces traitements et répartir les opérations dans les intervalles temporels ainsi libérés. Une telle opération est dite *semi-dynamique*. De cette manière nous respectons les contraintes temps réel des tâches et chaînes de tâches non concernée par la reconfiguration en cours.

4.3. Repartition bord-sol

La reconfigurabilité d'un logiciel embarqué sur satellite implique un système transverse sol-bord de planification et de mise en oeuvre. Du côté du segment sol, ce système est composé d'une suite d'outils et de procédures permettant d'analyser la nouvelle architecture fonctionnelle et structurelle du logiciel, de développer les nouveaux éléments de code, de planifier les opérations de modification, mais aussi de tester et valider la reconfiguration avant toute mise en oeuvre. Le segment sol doit s'assurer que le système bord fonctionne tel que souhaité. Du côté du segment bord, outre la présence d'un OS supportant la modularité et la reconfigurabilité définis, la mise en oeuvre des modifications implique l'existence de moyens spécifiques de commande des modifications, le *configurateur bord*. Dans notre approche, nous définissons la chaîne de reconfiguration présentée figure 2.
L'OS bord doit offrir un nombre fini de primitives permettant la manipulation, la création et la destruction des abstractions logicielles utilisées (définies au paragraphe 4.1) pour construire une application, mais aussi pour contrôler son exécution : l' *inteface générique*. Composées, ces primitives permettent d'atteindre l'ensemble des modifications souhaitées. Ainsi le segment sol doit émettre une succession de commandes, de plus ou moins haut niveau, de manière à mettre en oeuvre les reconfigurations. Le logiciel bord doit donc incorporer un interpréteur de commandes capable de traduire des commandes de haut niveau en la succession d'appels aux primitives nécessaires.
Le logiciel bord n'implémentant aucune autonomie, les commandes doivent être émises et interprétées à bord suivant une séquence dont la logique et la cohérence ont été vérifiées au sol avant émission. En effet les opérations de modification n'ont un sens que dans un contexte donné et suivant une structure de séquence précise : une commande de retrait d'une tâche de la configuration n'a de sens que si la tâche existe ; de même il est dangereux de modifier l'implémentation d'une procédure appelée par une tâche avant d'avoir stabilisé l'exécution de la tâche concernée.
En choisissant de définir une configuration comme un assemblage d'abstractions logicielles régies par des règles simples, il est possible, si le code utilisé est entièrement relogeable en mémoire et les interfaces parfaitement définies, de rendre indépendante l'architecture du logiciel de l'implémentation de ces modules. La connaissance de la fonctionnalité de chacun d'entre eux permet de répartir le travail entre un architecte en charge de la conception du logiciel à haut niveau et un développeur en charge de la programmation des fonctionnalités implémentées par chaque élément logiciel exécutable (tâches, timers logiciels et procédures). L'architecte a alors la possibilité de décrire une configuration de manière abstraite ou textuelle. L'interêt d'un telle approche a d'ailleurs été présenté dans [6].
Les modifications à apporter à l'application peuvent s'exprimer en se basant sur cette même représentation

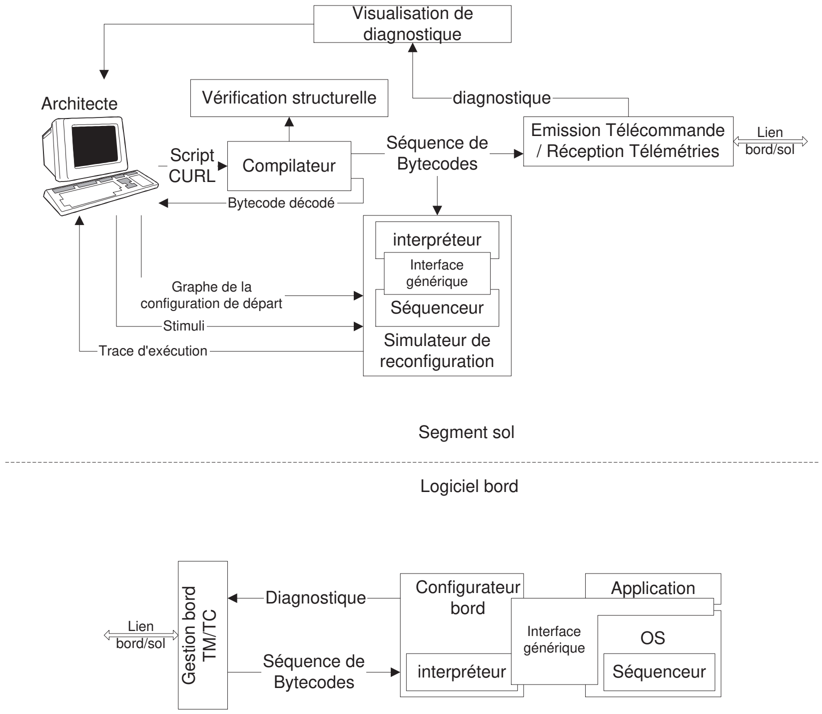

Fig. 2 – Chaîne de reconfiguration de logiciel embarqué

de la configuration. Ainsi nous adoptons la notion de langage de configuration, déjà largement étudiée pour répondre au problème de construction d'applications réparties (cf. par exemple [15], [14]) ou pour proposer un environnement distribué reconfigurable (cf. [3]). Une telle approche permet de définir en un même langage simple et humainement compréhensible à la fois une configuration (description statique), les opérations d'installation d'une configuration et les opérations de reconfiguration. En outre un langage peut masquer, par des instructions de haut niveau, un grand nombre de détails des commandes sous-jacentes, réduisant ainsi l'impact du facteur humain. Autre avantage, en compilant un script il est possible non seulement de générer une représentation logique d'une configuration, mais aussi de produire un bytecode compact que l'on peut directement envoyer au satellite sous forme de télécommandes. Celles-ci seront interprétées par le configurateur bord. On peut alors, à partir d'un ensemble de règles simples que doivent respecter configurations et opérations de reconfigurations, effectuer automatiquement des vérifications structurelles sur le graphe logique de la configuration produite et des vérifications de cohérence sur les séquences d'opérations de reconfiguration nécessaires pour l'atteindre.

La classe de logiciel à laquelle nous nous intéressons présente des aspects temps réel et nécessite que l'on teste, valide ou rejette une séquence de reconfiguration avant son envoi au satellite pour exécution. Par ailleurs ces reconfigurations ne sont pas des opérations statiques puisque des portions de logiciel continuent à s'exécuter durant les modifications. Le système dans son ensemble doit montrer une continuité d'exécution. Il est nécessaire, en plus des vérifications structurelles et de cohérence des scripts, de vérifier tout d'abord que la séquence de reconfiguration effectue les modifications escomptées, ensuite que son déroulement est correct : les tâches sont arrêtées, démarrées et synchronisées correctement, l'application reste dans un état cohérent à tout moment, les opérations de modification co-existent sainement avec les activités du logiciel et les contraintes temps réel de l'ensemble du système sont respectées tout au long des reconfigurations. Un outils simulant temporellement une configuration et fonctionnellement le configurateur bord, le *simulateur de reconfiguration*, permet d'atteindre ce but (se référer à [16] pour une large discussion sur les intérêts de la simulation dans la conception de logiciels spatiaux). A partir d'une représentation logique d'une configuration, les éléments exécutables étant réduits à leur durée et à leur date d'accès de communications ou d'appels, une simulation de l'ordonnanceur sous-jacent permet d'obtenir une trace d'exécution de l'application. Ensuite, en intégrant un interpréteur capable d'exécuter les séquences de commandes bytecodées on obtient une approximation du déroulement temporel d'une reconfiguration. Il est ainsi possible de défricher rapidement, avec un degré de confiance statistiquement suffisant, les différentes séquences de reconfiguration possibles et les divers scénarios d'exécution pour une séquence de reconfiguration donnée. Il ne reste ensuite à effectuer que quelques tests sur une cible réelle pour s'assurer de la validité fonctionnelle de la configuration et des opérations de reconfiguration, avant émission des télécommandes vers le satellite.

Il faut également vérifier que les opérations de reconfiguration se sont correctement déroulées à bord. Dans les techniques proches du patch, la seule solution pour s'assurer que le logiciel est correctement installé est d'envoyer au sol une copie de la zone mémoire concernée. Ceci peut être très gourmand en ressources, tout particulièrement dans le cadre de missions interplanétaires où le débit de communication est faible. En outre, lorsque les reconfigurations s'effectuent de manière dynamique, il faut que les informations de diagnostic puisse partager le lien descendant avec les données utiles monopolisant souvant la bande passante disponible. Notre approche offre plusieurs avantages à ce sujet : il est possible de reconstruire au sol une image précise du logiciel en n'envoyant que quelques informations sur les éléments logiciels installés (l'adresse et l'identifiant du code des éléments exécutables, l'identifiant associé à chaque élément logiciel, l'adresse du descripteur de cet élément) ainsi que des informations sur les liens entre ceux-ci. Des techniques de *checksum* ou de *CRC* inclus dans le code téléchargé ou stocké en EEPROM permettent de s'assurer qu'il n'ont pas été corrompus. En outre chaque commande de la séquence de reconfiguration exécutée à bord peut être acquittée, acquittement inclu dans le diagnostic émis. L'analyse de ces informations peut être faite a posteriori en utilisant des outils dédiés et devrait permettre de reproduire l'état du système embarqué de manière à aider les opérateurs à conclure sur la réussite de la reconfiguration.

5. Réalisation

5.1. BUSARD : un intergiciel dédié supportant la reconfiguration

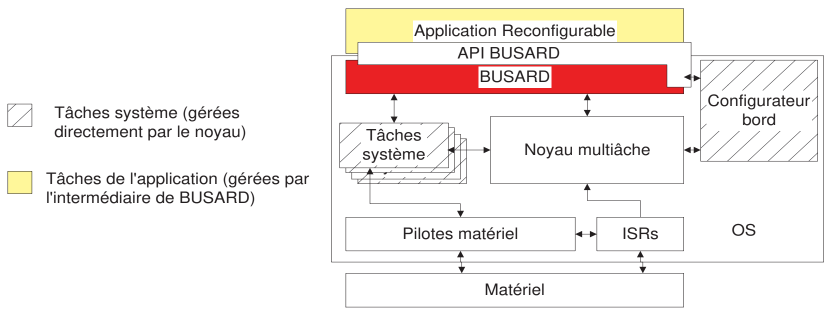

Fig. 3 – L'intergiciel BUSARD

5.1.1. Principes

Pour atteindre le grain de reconfiguration défini, l'OS bord doit gérer l'application comme un assemblage de modules logiciels, abstractions offertes par le noyau multitâche utilisé, mais doit aussi assurer que celles-ci sont indépendantes les unes des autres de manière à limiter l'impact des modifications et le nombre des opérations à mettre en oeuvre. En outre, l'OS doit offrir des points de contrôle d'exécution et des méthodes de gestion des éléments de l'application, permettant le démarrage, l'arrêt, la création, la destruction et la synchronisation des éléments ordonnançables (tâches, timers). Ces deux caractéristiques ne sont pas offertes de manière native par ThreadX, le noyau que nous utilisons, ni par la majorité des noyaux multitâche du commerce. Il est nécessaire d'extraire de ceux-ci une interface commune (générique) de programmation compatible avec notre approche, la rendant indépendante de ces spécificités d'implémentation. Nous atteignons ce but en définissant l'intergiciel BUSARD, généralisant les abstractions des noyaux multitâche et

offrant des moyens de gestion de configuration de l'application. Cet intergiciel peut être porté sur le noyau multitâche choisi, définissant ainsi un OS générique. Dans la version développée, BUSARD se présente sous la forme d'une bibliothèque de primitives écrites en C, encapsulant les appels au noyau multitâche ThreadX. Il offre en outre à l'application une interface figée de communication avec l'OS et le matériel (les senseurs et actuateurs) sous forme de média de communication spécifiques, non reconfigurables, ne dépendant que du système, qui font office de points d'entrée et de sortie de l'application.
BUSARD offre deux interfaces de programmation, schématisées par l'*interface générique* de la figure 2. La première, utilisable par l'application, est une collection de primitives responsables des accès aux média de communication (écriture, lecture dans une queue de message, positionnement, test d'événements au sein d'un groupe d'événements, etc.) et des appels de procédures reconfigurables. C'est par l'intermédiaire de ces primitives que l'ordonnanceur de l'application est invoqué. La deuxième offre au configurateur bord un ensemble de primitives lui permettant de gérer les modules logiciels (tâches, timers, procédures, média de communication), en les créant, les mettant à jour, les détruisant, effectuant les liens, etc. Le bus BUSARD est une collection de quatre mandataires typés, appelés communément *proxy* (procédures, channels, sémaphores, groupes d'événements). Ceux-ci contrôlent les accès aux média de communication et aux procédures, invoquent l'ordonnanceur lorsque nécessaire et assurent l'indépendance nécessaire entre éléments. Ils gèrent également l'état de l'application lors des reconfigurations. Tout accès à un élément du logiciel se fait donc par l'intermédiaire du bus, avec l'inconvénient d'un léger ralentissement des appels. Ainsi, la majeure différence entre l'usage de BUSARD et celui du noyau sous-jacent seul est que l'appel fait référence à un numéro d'entrée dans un proxy et non plus à l'adresse directe du média ciblé. Par ailleurs les appels aux procédures reconfigurables se font en mode message, le proxy procédures transmettant une structure de donnée C dont il ignore le contenu, relevant de la responsabilité du développeur. Les spécificités de la reconfiguration sont donc très peu intrusives dans le mode de programmation de l'application.

5.1.2. (Semi-)Dynamisme

Pour qu'une tâche ou une chaîne de tâches puisse être reconfigurée tout en laissant le reste de l'application se dérouler normalement, il faut isoler la chaîne en question et rendre passive l'ensemble des tâches et timers qui la constituent. Pour rendre passif un timer il suffit d'en arrêter le décompte. Chaque tâche est composée d'un ensemble de ports correspondant aux points d'accès à BUSARD. Nous définissons chacun de ces ports comme un point de passivation potentiel. L'architecte, lorsqu'il décrit une reconfiguration détermine sur quel port chaque tâche doit devenir passive pour permettre un redémarrage cohérent et sans inter-blocages en fin de reconfiguration. Pour devenir passive, une tâche doit joindre le port défini, l'accès à ce dernier la suspendant. De manière à limiter le temps de passivation et à éliminer les données en transit, chaque port de la tâche à passiver est verrouillé, c'est-à-dire que l'accès n'est pas effectif. L'opération de passivation étant comparable en durée aux cycles d'exécution des tâches, il n'est pas possible de garantir, dans cette première implantation, que l'ensemble de la reconfiguration prendra un temps négligeable par rapport au temps caractéristique de l'application. Notre solution, dans sa première version, permet donc des reconfigurations *semi-dynamiques*. L'isolation des chaînes de traitement reconfigurées est nécessaire pour que les tâches frontière subissent un minimum de perturbation et traitent d'éventuelles indisponibilités de service sans générer d'erreur. L'isolation équivaut à bloquer les accès aux ports associés aux média de communication en amont et en aval des chaînes reconfigurées. Il est de la responsabilité de l'application de prendre en compte ces indisponibilités suivant la politique de gestion des fautes en vigueur. De par cette interruption de service, il est primordial d'éviter que des données ne restent en transit dans ces chaînes durant la reconfiguration. Contrairement aux bus logiciels n'offrant qu'un bus de communication entre les éléments exécutables, comme c'est le cas des ORB ([14]) ou de systèmes tels OLAN ([5]), la causalité des communications n'a pas de sens dans notre cas : l'ordre d'exécution et de traitement des messages n'est pas régi par leur priorité mais par celle des tâches. BUSARD se charge donc de la destruction des messages et signaux en transit dans la chaîne reconfigurée ou ceux fournis en entrée de celle-ci au cours de la reconfiguration.
Pour reconfigurer une chaîne composée de plusieurs tâches il faut tout d'abord l'isoler, puis verrouiller l'ensemble des ports de ces tâches, ensuite indiquer pour chacune d'entre elles le point de passivation, puis enfin les passiver. Les autres tâches continuent leur exécution. Le démarrage des portions de chaînes reconfigurées est l'opération inverse : les port des tâches de la chaîne sont dé-verrouillés, la chaîne dé-isolée

et les tâches activées. De manière à pouvoir assurer la synchronisation avec le reste de l'application, c'est l'opérateur qui définit pour chaque tâche un delai de démarrage à partir d'un top de synchronisation.

5.1.3. Performances

Les premières mesures de performance de BUSARD montrent une faible dégradation en terme d'occupation du CPU : les accès aux média de communication sont augmentés de quelques 60 cycles, soit une dégradation de 10% par rapport aux performances de ThreadX seul. Ceci correspond au meilleurs cas de notre spécification et est parfaitement négligeable au regard des temps de l'application. En terme d'encombrement mémoire BUSARD occupe environ 2.5 kilo-mots, à comparer aux 8.5 kilo-mots nécessaires à ThreadX (une augmentation de 30%). En outre, BUSARD nécessite pour son propre fonctionnement 500 mots ajoutant 10% du volume de données nécessaire au fonctionnement de ThreadX. Finalement, les données nécessaires à la gestion d'une application sont de quelques 6.5 kilo-mots, dépendant du nombre d'éléments logiciels installés, là où ThreadX seul nécessite 5 kilo-mots. Notre implémentation de l'OS bord sous forme d'intergiciel ne dégrade donc que faiblement les performances du système embarqué. Ces performances ne devraient pas changer radicalement en portant BUSARD sur un autre noyau que ThreadX.

5.2. CURL

Ce paragraphe présente notre implémentation des différents éléments de la chaîne de reconfiguration (figure 2), du langage de configuration au configurateur bord.

CURL est un langage capable d'exprimer à la fois la structure d'une configuration et des opérations de reconfiguration. Chaque script CURL débute par une partie déclarative qui permet de définir l'ensemble des éléments invariants de la configuration. Ceux-ci sont les média de communication reflétant les accès aux senseurs et actuateurs et la description des modules exécutables et de leurs caractéristiques intrinsèques dans le cas d'un script de configuration. Un script de reconfiguration débute par la référence à la configuration de départ, préalablement mémorisée, ainsi qu'à la définition de tous les modules exécutables supplémentaires nécessaires. Ensuite le script est une succession d'instructions permettant d'installer ces modules exécutables, d'instancier des média de communication, d'effectuer les liens entre ces objets et de gérer l'exécution de l'application.

La figure 4 montre un script de reconfiguration écrit dans le langage CURL. Le compilateur développé (appelé psyCoPaT pour Corot Parser Tool), crée la séquence de commandes bytecodées nécessaire pour effectuer la reconfiguration définie, présente à l'opérateur une vue décodée du bytecode et traduit ce script en une représentation interne objet de la configuration obtenue.

PsyCoPaT est d'ores et déjà capable d'effectuer un certain nombre de vérifications syntaxiques et grammaticales sur le script, mais il reste à développer un outil de vérification structurelle de la configuration et des opérations de reconfiguration. Par ailleurs, il est prévu d'intégrer au compilateur un outil de gestion mémoire.

Lors de la définition du format des commandes de reconfiguration à envoyer au logiciel bord, nous avons adopté un souci de compacité, de sûreté et de pertinence. Chaque commande correspond à une succession de mots de 32 bits (le format du processeur utilisé, qui tend à se généraliser dans le spatial), bytecodés, correspondant au type de la commande (*l'opcode*), suivi des paramètres nécessaires. Du jeu de 58 commandes que nous avons identifié, la plus longue comporte sept paramètres, soit 8 mots, la plus courte deux paramètres. 76% des commandes possède deux paramètres, 14% trois paramètres et 7% cinq paramètres. Pour chaque mot d'une commande, les 4 premiers bits définissent le type du bytecode (opcode ou type du paramètre parmi les 11 types identifiés), les bits restants codant la valeur du paramètre ou le code de la commande (6 bits seulement sont nécessaires pour coder l'ensemble des commandes identifiées). De cette manière il est aisé de trouver le début d'une commande dans une trame de mots, de s'assurer de la correction des paramètres associés à une commande et donc de repérer toute corruption des télécommandes reçues. Les commandes identifiées ne contiennent que l'information nécessaire pour être mise en oeuvre à bord sans déduction, ni choix, se déclinant ainsi en plusieurs appels de primitives BUSARD.

Dans notre version actuelle, le configurateur bord est en cours de validationt. Il est basé sur 3 tâches système, l'une interprétant les commandes de reconfiguration, l'autre se chargeant du démarrage des tâches reconfigurées et de leur synchronisation avec le reste du logiciel, la dernière construisant un diagnostic de la reconfiguration. L'interpréteur bord charge et décode chacun des bytecodes jusqu'à obtenir

```
//Partie déclarative
#memory

useConfig :: "e:/proj/code/reconfiguration/appli_sol/test/config/strt_conf.cfg";

defineTask ::
     tache4 = "e:/proj/code/reconfiguration/appli_sol/test/codes/tache4.coff"
{
                  implSize = 348;
                  codeStackSize = 354;
                  descrAddress = 840;
                  dataIn :: {  source1 = 3;};
                  dataOut :: { destination = 5; };
                  procedures :: {traitement = 2;};
  };

defineProcedure ::
          proc2 =  "e:/proj/code/reconfiguration/appli_sol/test/codes/proc2.coff"
{
               implSize = 289;
               codeStackSize = 26;
               descrAddress = 1032;
 };

// Instructions de reconfiguration
stopTask :: tache1 on in1;
stopTask :: tache2 on in2;

unlink :: tache2.in2;

loadTask :: tache4
{
     allocatedStackSize = 600;
     stackAddress = 100;
     implAddress = 1800;
};

loadProcedure :: proc2
{
      implAddress = 29500;
};

declareChannel :: channel2
   {
               messageSize = 1;
               queueDepth = 3;
               descrAddress = 512;
               bufferAddress = 904;
   };

//Liens
read :: tache4.source1 -> channel1;
write :: tache4.destination -> channel2;
call :: tache4.traitement -> proc2;
read :: tache2.in2 -> channel2;

startTask :: tache1 at 10 withPriority 10;
startTask :: tache4 at 11 withPriority 8;
startTask :: tache2 at 12 withPriority 12;
```

Fig. 4 – Exemple de script CURL

une commande complète et exacte, qu'il peut exécuter. Chaque commande correspond à une séquence figée d'appels de primitives de BUSARD. A chaque exécution d'une commande, une télécommande d'acquittement est émise en écho. Lorsque l'interpréteur détecte une commande erronée, il émet une télémétrie d'erreur et charge une configuration par défaut en exécutant une séquence de commandes préalablement sauvegardée à bord. Une version de l'interpéteur a été développée pour le simulateur (cf. paragraphe 5.3). Comme elle utilise une interface identique à celle offerte par BUSARD, celle-ci est parfaitement représentative et portable sur le logiciel bord.

Finalement, lorsque l'ensemble des commandes a été exécuté, les informations sur les éléments installés et le contenu des proxies sont collectées et transmises au sol afin que l'opérateur puisse reconstruire une image de la configuration effectivement installée à bord. Le format de ces informations, ainsi que l'outil d'analyse de diagnostic sont en cours de validation.

5.3. Le simulateur de reconfiguration

De manière à valider le langage CURL lors de la phase de conception de notre solution et à proposer les prémices d'un outil de validation de scripts de reconfiguration, nous avons développé un simulateur de reconfiguration. Ce simulateur reçoit une représentation objet d'une configuration. Il associe à chacun des modules exécutables définis dans celle-ci une description de la séquence d'exécution de son code : sa durée et les dates d'accès à ses ports. En simulant le comportement de BUSARD (et du noyau sous-jacent) à chacun de ces accès, et en simulant les stimuli injectés dans le système, cet outil fournit une trace du déroulement temporel d'une configuration. L'adjonction d'un interpréteur de bytecode, copie conforme de l'interpréteur bord, permet en outre de simuler l'exécution d'une reconfiguration en la replaçant dans un contexte dynamique. De manière à augmenter l'efficacité de la simulation, le temps est exprimé en *pas d'exécution* correspondant à la durée moyenne d'un appel BUSARD (quelques 600 cycles, avec très peu de dispersion entre les différents appels), durée très inférieure aux durées de cycle des tâches de l'application. Ainsi, chaque accès à un port d'une tâche ou d'un timer, chaque appel à une primitive de BUSARD par l'interpréteur, chaque ajout d'une donnée dans l'un des médias de communication d'entrée prend un pas d'exécution.

Le simulateur nous donne donc, pour une série de séquences de stimuli donnée et pour plusieurs valeurs de paramètres (tels la date de début d'exécution du bytecode), des séries de traces d'exécution d'une

configuration et d'une même reconfiguration. De par la définition d'un pas d'exécution, ces traces ne sont pas exactes mais fournissent une première approximation sur la validité temporelle d'une configuration et sur la pertinence d'une séquence de reconfiguration. Ces traces aident à décider d'envoyer ou non la séquence de reconfiguration simulée. Par ailleurs, elles aident à déterminer si la configuration visée respecte les contraintes temporelles du systèmes. A l'issu de chaque exécution du simulateur, la configuration obtenue est mémorisée et réutilisable comme configuration initiale d'une prochaine simulation.
Des tests comparatifs entre simulation et réalité doivent être menés de manière estimer les performances de cette approche, mais ce simulateur nous a d'ores et déjà permis de démontrer la faisabilité de notre approche et de valider le langage CURL et le bytecode défini par psyCoPaT.

6. Comparaison avec les travaux existants

En plus des études sur la reconfiguration dans les systèmes répartis mentionnées dans les sections précédentes, nous comparons nos travaux avec d'autres environnements de reconfiguration.
Tout d'abord le *Advanced Manipulators Laboratory* propose avec Onika [8] et Chimera [20] un environnement de conception d'applications reconfigurables pour systèmes embarqués et temps réels. Cette solution vise à tirer pleinement avantage des architectures matérielles reconfigurables (bras robotique ayant plusieurs outils possibles, chacun associé à une fonctionnalité, donc à un module logiciel particulier). Ainsi elle propose de concevoir une application sous forme d'un agencement de composants communicants avec les autres composants et avec les ressources qui leur sont associées [19]. Les communications se font par l'intermédiaire de variables d'état globales dont chaque composant possède une copie locale, périodiquement mise à jour. Cette approche, très orientée robotique, ne répond pas à notre domaine applicatif d'une part parce qu'il n'a pas été possible d'identifier de composants génériques pour les logiciels de traitement scientifiques, d'autre part parce qu'elle suppose un système périodique, ce qui n'est pas le cas de Corot. En outre elle n'assure pas le dynamisme et les séquences de reconfiguration doivent être prédéterminées avant installation du logiciel sur la cible.
La mission Deep Space 1 de la NASA/JPL a montré avec succès la possibilité de contrôler une sonde par un logiciel à agent autonome embarqué, Remote Agent [12]. Ce système est basé sur la collaboration de plusieurs processus chargés de remplir les objectifs définis par l'opérateur au sol sous forme de buts généraux. Orienté objectif, Remote Agent se charge du détail du contrôle et des manoeuvres de la sonde, mais surtout il a pour rôle de réagir en temps réel et de façon autonome à toute condition exceptionnelle. Il gère notamment les diagnostics d'avaries, la reconfiguration du matériel en cas de défaillance d'un composant, la sélection et l'ordonnancement des tâches nécessaires à l'aboutissement de la mission, choisies parmi des bibliothèques prédéfinies. C'est en ce sens que Remote Agent implémente la reconfiguration. Celle-ci correspond à un grain bien supérieur à celui que nous utilisons. Par ailleurs, un tel système est extrêmenent gourmand en terme de ressources et ne permet pas l'ajout ou la modification de fonctionnalités à bord du satellite. Plutôt que concurrentes, nos approches sont donc complémentaires.

7. Conclusion

Nous avons proposé une approche apportant une plus grande flexibilité dans les logiciels multitâches embarqués sur satellites, basée sur un système transverse bord-sol cohérent. Ce système incorpore une spécification d'OS générique, implémentable sous forme d'un intergiciel et d'un interpréteur de bytecode pour la partie embarquée. La partie sol est composée d'une succession d'outils s'articulant autour d'un langage de configuration produisant automatiquement les séquences de commandes et les vérifications nécessaires.
L'implémentation que nous en avons faite montre de bonnes performances. Elle apporte un gain substantiel en termes de fiabilité et de sûreté dans la planification, la mise en oeuvre et la vérification des modifications semi-dynamiques des applications bord. En outre notre solution reste suffisamment générique pour pouvoir être généralisée à d'autres applications spatiales que Corot, la mission que nous avons utilisée comme cadre applicatif.
Outre l'outil de vérification structurelle de configuration et l'allocateur mémoire qu'il nous reste à finaliser, un certain nombre de tests de robustesse et de performances sont à mener avant de pouvoir définitivement conclure sur la pertinence de cette approche.

L'un des axes possibles de développement de cette approche est de proposer une solution générique, réutilisable, à base de machine virtuelle modulaire adaptable aux spécifications des différents types de logiciels embarqués en environnement contraint.

Bibliographie

1. Auvergne M., Baglin A., et al. – Du coeur des étoiles aux planètes habitables, les enjeux de Corot – Journal des Astronomes Français, N. 60, p. 27–34, 1999.
2. Barr M. – Architecturing Embedded Systems for Add-on Software Modules – Embedded Systems Programming, Sept 1999.
3. Bellissard L. – Construction et configuration d'applications réparties – PhD thesis, Institut National Polytechnique de Grenoble, ENSIMAG, 1997.
4. Cailliau D., Bellenger R. – The Corot instrument's software : towards intrinsically reconfigurable real-time embedded processing software in space-borne instruments – Proceedings of the 4th IEEE International Symposium on High Assurance System Engineering (HASE'99), Washington, DC., p. 75–80, Nov. 17-19, 1999.
5. De Palma N., Bellissard L., Riveill M. – Dynamic Reconfiguration of Agent-Based Application – Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal, 23-28 April 1999.
6. DeRemer F., Kron H. H. – Programming-in-the-large versus programming-in-the-small – IEEE Trans. on Software Engineering, Vol. SE-2, No. 2, p. 80-86, Jun. 1976.
7. Folliot B., Cailliau D., Piumarta I., Bellenger R. – PLERS : Plateforme logicielle embarquée reconfigurable pour satellites – Proceedings of the 12th Rencontres Francophones du Parallélisme, des Architectures et des Systèmes (RENPAR2000), Besançon, 19-22 Juin 2000, pp 191-196, Ed. Université de Franche-Comté, UFR des Sciences et Techniques, Besançon.
8. Gertz M. W., Stewart D. B., Kholsa P. K. – A human-macine interface for reconfigurable sensor-based control systems – Proceedings of the AIAA Conf. on Space Programs and Technologies, Huntsville, Al., Sept., 1993.
9. Katz D. S., Springer P. L. – Development of a Spaceborne Embedded Cluster – Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER2000), Technische University Chemnitz, Saxony, Germany, Nov. 28 - Dec 02, 2000.
10. Martinelli A., Torchia F. – The software maintenance in the BeppoSax scientific mission – Proceedings of the Data Systems in Aerospace Conference (DASIA'97), Sevilla, Spain, p.369–374, May 26-29, 1997.
11. Magee J., Kramer J., Sloman M. – Constructing distributed systems in Conic – IEEE Trans. on Software Engineering, Vol. 15, No. 6, pp 663-675, Jun. 1989.
12. Muscettola N. et al. – Remote Agent : to boldly go where no AI system has gone before – Artificial Intelligence, Vol. 103, No. 1-2, p 5-48, 1998.
13. Olive J. P. – Soho Recovery – Proceedings of the Flight Mechanics Conference, 1999.
14. Pellegrini M-C. – Reconfiguration d'application réparties : application au bus logiciel CORBA – mémoire de thèse de l'Institut National Polytechnique de Grenoble, 1999.
15. Purtilo J. M. – The Polylith software bus – Proceedings of ACM TOPLAS, Vol. 16, No. 1, p. 151-174, Jan. 1994.
16. Reinholtz K. – Applying simulation to the development of spacecraft flight software – Proceedings of the IEEE Aerospace Conference, Snomass, CO, Mar. 6-13, 1998.
17. Segal M. E., Frieder O. – On-the-fly program modification : systems for dynamic updating – IEEE Software, pp 53-65, March, 1993.
18. Statman J. L. – Optimizing the Galileo space communication link – TDA progress Report 42-116, Feb., 1994, NASA-JPL.
19. Stewart D. B. – Designing software components for real-time applications – Proceedings of 1999 Embedded Systems Conference, San Jose, CA, Sept. 1999.
20. Stewart D. B., Schmitz D. E., Kholsa P. K. – The Chimera II real-time operating system for advanced sensor-based control applications – IEEE Trans. on Systems, Man, and Cybernetics, Vol. 22, No. 6, 1992.