References

[1] **R. Airiau, J.M Berge, V.Olive, and J.Rouillard**,VHDL du langage à la modélisation. Polytechnic Presses and Universiter Romande and CNET-ENST , 1990.

[2] **H. Aida, J.A Goguen and J. Meseguer**, Compiling Concurrent Rewriting onto the Rewrite Rule Machine. Technical Report SRI-CSL-90-03R, Computer Science Laboratory, SRI International, Fabruary 1990, rev Dec 1990.

[3] **H. Aida, S. Leinwand and J. Meseguer**, Architectural Design of the Rewrite Rule Machine Ensemble. Technical Report SRI-CSL-90-17, Computer Science Laboratory, SRI International, Decembre 1990.

[4] **F. Biemans and P. Blouk**, On the Formal Specification and Verification of CIM. Architectures Using LOTOS, Computer in industry, vol7, n° 6, PAGES 491-504, 1986.

[5] **A.Cau, H.Zedan, N.Coleman and B.Moszkowski**, Using ITL and Tempura for Large-Scale Specification and Simulation. In Proc of the Parallel and Distributed Processing, pages 493-500, january 24-26, 1996.

[6] **M. Faci and L. Logrippo**, Specifying Hardware Systems in LOTOS. In Proc of the FFIP Conference on VHDL and their Application, pages 305-312, april 1993.

[7] **H.Garavel**, Compilation et Verification de programmes LOTOS. In PhD Thesis, University of Grenoble I, 1991.

[8] **J.A Goguen, C.Kirchner et J. Meseguer**, Concurent Term Rewriting as a model of computation. In R.Keller and J.Fasel, Editors, Proc Workshops on graph reduction, Santa Fe, New Mexico, pages 53-93, Springer Verglas , LNCS 279, 1987.

[9] **J.A Goguen, J. Meseguer, S. Leinwand, T. Winkler and H. Aida**, The Rewrite Rule Machine Project. Technical Report SRI -CSL-87-1, Computer Science Laboratory, SRI International, May 1987.

[10] **P. Lincoln, N.marti-Oliet, J. Meseguer and J. Ricciulli**, Compiling Rewriting onto SIMD, MIMD /SIMD Machine.

[11] **J.T Odonnell**, Hydra: Hardware Description in a functional language using recursion equations and high order combining forms. The Fusion of Hardware Design and Verification, IFIP 1988.

[12] **S.L.Pandey, K.R.Subramanian, and P.A.Wilsey**, A semantic Model of VHDL for Validing Rewriting Algebras. In Proc of EUROMICRO-22 Beyond 2000: Hardware and Software Design Strategies, pages 167-176, September 2-5, 1996.

[13] **R.O Sinnott**, The Formally Specifying in LOTOS of Electronic Components. M.Sc. Thesis, University of Stirling, UK, 1993.

[14] **K. J Turner and R.O Sinnott**, DILL : Specifying Digital Logic in LOTOS. Technical Report, Department of Computer Science, Stirling, Scotland ,1993.

[15] **D. Cyrluk and P. Narendran**, Groud Temporal Logic: A Logic for Hardware Verification. In 6th International Conference on Computer Aided Verification, Stanford, 1994.

Un Nouveau Modèle pour la Restructuration de Programmes Temps Réel

Mohamed-Tahar Kimour*, Said Ghoul**
* Laboratoire de Méthodologie de Résolution de Problèmes
Institut d'informatique, Université d'Annaba
BP12, Annaba 23000, Algérie.
Fax: (213)-8-87-27-56, e-Mail: kimour@ yahoo.com

**Philadelphia University, Computer Science & Information Systems,
Sciences College, PO Box 1101 Sweilah, Amman, Jordan
e-Mail: philad@go.com.jo

Résumé: La restructuration de programmes consiste à apporter des modifications sur la structure interne de ces programmes, pour différents buts tels que la facilité de compréhension et de maintenance, l'optimisation, l'ordonnancement de tâches, la facilité de conversion d'un langage à un autre, ou la détermination de modules réutilisables. Dans ce papier nous présentons un modèle de restructuration de programmes temps réel, qui peut servir particulièrement à l'amélioration de l'ordonnançabilité des applications de ces domaines ou à la compréhension des comportements de programmes. En effet, les programmes associés à des systèmes temps réel doivent être logiquement corrects et doivent s'exécuter sans fautes temporelles. Afin de satisfaire leurs échéances, les tâches de ces systèmes sont ordonnancées par des techniques adéquates. Dans le cas par exemple, où un ensemble de tâches serait non ordonnançable, le programmeur procède à des opérations manuelles de modification, de réglage et de restructuration de l'application, afin de la rendre ordonnançable. De telles opérations sont lentes, coûteuses, et non contrôlables. Les solutions existantes dans ce domaine, sont de nature à améliorer l'ordonnançabilité d'une tâche. Basées sur la décomposition de tâches, elles utilisent un modèle de dépendance de programmes monolithiques. Ce modèle est imprécis et ne permet pas de représenter de façon explicite les contraintes de temps sur les tâches temps réel.
A base d'un modèle de dépendance plus précis, notre approche de restructuration traite des programmes plus complexes, qui comprennent des constructions de contraintes de temps et de concurrence.

Mots-Clés: Systèmes Temps Réel, Contraintes de Temps, Ordonnancement de Tâches, Restructuration de Programmes, Tranches de Programmes.

1. INTRODUCTION

Dans les systèmes temps réels de contrôle et de supervision, on doit considérer non seulement le comportement fonctionnel, mais aussi et surtout le comportement temporel. Dans ces systèmes, l'exécution d'une tâche se fait souvent périodiquement, toutes les T unités de temps (période). La principale contrainte est que l'invocation d'une tâche doit être effectuée durant D unités de temps(échéance), depuis son passage à l'état prêt. Pour satisfaire leurs échéances, les tâches doivent être ordonnancées convenablement. Mais l'évolution des systèmes temps réel, peut entraîner des changements de caractéristiques de ces tâches (échéances, temps d'exécution,...) [POS92], ou des surcharges de ressources. Du fait de leur évolution, des programmes tendent à devenir peu structurés. En plus du changement de leurs caractéristiques, ils deviennent non conformes aux standards, difficiles à comprendre.
La restructuration est par conséquent une option importante pour contrôler les coûts de maintenance du logiciel, et améliorer sa qualité. Il existe d'autres raisons pour la restructuration telles que: faciliter les tests, réduction potentielle de complexité de programmes, mise en place de standards pour les structures de programmes, étendre la durée de vie des programmes en maintenant une flexibilité à travers une bonne structure, préparer les programmes pour faire l'objet d'entrée à des outils (parallélisme, test, etc.), préparer les programmes pour la conversion, les préparer pour ajouter de nouvelles caractéristiques au logiciel.

Toutefois, la restructuration est aussi pratiquée durant la phase de développement de logiciel. Particulièrement, lorsque dans un environnement de développement où des méthodes rigoureuses de programmation et de structuration de code ne sont pas suivies. En fait, la restructuration du logiciel n'est pas une fin en soi. Elle est un moyen pour réaliser des objectifs fixés. Par exemple, pour la compréhension, on restructure afin de pouvoir entre autres:

1- Construire un style de code: Cette approche de restructuration qui modifie le code pour le rendre facile à comprendre, sans altérer ces dépendances de contrôle ou de données.

2- formater le code: c'est une technique qui restructure le texte du programme pour respecter l'indentation, laisser l'espace entre des sections, une instruction par ligne, etc.

3- Rendre le flux de contrôle du programme facile à suivre, élimination des goto, etc.

La restructuration est souvent basée sur des modèles de comportement de programmes [HOR90, BIN93, BEN97], sur la base desquels on effectue des décompositions et recomposition de fragment de code. Dans une perspective d'amélioration de l'ordonnançabilité de tâches temps réel, Gerber et Hong [GER97] ont proposé un modèle de décomposition de programmes temps réel utilisant le concept de tranche. Une tranche de programme, par rapport à un point *p* et une variable v, consiste en des instructions du programme, *qui peuvent affecter la valeur de* v, au point *p* [WEI84]. Son comportement est identique au programme initial par rapport à un critère donné. Nous pouvons par exemple, isoler une tranche par rapport à une instruction critique dans le programme, une instruction d'émission d'une commande vers un actionneur ou une instruction d'alarme dans une tâche de contrôle.

Cependant, cette approche porte uniquement sur des cas simples de programmes monolithiques et sans prendre en compte les constructions répétitives. Ce qui est restrictif par rapport à la réalité temps réel, où l'on trouve souvent des structures de programmes multi-procéduraux [DUE91]. En outre, le critère d'extraction des tranches produit des fragments imprécis et ceci est dû au fait que l'élément d'extraction utilisé (instruction) est d'une granularité inadéquate.

Dans un précédent travail [KIM99], nous avons proposé une solution de restructuration de programmes monolithiques à base d'une décomposition par tranche, mais qui prend en compte les contructions répétitives qui prennent la plus grande partie des temps d'exécution des programmes. La restructuration est faite non pas sur une tâche isolée, mais elle est intégrée à une technique d'analyse d'ordonnançabilité du code source qui peut représenter plusieurs tâches temps réel.

Nous proposons, dans cet article, un nouveau modèle de dépendance où la granularité est au niveau de l'action sur la variable, et qui opère sur des programmes temps réels multiprocéduraux. Par conséquent, il peut représenter des constructions de langages plus généralisées que celles traitées dans [GER97, KIM99] et permet d'obtenir des tranches plus précises. En outre, notre modèle de dépendance offre des mécanismes pour représenter les contraintes de temps qui sont essentielles pour tout système temps réel.

La section suivante est consacrée à l'analyse des approches existantes et la mise en relief de leurs insuffisances. La section 3 est consacrée à l'approche de restructuration de programme temps réel. La section 4 détaille le modèle de comportement fonctionnel et temporel et présente les extensions que nous avons proposées. A la fin, nous terminons par une conclusion.

2. TRAVAUX ACTUELS

Les algorithmes d'ordonnancement se basent presque tous sur le fait que toutes les tâches doivent s'exécuter à l'intérieur de leurs périodes. Si l'ensemble de tâches est non ordonnançable, ces algorithmes n'apportent aucune aide aux programmeurs pour résoudre ce problème. Ils n'offrent aucun moyen pour assister les programmeurs dans la modification du système ou son réajustement pour rendre ordonnançable un ensemble de tâches. Ceci est dû au fait que la tâche est considérée comme étant un bloc non interprétable.

Dans une perspective d'amélioration de l'ordonnançabilité de tâches temps réel, Gerber et Hong [GER97] ont proposé un modèle de décomposition de programmes temps réel utilisant le concept de tranche. Dans cette approche, un programme temps réel est modélisé par un Graphe de Dépendance de Programme (GDP). Dans ce type de graphe les nœuds représentent les instructions et les arcs les liens sémantiques (flux de données et de contrôle). L'extraction d'une tranche revient à un parcours dans ce

graphe à partir d'un sommet constituant le critère d'extraction. Utilisant le paradigme des événements dans les systèmes temps réels, la tranche temporelle est déterminée par rapport aux instructions qui ont un effet visible du monde extérieur. On peut en citer les instructions *Input* et *Output*. Les tranches de tels critères formeront le fragment temporel.

Par exemple, le programme de la figure 1 représente le code source d'une tache temps réel. Ce programme est représenté par le GDP de la figure 2. Pour déterminer le fragment temporel, on determine le critère d'extraction d'une tranche. Ce critère est donné par l'instruction *Output ('L10: **Output(Actuator, cmd))*** qui a évidemment un effet visible sur le monde extérieur. Sur le GDP et partant du nœud correspondant à cette instruction, on effectue un parcours arrière jusqu'au nœud Entry [BOU96]. Toutes les instructions marquées formeront le fragment temporel (partie hachurée de la Figure 2). Le fragment non temporel est constitué des instructions restantes. Le code non observable est déplacé pour écourter son temps d'exécution.

```
    every 15 ms
    {
L1: input(Sensor, &data);
L2: if (!null(data))
    {
L3:   t1 = F1(state);
L4:   t2 = F2(state,t4);
L5:   t3 = F3(data);
L6:   t4 = F4(t1,data);
L7:   state = F5(t1, t2, t3);
L8:   cmd= F6(t1,t4);
L9:  Output(Actuator, cmd);
    }
L10: status_dump("logfile", cmd, state);
}
```

Figure 1: programme source de la tâche τ_3

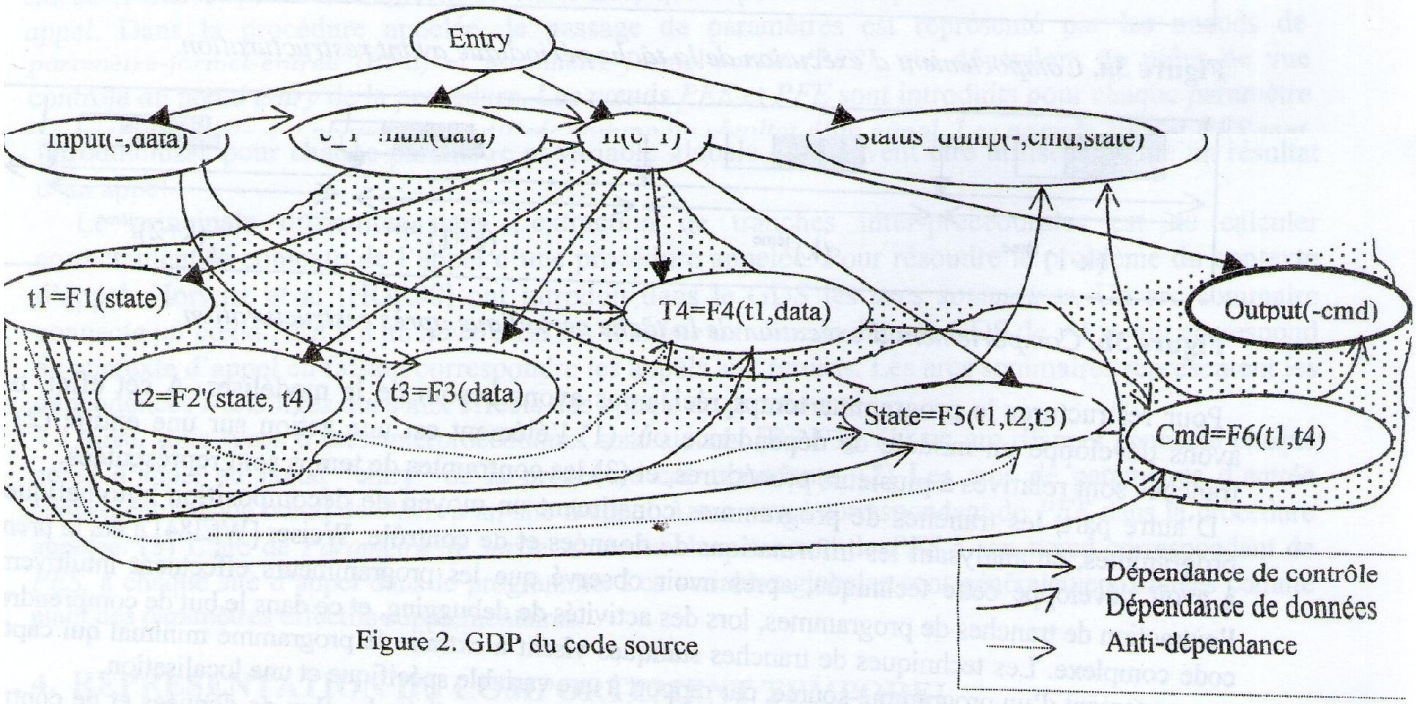

Figure 2. GDP du code source

Cependant, dans [GER97, KIM99] les modèles proposés ne permettent de traiter que des programmes monolithiques. Ils ne comportent pas de mécanismes explicites pour représenter les contraintes de temps, ni au niveau tâche, ni au niveau instruction. Par exemple, l'instruction «Every 40 ms» n'est pas représentée. Dans [GER97], les constructions répétitives ne sont pas traitées alors que c'est là où l'on consomme le plus de temps d'exécution. De plus, l'approche est limitée à l'ordonnancement de code intra-tâches et ne traite pas les types globaux de cas d'ordonnancement temps réel.

3. APPROCHE DE RESTRUCTURATION

La restructuration est un moyen de changer la structure interne d'u n programme pour différents objectifs: la maintenance (faciliter la compréhension), l'optimiosation, l''ordonnancement (améliorer l'ordonnançabilité), faciliter les tests, le déboggage, la conversion de programmes d'un langage à un autre, etc.

Par exemple dans le cas de l'amélioration de l'ordonnançabilités des tâches temps réel, à travers le changement de la structure interne d'un programme nous avons montré dans nos travaux précedents que nous pouvons obtenir des échéances flexibles de tâches, car l'ordonnançabilité est améliorée à mesure que l'on augmente la flexibilité des échéances. Nous pouvons réaliser la flexibilité en restructurant la tâche pour obtenir deux fragments. Le premier est un fagment temps contraint (noté FT) qui doit satisfaire son échéance. Le second est non temps contraint (noté FnT), qui correspond à un calcul interne. Ce dernier fragment peut tolérer un certain retard. Ensuite, les fragments seront séquentiellement recomposés, pour constituer un nouveau programme de la tâche en question, qui est sémantiquement équivalent au programme initial.

La Figure 3a représente le comportement d'exécution d'une tâche périodique. Les sections hachurées dans les figures 3a et 3b représentent les composants temps contraint du programme de la tâche. A la $(k+1)^{ième}$ période, l'exécution de cette tâche dépasse son chéance et devient donc, non ordonnançable (Figure 3a). Toutefois, dans le cas où les actions temps contraint s'effectuent à l'intérieur de la fenêtre de temps prescrite, l'exécution de la tâche entière est acceptable (Figure 3b).

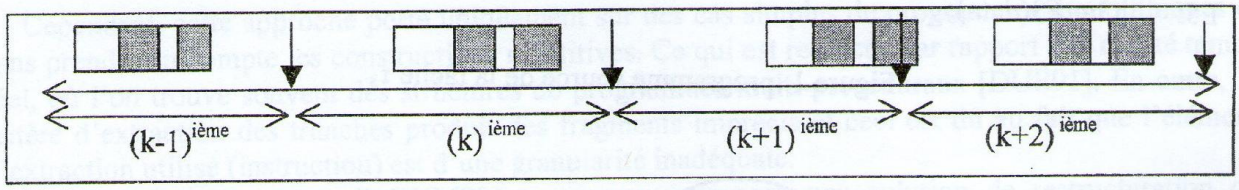

Figure 3a. *Comportement d'exécution de la tâche périodique avant restructuration.*

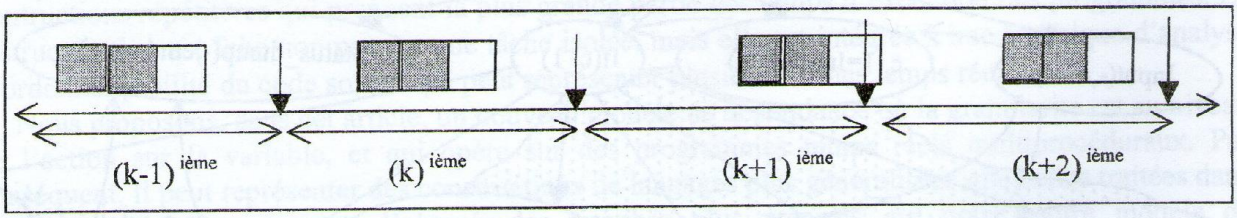

Figure 3b. *Comportement d'exécution de la tâche périodique après restructuration*

Pour restructurer le programme temps réel, nous avons besoin de le modéliser. A cet effet, nous avons développé un modèle de dépendance où: (1) l'élément est une action sur une entité, (2) les tranches sont relatives à plusieurs procédures, et (3) les contraintes de temps sont représentées.

D'autre part, les tranches de programmes constituent un moyen de décomposition automatique de programmes, en analysant les informations de données et de contrôle. Weiser [WEI84] a été le premier à avoir développé cette technique, après avoir observé que les programmeurs effectuent intuitivement l'extraction de tranches de programmes, lors des activités de debugging, et ce dans le but de comprendre un code complexe. Les techniques de tranches statiques visent à extraire le programme minimal qui capte le comportement d'un programme-source, par rapport à une variable spécifique et une localisation.

Par conséquent, nous avons besoin de représenter de façon explicite les flux de données et de contrôle. Nous pouvons le faire au moyen d'un graphe de dépendance, qui réduit l'extraction de tranches à un problème de parcourt de graphe [FER87]. Une tranche est déterminée par un parcourt arrière en empruntant les arcs représentant les flux de contrôle et de données. La tranche correspond au sous-graphe contenant les nœuds marqués et leurs arcs.

Un programme monolithique (programme sans les appels de procéduress) est représenté par un Graphe de Dépendance de Procédure (GDP). Un programme contenant une collection de procédures est représenté par un Graphe de Dépendance de Système(GDS) [BIN93, BEN97].

3.1. Modélisation du Comportement Fonctionnel

Un GDP est composé de nœuds et d'arcs représentant respectivement les différentes actions du programmes et leurs dépendances. A l'exception des actions d'appel de procédures, un arc représents une action d'affectation, de "input" et de "output", et de predicat de condition (if) et de boucle "while". De plus, on ajoute un arc spécifique appelé *entry*, et un arc de séquencement pour chaque variable qui peut être utilisée avant sa définition. La source de l'arc de *dépendance de contrôle* est soit un arc *entry*, un arc *prédicat*, ou un arc *d'appel*. Chaque arc est étiqueté soit par *vrai* ou par *faux*. Un arc de *dépendance de contrôle*, du nœud *v* au nœud *u*, signifie que durant l'exécution, à chaque fois que le prédicat représenté par *v* est évalué et sa valeur est égale à la valeur de l'étiquette sur l'arc menant à *u*, alors le composant du programme représenté par *u* sera éventuellement exécuté, pourvu que le programme se termine normalement.

Un arc de *dependance de données* du nœud *v* au nœud *u* signifie que le comportement du système peut changer si l'ordre relatif des composants représentés par *v* et *u* est inversé. Nous distinguons trois types de dépendances de données, *dépendance de flux, dépendance de séquencement* et *anti-dépendance*. Un arc *de dépendance de flux* connecte le nœud *v* qui représente une affectation à une variable *x*, au nœud *u* qui représente une utilisation de cette variable. Un arc de *dépendance de séquencement* connecte le nœud *v* qui représente une déclaration d'une variable *x*, au nœud *u* qui représente une initialisation de cette variable. Un arc *d'anti-dépendance* connecte le nœud *v* qui représente une utilisation d'une variable *x*, au nœud *u* qui représente une affectation à cette variable.

Pour représenter les dépendances d'un programme multi-procédural, nous utilisons un Graph de Dépendence de Système (GDS) [BIN93, BEN97]. Il est composé d'un un ensemble de GDPs (figure 4), connectés par des arcs qui symbolisent les dépendances d'*appels* (Figure 5). Une instruction d'appel est représentée par un nœud d'appel et quatre types de nœuds paramètres qui représentent le passage de paramètres. A l'appel, le passage de paramètre est représenté par les nœuds *paramètre-effectif-entrée (PEE)* et *paramètre-effectif-sortie (PES)*, qui dépendent du point de vue contrôle, du nœud *appel*. Dans la procédure appelée, le passage de paramètres est représenté par les noeuds de *paramètre-formel-entrée (PFE)* et *paramètre-formel-sortie (PFS)*, qui dépendent du point de vue contrôle du nœud *entry* de la procédure. Les nœuds *PEE* et *PFE* sont introduits pour chaque paramètre et variable globale qui peuvent être utilisés comme un résultat d'un appel. Les noeuds *PFS* et *PES* sont introduicluses pour chaque paramètre et variable globale qui peuvent être utilisés comme un résultat d'un appel

Le principale difficultée dans l'extraction de tranches inter-procédurales est de calculer correctement le contexte de l'appel d'une procédure appelée. Pour résoudre le problème du contexte d'appel, Horwitz et al [HOR90] ont introduit dans le GDS les *arcs sommaires*. Un arc sommaire connecte un nœud *PEE v* à un *nœud PES u* s'il existe un chemin dans le GDS de *v* à *u*, qui correspond au contexte d'appel en faisant correspondre les appels aux returns. Les arcs sommaires représentent les dépendances *transitives* dues aux effects des appels de procédures.

Ainsi, trois types d'arcs interprocéduraux sont ajoutés [BIN93]: (1) un arc d'appel connecte chaque nœud d'appel au nœud "entry" de la procédure correspondante. (2) Les arcs de paramètres d'entrée connectent chaque nœud de *PEE* à un site d'appel, à son nœud correspondant de *PFE* dans la procédure appelée. (3) L'arc de *Paramètre de sortie* connecte chaque nœud de *PFS* à son nœud correspondant de *PES*, à chaque site d'appel dans le programme. Les variables globales sont généralement traitées comme étant des paramètres effectifs supplémentaires.

4. REPRESENTATION DU COMPORTEMENT TEMPOREL

En examinant les différentes constructions temporelles dans les langages temps réel usuels, nous constatons que les contraintes de temps sont exprimées de façon explicite ou implicite au niveau du code source (langage ADA, concurrent C, FLEX) ou au niveau assembleur [KLI86, KEN91, RAM91, SCH92, CHU95].

L'exemple suivant (Figure 4) est un fragment d'un programme de contrôle, noté de façon informelle, qui peut facilement être trouvé dans une application réelle. Cet exemple montre que les contraintes de temps, sont exprimées par des relations temporelles entre les actions ou blocs d'actions.

i_1 : Charger r_1, capt1 /*Valeur chargée à partir du capteur1*/
@ i_1 doit s'exécuter au plus tard 1µs après i_3
i_2 : Stocker act_1, r_1 /*Valeur envoyée à l'actionneur$_1$*/
@ i_2 doit s'exécuter au plus tard 1 µs après i_5
i_3 : Charger r_2, $capt_2$ /* Valeur chargée à partir du capteur$_2$*/
@ i_3 doit s'exécuter au plus tard 1µs après i_1
i_4 : Ajouter r_3, r_1, r_2 /* Ajouter les entrées des capteurs r_1, r_2 dans r_3*/
i_5: Stocker act_2, r_3 /*Le résultat est envoyé à l'actionneur2*/

Figure 4: expression formelle d'un exemple de fragment de code temps réel.

Les contraintes de temps sont donc des dépendances entre les actions ou blocs d'actions. A cet effet, les langages comprennent des constructions pour les représenter comme par exemple, l'instruction delay() dans ADA, ou l'instruction cycle() et autres expressions temprelles d'actions, ajoutées au langage C par Chung et Dietz [KLI86, KEN91, RAM91, SCH92, HON93, Chu95].

Par ailleurs, et dans le contexte de système temps réel de contrôle et de supervision, une action dans un programme peut être classée comme action vue externe (AVE) ou action vue interne (AVI), selon l'effet de l'action. L'effet d'une AVI est limitée à un calcul interne, mais une AVE change le statut de l'environnement de contrôle. Par exemple, les actions sur les variables définies "volatiles" dans le langage C ou les commandes de contrôle de robot utilisant le langage ADA sont des AVIs [KLI86, SCH92], et leurs exécutions doivent satisfaire les contraintes de temps spécifiées. Etant donné que les AVEs peuvent dépendre des valeurs calculées par les AVIs, de telles dépendances impliquent aussi de relatives contraintes de temps (ordre d'exécution) devant être préservées.

Le problème de représentation des contraintes de temps est bien étudié au niveau des modèles de spécification des systèmes temps réel tels que dans les réseaux de petri, la logique temps réel, etc [GHE90, GHE91]. En se basant sur ces travaux et en examinant les différentes contsructions temps réel et de concurrence, existantes dans les langages de programmation du style Ada, nous proposons de représenter une contrainte de temps entre deux actions ou blocs d'actions par le quadriplet $<A_s, \eta, \delta, A_e>$ où: *As:* Action source, *Ae:* Action cible, η: Opérateur relationnel, et δ: déplacement temporel. η peut prendre l'une des valeurs suivantes: Avant (<), Après (>), Concurrent (=), ou Exclusif (≠).

Ainsi, et indépendamment de tout langage de programmation, nous représentons toute relation temporelle η, entre *As* et *Ae*,comme suit:

1- $Ae<As+\delta$ (*Ae* doit être lancée au plus tard δ après *As*),
2- $Ae>As+\delta$ (*Ae* doit être lancée au plus tôt δ après *As*),
3- $Ae=As+\delta$ (*Ae* doit être lancée à exactement δ après *As*),
4- $Ae\neq As+\delta$ (*Ae* ne doit pas être lancée à δ après *As*).

Ces relations peuvent être déterminées à partire seyulement du code source. Elles y peuvent être exprimées soit explicitement par des primitives telles que wait(), delay, cycle(), ou implicitement par des emplacement adéquats des portions de code.

Pour modéliser le comportement d'un programme temps réel, nous devons donc modéliser tous ces types de dépendances (données, contrôle, temps). Pour cela, nous définissons le graphe $G=(\Gamma,\Theta)$, comme étant un graphe orienté qui consiste en un ensemble de nœuds Γ et un ensemble d'arcs Θ. Les nœuds sont associés aux actions sur les entités, et les arcs sont associés aux différents types de dépendances et aux contraintes de temps. Dans un programme monolithique, un arc représente l'un des type de dépendances suivants:

- Une dépendance de données (de flux, anti-dépendance, ou de sortie)
- Une dépendance de contrôle,
- Une dépendance de séquencement (par exemple, la déclaration d'une variale précède son initialisation).

Pour représenter les contraintes de temps, nous généralisons la notion de dépendance [BEN97, BOU98] pour qu'elle intègre la relation temps. Cette dernière peut prendre l'une des quatre valeurs de l'opérateur η, défini précédemment.
Toutefois, et comme nous pouvons facilement trouver entre deux actions, l'un des trois types de dépendances et une relation temporelle à la fois (figure 5), et afin de simplifier le graphe de représentation, nous proposons que la contrainte de temps soit exprimée par une étiquette sur l'arc reliant deux nœuds qui représentent les deux actions.
De cette façon, l'arc peut représenter la dépendance de donnée, de contrôle, ou de séquencement et l'étiquette représente la dépendance de temps. L'étiquette est composée de deux valeurs (η, δ). η peut être égale à **null**, pour représenter seulement les dépendances de données, de contrôle ou de séquencement, en l'absence de dépendance de temps. De même, si entre deux actions il n'existe que des dépendances de temps (en l'absence des autres types de dépendance), les nœuds représentant ces actions seront reliés par un arc de séquencement augmenté de l'étiquette de la relation de temps.

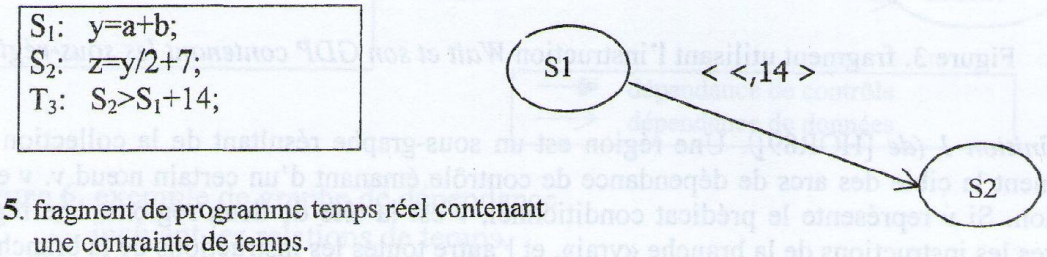

Figure 5. fragment de programme temps réel contenant une contrainte de temps.

Entre l'instruction S_1 et S_2 (Figure 5), il existe une dépendance de donnée et une relation temporelle. Cette relation temporelle est exprimée par l'expression temporelle de la ligne T_3. Graphiquement, entre le nœud S_1 et le nœud S_2 il existe un arc de type flux de donnée, augmenté du biplet < <,14>, pour exprimer aussi la dépendance de temps.

De cette façon, toute contrainte entre actions peut être (spécifiée par notre modèle - même les contraintes qui ne peuvent pas être satisfaites par un ordonnancement séquentiel (c'est-à-dire contraintes qui impliquent une exécution parallèle).

En outre, il est significatif que le simple ordre (précédence ou séquencement) puisse être représenté sans introduire aucun type supplémentaire. Chaque contrainte de précédence de la forme "x utilise le résultat de y" peut être prise comme une "contrainte après"; (x est lancée au plus tôt 0 après y). Ainsi, les contraintes de temps et de précédence peuvent être exprimées en utilisant la même syntaxe.

4.1 Représentation de l'instruction d'attente:

L'instruction d'attente (wait(), delay(),...) est utilisée dans presque tous les langages de programmation. Elle est particulièrement utilisée dans les langages temps réel pour représenter des contraintes de temps. C'est le cas du langage Ada ou RT-Euclid [KLI86,SCH92] par exemple. Elle indique au processeur d'attendre *t* unités de temps (*t* est un paramètre donné) avant de continuer l'exécution de la suite des instructions du programme. Dans le système temps réel CHIMERA [SKE97], l'instruction wait(*d*) a pour effet :

1- Le déclenchement d'un temporisateur *T*, initialisé à la valeur de *d*.
2- Le temporisateur est décrémenté à chaque tick de l'horloge.
3- Dès que le temporisateur atteint la valeur 0, l'exécution du bloc suivant est lancée.

En fait, l'instruction d'attente exprime une relation temporelle non seulement entre deux instructions, mais entre deux blocs d'instructions. Considérons l'exemple de la figure 3, composé des instructions S_1, S_2, wait(d), S_3, et S_4:
L'exécution du bloc $\{S_3, S_4\}$ est lancée après une attente de *d* unités de temps après la fin de l'exécution du bloc $\{S_1, S_2\}$. Cette instruction introduit donc une relation de temps entre le premier bloc et le deuxième bloc. Pour représenter graphiquement cette relation de temps, nous utilisons la notion de région.

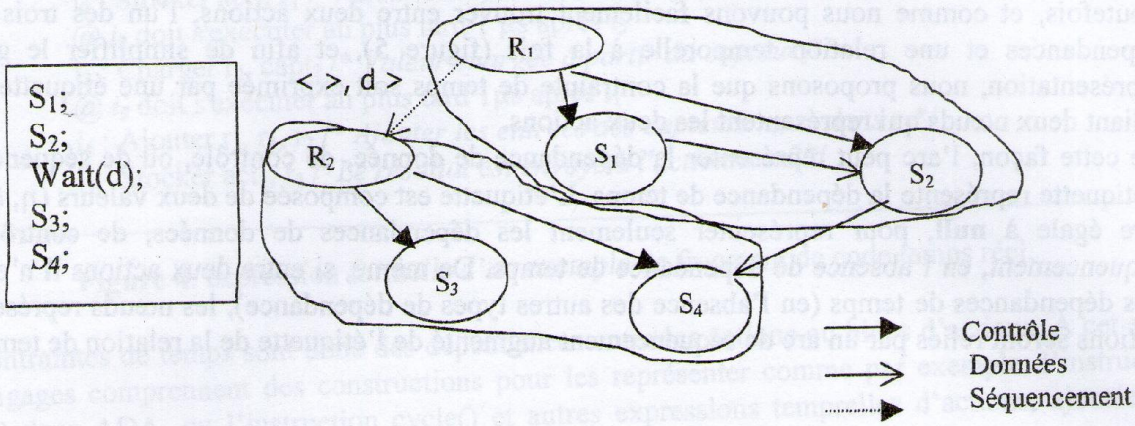

Figure 3. fragment utilisant l'instruction *Wait et son GDP contenant les sous-régions*.

Définition 1 (de [HOR89]*).* Une région est un sous-graphe résultant de la collection de nœuds qui forment la cible des arcs de dépendance de contrôle émanant d'un certain nœud *v*. *v* est la tête de la région. Si *v* représente le prédicat conditionnel, *v* est la tête de deux régions, une région comprend toutes les instructions de la branche «vrai», et l'autre toutes les instructions de la branche «faux».

Etant donné que l'instruction d'attente divise l'ensemble des nœuds de la région auquelle elle appartient en deux parties dépendantes par une relation de temps, nous insérons un nœud spécial pour chaque partie et qui sera le nœud tête de la sous-région définie déterminée par cette partie. Ce nœud de tête de la sous-région sera relié à chaque nœud de la partie par un arc de contrôle. Supposons que dans l'exemple de la figure 3, nous avons seulement les dépendance de données <S_1, S_2> et <S_2, S_4>. L'instruction *Wait(d)* introduit une relation temporelle entre les deux sous-régions R_1 (S_1, S_2) et R_2 (S_3, S_4). Cette relation peut être exprimée par $R_1 > R_2 + d$. La sous-région R_2, formée par S_3 et S_4 sera exécutée après attente de *d* unités de temps après l'exécution de R_1, formée par S_1 et S_2.

Définition 2. Une sous-région est définie par rapport à une région, par un nœud inséré dans la région (nœud de tête de la sous-région) et un sous-ensemble de nœuds de cette région.

Dans le GDP du fragment de programme (figure 6), nous insérons dans le graphe les nœuds R_1 et R_2, entre lesquelles l'arc représente la relation de temps introduite par l'instruction d'attente.

4.2 Représenation du bloc cyclique

Un bloc cyclique d'action correspond aux évènements périodiques, phénomènes que nous trouvons souvent dans les systèmes temps réel. Considérons l'exemple légèrement modifié, tiré de [CHU95]. **cycle** est un mot–clé qui indique que le bloc suivant entre { et } est un bloc cyclique. Dans Ada, on utilise la clause loop, pour indiquer un bloc cyclique. Un bloc cyclique est un ensemble d'actions (bloc d'actions) qui s'exécute répétitivement toutes les δ unités de temps (40 µs dans l'exemple). A la ligne 4, l'action *input* est étiquetée par la variable temporelle *tv1*. A la ligne 5, l'action d'affectation est étiquetée par *tv2*. De cette façon, nous pouvons représenter la relation temporelle entre ces deux actions (ligne 6).

La ligne 6 indique que l'action *tv1* doit être lancée au plus tard à 14µs après la fin de *tv2*.

La racine du graphe (Figure 6) est le nœud *entry*. L'instruction *cycle* est représentée par un nœud qui est relié à la racine par un arc représentant un flux de contrôle.

Cette instruction contrôle un bloc cyclique. Chaque nœud de chaque instruction de ce bloc est donc relié au nœud cycle, par un arc représentant le flux de contrôle.

Une contrainte de temps (période) est représentée par un arc, exprimant une relation réflexive sur le nœud "cycle". Cette relation réflexive représente en fait, une relation temporelle entre les invocations (instances) i et i+1 du bloc. L'invocation i+1 doit se faire au plus tard 40 µs après l'invocation i du bloc. L'arc représentant la relation réflexive est donc un arc de séquencement augmenté du biplet < >,40 >. Une dépendance de données existe entre l'instruction à la ligne 5 et l'instruction à la ligne 3. Un arc de

dépendance de données relie donc les nœuds correspondants. Une dépendance de données et de temps existe entre l'instruction à la ligne 4 et l'instruction à la ligne 5. Notons que l'ensemble des nœuds atteints par les arcs de contrôle provenant du nœud «cycle», forme une région dont la tête est le nœud «cycle».

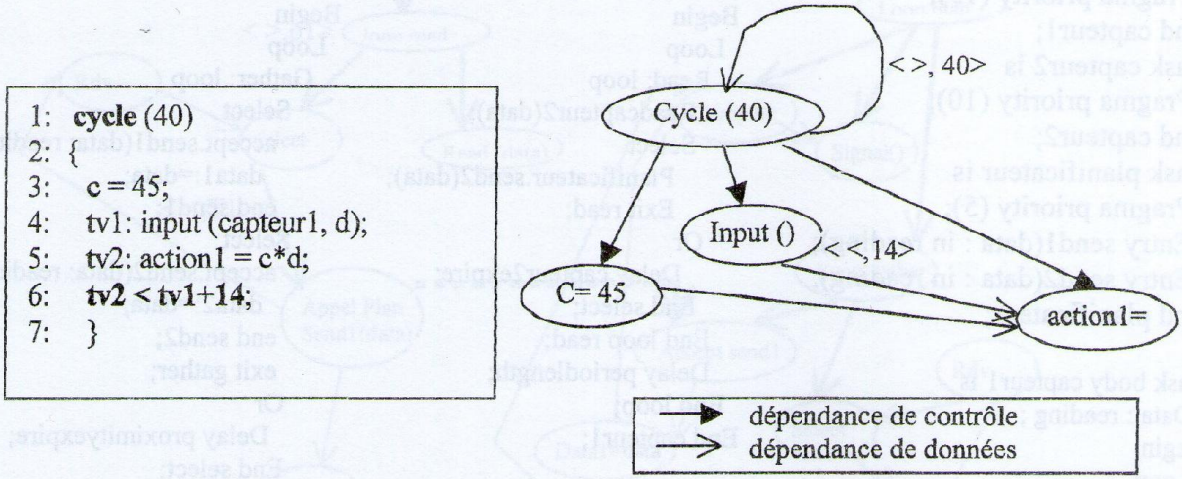

Figure 6. exemple de graphe de dépendance incluant les relations de temps.

4.3 Application sur Ada

Les logiciel de contrôle des systèmes concurrents et temps réel sont souvent écrits en Ada (COR96). En fait, Ada a été créé spécifiquement pour ce domaine d'applications (système de pilotage d'un avion, bateaux, missiles, etc.). Il contient des instructions de créations de multiples threads de contrôle (tâches) et de contrôle de leur exécution.

Il supporte la concurrence au niveau du code source. Deux tâches Ada peuvent synchroniser via un RendezVous. C'est une communication de synchronisation dans laquelle une tâches execute une instruction d'appel d'entrée (entry call), pour appeler une entrée nommée d'une autre tâche, qui peut accepter l'appel (accept) en exécutant une instruction d'acceptation (accept) qui nomme l'entrée. Une instruction select permet à une tâche d'attendre un appel d'entrée à l'un des points d'entrées (il peut y en avoir plusieurs). On peut borner le temps que la tâche met pour attendre un rendezvous. L'instruction delay permet à une tâche de se bloquer pour une durée spécifiée.

***Un exemple de robotique*:**

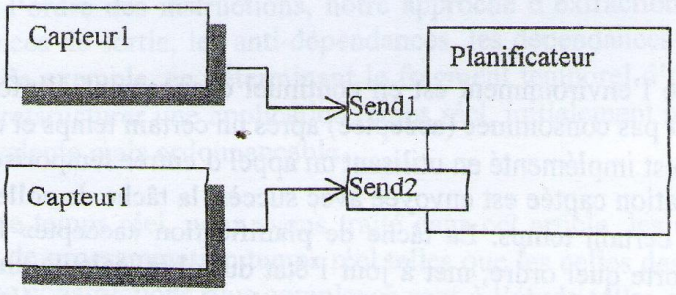

Figure 7. Structure de l'exemple de robot.

Il s'agit d'un contrôleur de robot décrit dans [COR96]. Le programme collecte des informations par deux capteurs et les utilisent pour mettre à jour l'état courant du robot et calculer la commande à lui envoyer périodiquement. Il y a donc deux tâches de capteurs et une tâches de planificateur. L'architecture de ce système est donnée à la figure 7 et son code source à la figure 8. La tâche i (au

niveau capteir i=1, 2) capte l'information et l'envoie à la tâche planificateur via un rendezvous au point d'entrée send$_i$ de la tâche planificateur.

```
Task capteur1 is
  Pragma priority (15);
End capteur1;
Task capteur2 is
  Pragma priority (10);
End capteur2;
Task planificateur is
  Pragma priority (5);
  Entry send1(data : in reading);
  Entry send2(data : in reading);
End planificateur;

Task body capteur1 is
  Data : reading ;
Begin
  Loop
    Read: loop
      Readcapteur1(data);
      Select
        Planificateur.send1(data);
        Exit read;
    Or
      Delay capteur1expire;
     End select;
    End loop read;
    Delay periodlength;
  End loop;
End capteur1;
```

```
Task body capteur2 is
  Data : reading ;
Begin
  Loop
    Read: loop
      Readcapteur2(data);
      Select
        Planificateur.send2(data);
        Exit read;
      Or
        Delay capteur2expire;
      End select;
    End loop read;
    Delay periodlength;
  End loop;
End capteur1;
```

```
Task body planificateur is
  Data1, data2 : reading;
  Com: command;
Begin
  Loop
    Gather: loop
      Select
        accept.send1(data: reading);
          data1:=data;
        end send1;
      Select
        accept.send2(data: reading);
          data2:=data;
        end send2;
        exit gather;
      Or
        Delay proximityexpire;
      End select;
    End loop Gather;
    Compute(data1, data2, com);
    Signal(com);
  End loop;
End planificateur;
```

Etant donné que l'environnment est en continuel changement, l'information captée deviendra non valide si elle n'est pas consommée (acceptée) après un certain temps et une nouvelle information devra être captée. Ceci est implémenté en utilisant un appel d'entrée temporisé au point d'entrée send$_i$.
Lorsque l'information captée est envoyée avec succès, la tâche de collecte se met à l'état dormant (se bloque) pour un certain temps. La tâche de planification «accepte» l'information lue par les deux capteurs à n'importe quel ordre, met à jour l'état du robot, calcule une commande et la lui envoie. Une contrainte de proximité est nécessaire entre les deux instants d'obtention des informations des deux capteurs afin que la corrélation soit significative. L'intervalle formé par ces deux instant doit être donc borné. Si la tâche de planification reçoit l'information du premier capteur et ne reçoit l'information du deuxième, au bout d'un certain temps, elle doit rejeter l'information reçue. L'ordre des priorités des trois tâches, de la plus élevée à la plus basse est le suivant: Capteur1, Capteur2, Planificateur.

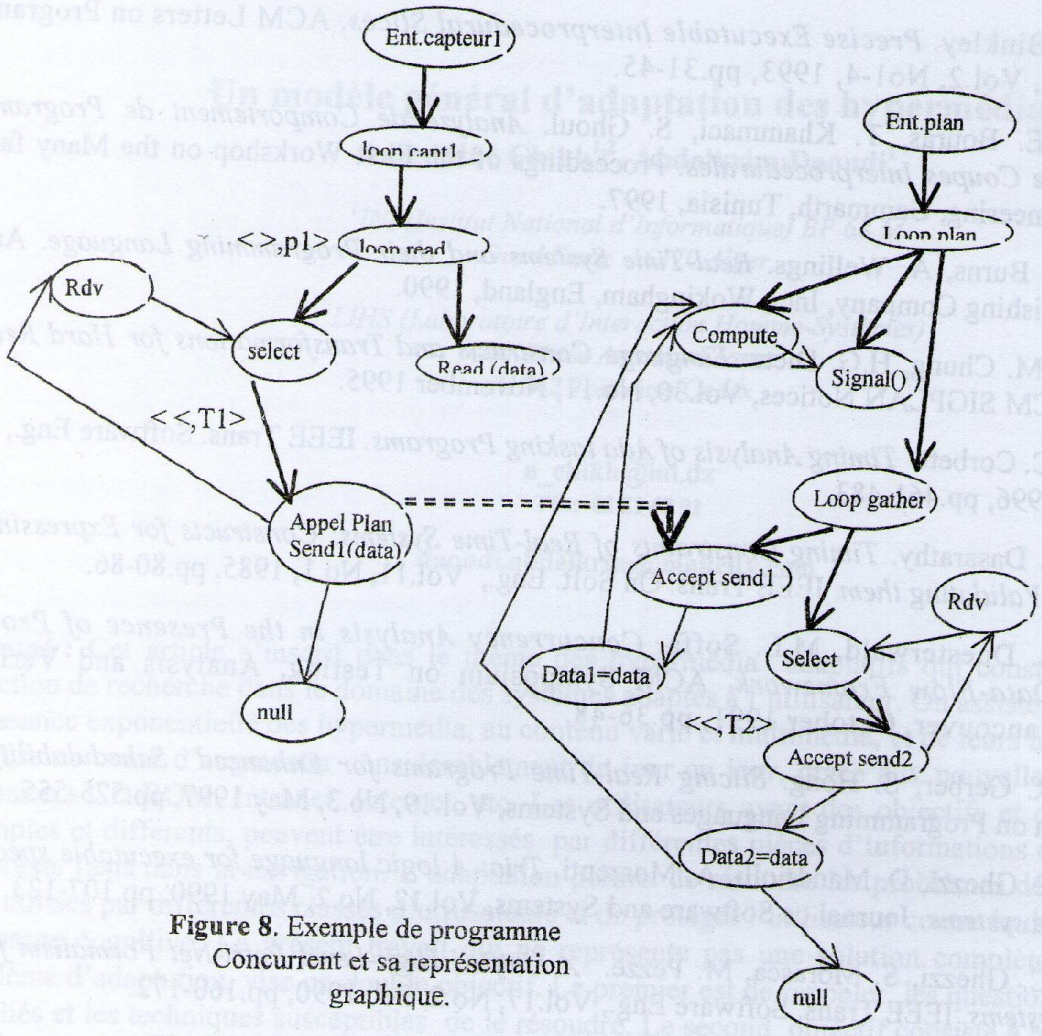

Figure 8. Exemple de programme Concurrent et sa représentation graphique.

5 CONCLUSION

La restructuration de logiciel temps réel est un outil qui concourt à la réalisation des objectifs de la maintenance et l'amélioration et la préservation de la structure des programmes tout au long de leur évolution. C'est aussi un moyen pour améliorer la qualité du logiciel et permettre de réaliser les objectifs de qualité de services fixés. Dans ce papier, nous avons présenté un modèle de comportement fonctionnel et temporel, suffisamment fin, pour obtenir automatiquement des tranches précises et exécutables. Ce modèle est à la base de l'approche de restructuration qui est une forme de transformation qui préserve la sémantique initiale du programme. De plus, étant donné que les transformations modifient l'ordre des instructions, notre approche d'extraction de tranche prend en considération les dépendances de sortie, les anti-dépendances, les dépendances de flux, ainsi que les dépendances temporelles. Pa exemple, en déterminant le fragment temporel d'une tâche, nous avons montré que nous pouvons retsructurer une application temps réel, initialement non ordonnançable, en une autre application équivalente mais ordonnançable.

Afin de se focaliser sur le temps réel, nous avons traité dans cet article, les constructions les plus utilisées dans les langages de programmation temps réel telles que les celles des contraintes de temps et de concurrence. D'autres constructions plus complexes sont à l'étude telles que les contraintes sur les ressources et la restructuration de systèmes temps réel distribués.

REFERENCES

[BEN97] M.S. Bendelloul, Z.E. Bouras, S. Ghoul, T. Khammaci. *Assistance à la Compréhension de Programme. Un Modèle et un Algorithme de Fragmentation.* La revue Génie Logiciel, No.45, Septembre 1997, pp.32-42.

[BIN93] D. Binkley. *Precise Executable Interprocedural Slices*. ACM Letters on Programming and Systems, Vol.2, No1-4, 1993, pp.31-45.

[BOU97] Z.E. Bouras, T. Khammaci, S. Ghoul. *Analyze de Comportement de Programmes - Extraction de Coupes Interprocédurales*. Proceedings of the First Workshop on the Many facets of Process Engineering, Gammarth, Tunisia, 1997.

[BUR90] A. Burns, A. Wellings. *Real-Time Systems and their Programming Language*. Addison-Wesley Publishing Company, Inc., Wokingham, England, 1990.

[CHU95] T.M. Chung, H.G. Dietz: *Language Constructs and Transformations for Hard Real-Time Systems*. ACM SIGPLAN Notices, Vol.30, No.11, November 1995.

[COR96] J.C. Corbett. *Timing Analysis of Ada tasking Programs*. IEEE Trans. Software Eng., Vol.22, No.7, July 1996, pp.461-483.

[DAS85] B. Dasarathy. *Timing Constraints of Real-Time Systems: Constructs for Expressing them, Method for Validating them*. IEEE Trans. On Soft. Eng., Vol.11, No.1, 1985, pp.80-86.

[DUE91] E. Duesterwald, M.L. Soffa. *Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework*. ACM Symposium on Testing, Analysis and Verification (TAV4), Vancouver, October 1991, pp.36-48.

[GER97] R. Gerber, S. Hong: *Slicing Real-Time Programs for Enhanced Schedulability*: ACM Transaction on Programming Languages and Systems, Vol.19, No.3, May 1997, pp.525-555.

[GHE90] C. Ghezzi, D. Mandriolli, A. Morzenti. *Trio: A logic language for executable specification of real-time systems*. Journal os Software and Systems, Vol.12, No.2, May 1990, pp.107-123.

[GHE91] C. Ghezzi, S. Morasca, M. Pezzë. *A Unified High-Level Petri-Net Formalism for Time-Critical-Systems*. IEEE Trans. Software Eng., Vol.17, No.2, Feb 1990, pp.160-172.

[HOR89] S. Horwitz, J. Prins, T. Reps. «Integrating Noninterfering Versions of Programs». ACM trans. Programming languages and Systems, vol. 3, N°3, July 1989, pp.345-387.

[HOR90] S. Horwitz, T. Reps, D. Binkley. *Interprocedural Slicing using Dependence Graph*. ACM trans. Program. Lang. Syst., Vol.12, No.1, January 1990, pp.26-60.

[KEN91] K.B. Kenny, K.J. Lin. *Building Flexible Real-Time Systems Using the Flex Language*, IEEE Computer, May 1991, pp. 70-78.

[KIM99] M.T. Kimour, M.S. Bendelloul. *Program Structuring to Enhance Schedulability*. Proceedings of the First Mediterranean International Conference on Computer Technologies, Tizi-Ouzou, Algeria, June 1999.

[KLE94] M. Klein, J. Lehoczky, R. Rajkumar. *Rate-Monotonic Analysis for Real-Time Industrial Computing*, IEEE Computer, Vol.27, No.1, January 1994.

[KLI86] E. Kligerman, A. Stoyenko. *Real-Time Euclid: A Language for Reliable Real-Time Systems*, IEEE Trans. Software Eng., September 1986, pp.941-949.

[LIU73] C.L. Liu, J.W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, Vol.1, No.1, 1973, pp.46-61.

[OTT84] K. Ottenstein, L. Ottenstein. *The Program Dependence Graph in a Software Development Environment*. ACM SIGSOFT/SIGPLAN, ACM Press, New-York,84, pp.177-184.

[RAM91] K. Ramamritham. *Real-Time Concurrent C: A Language for Programming Dynamic Real-Time systems*, Journal of Real-Time Systems, No.3, 1991, pp.377-405.

[SCH92] E. Schonberg, M. Gerhardt, CH. Hayden. *A Technical Tour of ADA*. Com. of the ACM, Vol.35, No.11, November 1992, pp.43-52.