Une approche pour l'analyse du pire cas du temps d'exécution des programmes temps-réels

Djemai KEBBAL

IUT de Tarbes - Département SERECOM
1, Rue Lautréamont 65016 Tarbes Cedex

Djemai.Kebbal@iut-tarbes.fr

Sections de rattachement : 27
Secteur : Secondaire

Résumé. *Dans cet article, nous proposons une approche pour le calcul automatique du pire cas du temps d'exécution (WCET) des programmes temps-réels. L'objectif est de pouvoir extraire automatiquement les informations de flot en relation avec la sémantique du programme. Cette information est utilisée pour réduire la surestimation du WCET. Nous nous intéressons particulièrement aux limites des boucles et aux chemins infaisables. L'approche prend en charge les boucles avec de multiples conditions de sortie et les boucles non-rectangulaires dans lesquelles le nombre d'itérations d'une boucle interne dépend de l'itération courante de la boucle externe. L'approche combine l'exécution symbolique et l'énumération de chemins afin d'éviter de déplier les boucles comme le font les approches basées uniquement sur l'exécution symbolique.*

Mots-clés. *Systèmes temps-réels stricts, analyse statique du WCET, analyse de flot.*

1. Introduction

Aujourd'hui, les systèmes temps-réels et embarqués occupent une place importante dans tous les domaines industriels et économiques (aviation, aérospatial, automobile, contrôle des processus industriels, etc.). Les systèmes temps-réels imposent des contraintes de temps qui doivent être vérifiées afin d'éviter des conséquences indésirables pouvant même être désastreuses dans le cas des systèmes temps-réels stricts. Cet article est consacré à l'analyse du WCET qui consiste à calculer un majorant du temps d'exécution des programmes sur des plateformes matérielles.

L'analyse statique du WCET consiste à effectuer une analyse de haut niveau du code source dans le but d'éviter de travailler sur les données du programme. Le résultat de l'analyse statique est seulement une estimation du WCET au lieu des valeurs exactes. Par conséquent, l'analyse du WCET doit respecter deux propriétés importantes : *fiabilité* et *précision* des valeurs calculées.

L'analyse du WCET procède généralement en trois étapes : analyse de flot,

analyse de bas niveau et calcul d'une estimation du WCET. L'analyse de flot caractérise les séquences d'exécution des composants du programme, leur fréquence d'exécution, etc. (chemins d'exécution). Deux types d'information de flot sont généralement considérés. La première catégorie est en relation avec la structure du programme et peut être extraite automatiquement. La seconde catégorie concerne les fonctionnalités et la sémantique du programme. Elle comporte des informations sur les limites des boucles et les chemins faisables/infaisables en particulier. Ce type d'information est complexe à inférer automatiquement et par conséquent fourni généralement par le programmeur [1, 2, 3]. Les approches d'analyse statique de WCET peuvent être classées en trois catégories : approches basées sur les chemins, IPET et approches basées sur les arbres. Les approches des chemins consistent à énumérer explicitement l'ensemble des chemins du programme [3]. L'inconvénient majeur de ces approches est le nombre exponentiel de chemins générés. Les approches IPET[1] considèrent que les chemins du programme appartiennent à la solution mais ne les énumèrent pas explicitement. [1, 2].

La plupart de ces approches impliquent le programmeur dans l'extraction des informations de flot, en particulier l'information de flot en relation avec la sémantique du programme (limites de boucles, chemins infaisables, etc.). Malgré le fait que les informations ainsi fournies peuvent être d'une grande précision, le processus est sensible aux erreurs. L'interprétation abstraite et l'exécution symbolique peuvent être utilisées pour inférer un sous-ensemble de ces informations automatiquement. La plupart de ces approches sont complexes (déroulement de boucles [4], méthodes utilisées pour le calcul ne sont pas très pratiques [5]). Dans [6], une approche pour le calcul des limites des boucles est proposée. Cependant, ils considèrent uniquement des boucles avec la variable de contrôle incrémentée d'une valeur constante à chaque itération.

L'exécution symbolique consiste à évaluer le programme en utilisant des symboles comme valeurs d'entrée des variables du programme. Le programme est déroulé jusqu'à sa terminaison, ce qui représente une exécution symbolique. Ceci permet de déterminer les valeurs de sortie des variables en fonction des données du programme. Les approches basées sur l'exécution symbolique sont complexes en temps et en espace mémoire, ce qui les rend peu attractives pour l'analyse du WCET. Notre objectif est de développer une approche pratique pour l'extraction automatique d'information de flot ayant un coût réduit. Nous proposons une approche basée sur l'exécution symbolique par blocs et l'énumération de chemins. L'approche permet d'améliorer la complexité de l'exécution symbolique. Dans la section suivante, nous présentons un ensemble de concepts utilisés par l'approche d'analyse de flot. Cette dernière sera décrite dans la section 3.

[1] Implicit Path Enumeration Techniques.

2. Concepts utilisés par l'approche

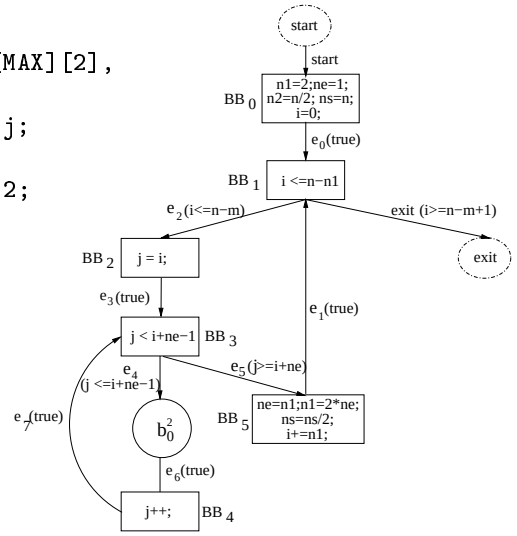
2.1. Représentation du programme

Nous utilisons le formalisme du graphe de flot de contrôle (CFG) pour exprimer le flot de contrôle du programme. Le code source ou objet du programme est décomposé en un ensemble de blocs de base. Un bloc de base est un ensemble d'instructions avec un unique point d'entrée situé au début du bloc et un unique point de sortie situé à la fin de bloc. Le CFG du programme est représenté par le graphe $G = (B, E)$, où B représente les blocs de base du programme et E les contraintes de précédence entre les blocs de base. Deux blocs artificiels notés *start* et *exit* sont rajoutés. La figure 1-b illustre le CFG d'un programme dont le code source est présenté à la figure 1-a.

```
void fft(unsigned int n, double f[MAX][2],
double t[MAX][2]) {
  unsigned int ne, n1, ns, n2, i, j;

  ne = 1; n1 = 2; ns = n; n2 = ns/2;
  for (i=0; i<=n-n1; i+=n1) {
    for (j=i; j<=i+ne-1; j++) {
      update t[j] from f
    }
    ne = n1; n1 = 2*ne;
    ns = n2; n2 = ns/2;
  }
}
```

a) Code source en C.



b) Graphe de flot de contrôle (CFG).

Figure 1. – *Exemple d'un programme.*

2.2. Les blocs et les graphes de bloc

Nous utilisons la notion de bloc, dans laquelle, un ensemble de blocs de niveau l sont groupés dans un bloc de niveau $l-1$. Les blocs complexes ainsi construits correspondent aux entités complexes des langages de programmation (boucles, fonc-

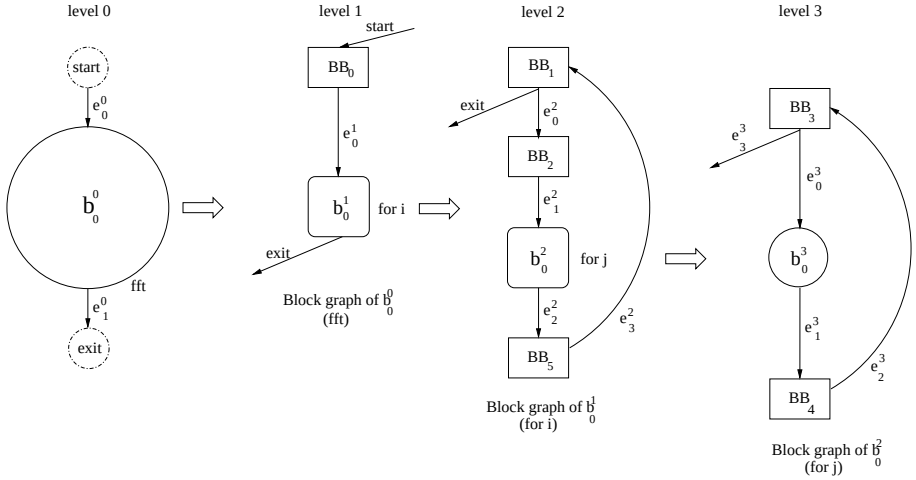


Figure 2. – *Les graphes de blocs de l'exemple.*

tions, tests, modules, etc.). La composition est effectuée d'une manière récursive jusqu'au niveau CFG. La figure 2 illustre les graphes de bloc construits pour l'exemple de la figure 1. Formellement, un bloc b de niveau l est définie par la formule 1 et composé de : un ensemble de sous-blocs (B_b) ; un ensemble de blocs entêtes $(B_b^h \subseteq B,\ B_b^h \subseteq B_b)$; un ensemble d'arcs E_b reliant les sous-blocs ; et un ou plusieurs arcs de sortie(*"exit"*, $E_b^e \subseteq E_b$).

$$b = \{B_b, B_b^h, E_b, E_b^e\}. \tag{1}$$

Chaque bloc b de niveau l est représenté par un *graphe de bloc* décrivant sa structure. L'ensemble B_b de blocs de b est composé de blocs de niveaux inférieurs $(m < l)$. L'ensemble des arcs E_b est construit comme suit : chaque arc du CFG reliant deux nœuds appartenant à deux blocs différents b_i et b_j de B_b produit un arc de niveau l reliant b_i à b_j. Les arcs vers des blocs n'appartenant pas à B_b produisent des arcs de type *exit*. Les arcs redondants sont élimines. Sur la figure 2, dans le graphe du bloc b_0^1 (boucle *for*), l'arc e_3 reliant les blocs de base BB_2 et BB_3 dans le CFG engendre l'arc e_1^2.

2.3. Chemins

Un chemin dans un bloc b est une séquence d'arcs dans E_b, où le nœud d'arrivée de chaque arc est le nœud source de l'arc suivant dans le chemin. Un chemin p peut être décomposé en une séquence de sous-chemins $p_0, p_1, \ldots, p_{k-1}$.

Chaque bloc b est associé l'ensemble de ses chemins de bloc. On distingue deux types de chemins de bloc : *chemins-sortants* et *chemins-itératifs*. Un chemin sortant dans b est un chemin partant du bloc entête de b et se terminant par un arc de sortie de b. De même, un chemin itératif dans b est un chemin partant du bloc entête de b et se terminant par un arc de retour de b. Sur la figure 2 et le tableau 1, le bloc b_0^1 possède un chemin sortant p_0^2 composé seulement de l'arc *exit* et un chemin itératif p_1^2 composé des arcs e_0^2, e_1^2, e_2^2 et e_3^2.

3. L'approche d'analyse de flot

Nous utilisons une approche d'analyse de flot de données afin de déterminer la valeur des variables du programme à différents points du programme. L'approche combine l'exécution symbolique et l'analyse de chemins. L'analyse de flot est effectué pour chaque bloc sans dérouler les boucles. Le nombre d'exécutions de chaque bloc est plutôt calculé analytiquement en utilisant l'exécution symbolique et l'énumération de chemins. Seulement, un sous-ensemble des états symboliques du programme[2] est calculé.

3.1. Condition de chemin et action de chemin

Chaque arc e du CFG est associé une condition de chemin $PC(e)$: un prédicat Booléen conditionnant l'exécution de l'arc étant donné l'état du programme s au nœud source de l'arc. De même, chaque bloc de base bb applique une action de bloc $BA(bb)$ qui représente l'effet de l'exécution de toutes les instructions du bloc sur l'état du programme s (règles de l'exécution symbolique). L'action de chemin d'un chemin p notée $PA(p)$ est la séquence des actions de bloc de tous les blocs composant p. De même, la condition de chemin de p est le *"et logique"* des conditions de chemins de tous les arcs constituant p ($PC(p) = PC(e_0) \wedge PC(e_1) \ldots \wedge PC(e_{n-1})$). Ces propriétés peuvent être également appliquées aux sous-chemins.

Afin de calculer l'action de chemin d'un chemin p, nous considérons l'ensemble des variables du programme affectées dans les différents blocs de p V_p. Si $BA(bb, v)$ est la fonction appliquée par le bloc de base bb sur la variable $v \in V_p$ qui représente l'effet de l'exécution de toutes les instructions du bloc sur v. L'action appliquée sur v par p ($PA(p, v)$) est la séquence des actions de bloc appliquées par tous les blocs constituant p dans l'ordre de leur occurrence dans p. $PA(p, v) = (BA(bb_0, v); BA(bb_1, v); \ldots; BA(bb_{n-1}, v))$ peut être représenté par une expression de la forme $av + b$ tels que a et b sont des entiers constants ($a > 0$). Ce qui signifie que l'instruction de mise à jour de la variable de contrôle de la boucle (ex. i) est de la forme $i = a * i + b$.

[2] Les états symboliques à l'entrée et à la sortie du bloc.

3.2. Évaluation des chemins

L'évaluation d'un chemin p utilise l'expression de condition du chemin $PC(p)$ et l'action du chemin $PA(p)$. $PC(p)$ est décomposé en expressions Booléennes élémentaires reliées par des opérateurs logiques. Chaque expression élémentaire e est de la forme i *op* *expr*, tel que *op* est un opérateur relationnel et *expr* une expression évaluée à un entier. Par conséquent, chaque expression élémentaire e définie une borne d'un intervalle entier. L'autre borne de l'intervalle est déterminée par la valeur initiale de i (ex. $i \leq 100$, l'intervalle est $[\alpha_0, 100]$). Par la suite, les paramètres suivants sont évalués pour chaque expression :

- *Type d'intervalle :* l'intervalle est croissant si *op* est "$<$" ou "$\leq$", constant si *op* est "$=$" et décroissant si *op* est "$>$" ou "$\geq$".
- *Direction :* si la variable i est incrémentée dans $PA(p)$, la direction est positive et négative si i est décrémenté dans $PA(p)$. Si i n'est pas mis à jour, la direction est nulle.

Le *type d'intervalle* et la *direction* sont utilisés dans l'évaluation du nombre d'itérations des chemins itératifs. Cette étape produit le nombre d'itérations d'un chemin itératif p par rapport à l'expression e I_p^e et l'état symbolique résultant s_p^e. La première étape consiste à vérifier si les chemins itératifs sont vides ou illimités, etc. Par la suite, les paramètres de boucle $(a, b, N_1 et N_2)$ sont déduits et fournis au module algébrique qui calcule le nombre d'itérations I_p^e.

Afin de calculer le nombre d'itérations, nous considérons la suite entière v_n représentant la suite des valeurs prises par la variable de contrôle de la boucle :

$$\begin{cases} v_0 & = & N_1 \\ v_{n+1} & = & av_n + b \quad \forall n \in \mathcal{N}. \end{cases}$$

Le nombre d'itérations I est donné par l'inéquation $N_2 - N_1 \geq \sum_{n=0}^{I-2} s_n$. Où $s_n = v_{n+1} - v_n$. Le nombre d'itérations I est alors donné par la formule 2.

$$I = \begin{cases} \lfloor \log_a(1 + \frac{(N_2 - N_1)(a-1)}{b+(a-1)N_1}) \rfloor + 1 & si \quad N_2 > N_1 \wedge a > 1 \wedge N_1(1-a) \neq b \\ \lfloor \frac{N_2 - N_1}{b} \rfloor + 1 & si \quad a = 1 \\ \infty & si \quad b = N_1(1-a) \end{cases} \tag{2}$$

Lorsque $b = N_1(1-a)$, la variable de contrôle v n'est jamais mise à jour $v_n = N_1 \ \forall n \geq 0$. Le nombre d'itérations est alors infini. Le nombre d'itérations final de p est déterminé à partir du nombre d'itérations de toutes les expressions élémentaires de $PC(p)$ comme suit : $I_p^{e_1 \vee e_2} = \max(I_p^{e_1}, I_p^{e_2})$ et $I_p^{e_1 \wedge e_2} = \min(I_p^{e_1}, I_p^{e_2})$.

3.3. Exécution symbolique par blocs

Avant de démarrer l'exécution symbolique, l'ensemble des informations caractérisant les blocs du programme est déterminé. Le tableau 1 résume les infor-

Bloc	**Chemin**	**Type**	**Composition**	**PC**	**PA**	**Arc**
b_0^2	p_0^3 p_1^3	$exit$ $loop$	$(e_3^3) = (BB_3)$ $(e_0^3, e_1^3, e_2^3) =$ (BB_3, b_0^3, BB_4)	$j \geq i + ne$ $j \leq i + ne - 1$	$(PA(b_0^3); j \rightarrow j + 1)$	e_2^2
b_0^1	p_0^2 p_1^2	$exit$ $loop$	$(exit) = (BB_1)$ $(e_0^2, e_1^2, e_2^2, e_3^2) =$ $(BB_1, BB_2, b_0^2, BB_5)$	$i \geq n - n1 + 1$ $i \leq n - n1$	$(j \rightarrow i; PA(b_0^2); ne \rightarrow n1; n1 \rightarrow 2 \times ne; ns \rightarrow ns/2; i \rightarrow i + n1)$	$exit$
b_0^0	p_0^1	$exit$	$(start, e_0^1, exit) =$ (BB_0, b_0^1)	$true$	$(ne \rightarrow 1; n1 \rightarrow 2; ns \rightarrow n; n2 \rightarrow n/2; i \rightarrow 0; PA(b_0^1))$	e_1^0
top	p_0^0	$exit$	$(e_0^0, e_1^0) = (start, b_0^0, exit)$	$true$	$(PA(b_0^0))$	

TAB. 1 – Définition des chemins du programme et leurs paramètres

Bloc	$\mathbf{V_{in}}$	$\mathbf{V_{out}}$	**Iters**
b_0^2	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >\}$	$\{< i, \alpha_0 >, < j, \alpha_0 + \alpha_2 >, < ne, \alpha_2 >\}$	α_2
b_0^1	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >, < n1, \alpha_3 >, < ns, \alpha_4 >, < n, \alpha_5 >\}$	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >, < n1, \alpha_3 >, < ns, \alpha_4 >, < n, \alpha_5 >\}$	I_1
b_0^0	$\{< i, 0 >, < j, \alpha_1 >, < v, \alpha_2 >, < x, \alpha_3 >, < N, \alpha_4 >\}$	$\{< i, \alpha_4 - 1 >, < j, \alpha_4 >, < v, \alpha_2^3 >, < x, \alpha_3^2 >\}$	1
top	$\{< i, undef >, < j, undef >, < v, \alpha_2 >, < x, undef >, < N, \alpha_4 >\}$	$\{< i, \alpha_4 - 1 >, < j, \alpha_4 >, < v, \alpha_2^3 >, < x, \alpha_3^2 >, < N, \alpha_4 >\}$	1

TAB. 2 – Exécution symbolique par blocs de l'exemple

mations caractérisant le programme des figures 1 et 2. Pour les chemins de sortie, l'arc du graphe du bloc englobant que devra prendre le flot d'exécution après avoir pris le chemin p est également mentionné. Le nœud source de l'arc constitue un *point de sortie* du bloc. La condition de chemin à ce point est le *ou logique* des conditions de chemin de tous les chemins de sortie se terminant à ce point.

L'algorithme d'exécution symbolique par blocs consiste à effectuer une évaluation des blocs récursivement en commençant par les blocs de haut niveau (les blocs les plus internes). L'exécution symbolique d'un bloc consiste à calculer le nombre d'exécutions de chaque chemin appartenant aux chemins de bloc P. Le résultat est le nombre d'itérations du chemin I_p et les états symboliques résultants S_p.

L'évaluation des chemins d'un bloc procède par déterminer pour un état symbolique s quel chemin $p \in P$ doit être exécuté en utilisant la condition de chemin

$PC(p)$. Par la suite, le chemin sélectionné est appliqué à l'état symbolique et les états symboliques résultants sont générés. La figure 3 présente un résumé de l'exécution du bloc b_0^1. Les arcs sont étiquetés par le chemin devant être appliqué à l'état symbolique au nœud source de l'arc. Initialement (état s_0), toutes les variables sont assignées des symboles et la condition de chemin est mise à $true$. L'ensemble des chemins de bloc $P = \{p_0^2, p_1^2\}$. p_1^2 est l'unique chemin itératif. $PC(p_1^2)$ se compose d'une seule expression $e = i \leq n - n_1$, dans laquelle n_1 est non constant. L'outil algébrique de l'analyseur doit réévaluer e afin d'obtenir une expression constante dans le membre droit de l'inégalité et $PA(p_1^2)$ afin de pouvoir appliquer la formule 2. e est ainsi réévalué à $3i \leq 2n - 4$ et $PA(p_1^2, i)$ à $i \rightarrow 2i - \alpha_0 + 2\alpha_3$. L'évaluation du chemin par rapport à e produit I_1 et le nouvel état s_1. La formule 2 est utilisée à cet effet ($a = 2$ et $b = 2\alpha_5 - \alpha_0$). $I_1 = \lfloor \log_2(1 + \frac{\frac{2\alpha_5 - 4}{3} - \alpha_0}{2\alpha_3}) \rfloor + 1$. Le nombre d'itérations de b_0^2 est également mis à jour, ce qui donne $\sum_{i=1}^{I_1} i(i \rightarrow 2i) = 2^I - 1$.

La formule 2 est également utilisée pour déduire la nouvelle valeur des variables du programme dans s_1. Nous nous contentons de mentionner ces valeurs par des astérisques étant données qu'elles sont trop longues.

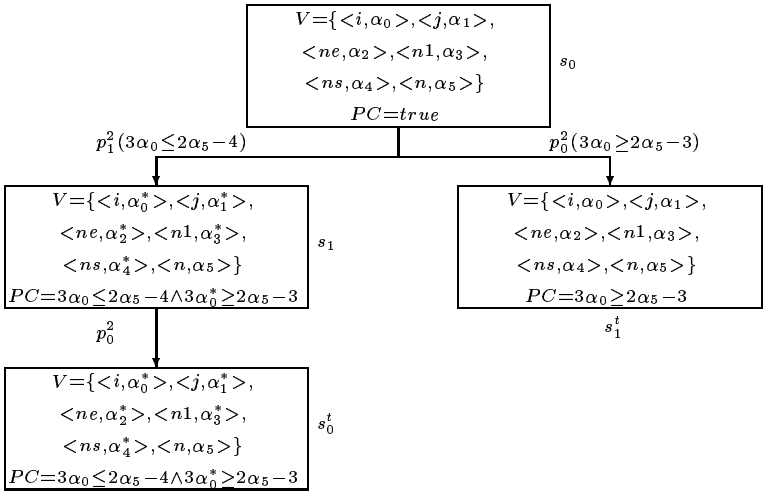


Figure 3. – *Exécution symbolique par blocs du bloc b_0^1*

4. Conclusion

Les systèmes temps-réels doivent être prédictibles en temps et en espace mémoire étant données les conséquences critiques pouvant être provoquées par les erreurs. L'analyse statique du WCET permet d'éviter de travailler sur les données du

programme, mais produit généralement des valeurs surestimées du WCET. Par conséquent, des techniques permettant d'améliorer la précision des valeurs calculées sont nécessaires. Malheureusement, ces techniques sont parfois difficiles à automatiser du fait qu'elles traitement des aspects sémantiques des programmes.

Nous avons proposé une approche pratique pour l'extraction automatique des informations de flot concernant les aspects sémantiques du programme. Notre approche a deux avantages principaux. Elle permet d'améliorer la précision du WCET estimé à travers les informations de flot en relation avec la sémantique du programme qu'elle fournit. De plus, l'approche prend en charge les boucles dites non-rectangulaires et permet d'éliminer la plupart des chemins infaisables. Le WCET estimé est fourni sous forme d'expressions symboliques fonction des variables d'entrée (paramètres de fonctions, etc.). Le second avantage concerne la complexité améliorée de l'approche à travers l'exécution symbolique par blocs qui permet de ne pas déplier les blocs itératifs.

Nous sommes en train d'implanter un prototype afin de pouvoir évaluer les performances de l'approche. Nous comptons également étendre les expressions utilisées dans l'évaluation des boucles aux formules de Presburger et utiliser les résultats obtenus pour ces formules.

Bibliographie

Y.-T. Steven Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-time Systems, La Jolla, California*, June 1995.

A. Ermedahl. *A modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 2000.

J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2) :61–74, 1998.

B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003*, pages 99–102, Polytechnic Institute of Porto, Portugal, 2003.

C. Healy, M. Sjodin, V. Rustagi, D. Whalley, , and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2-3) :129–156, May 2000.