Rapport de stage de DEA

effectué à l'université Louis Pasteur de Strasbourg

au LSIIT : Laboratoire des Sciences de l'Imagerie, de l'Informatique et de la Télédétection.

dans l'équipe ICPS : Image et Calcul Parallèle Scientifique.

Délégation de traitements et multilocalisation de données

Gilles Bitran

sous la direction de Benoît Meister et Philippe Clauss.

LSIIT, UMR 7005, ULP-CNRS.

29 avril 2004

Mots-clés : vivacité des données, graphe d'appels de fonctions, systèmes distribués, parallélisation automatique.

Remerciements

Je tiens à remercier l'équipe ICPS qui m'a chalheureusement accueilli.

Je remercie Philippe Clauss et, plus particulièrement, Benoît Meister (son point de vue et nos longues discussions ont considérablement enrichis mon travail), pour leur encadrement tout au long de ce stage.

Je remercie aussi Stéphane Genaud pour une relecture de mon rapport et Eric Violard pour ces conseils pour établir certains algorithmes et les vérifier.

Je remercie également l'ensemble des étudiants du DEA d'informatique qui ont passé cette année avec moi ainsi que mes amis et mes proches dont l'aide de quelques uns a été précieuse.

Table des matières

Introduction

Les systèmes embarqués (ou enfouis) ont constitué une cible privilégiée pour l'optimisation de programmes dès lors que leur architecture s'est rapprochée de celle des ordinateurs classiques, devenant capables d'exécuter des logiciels. L'optimisation recherchait alors, en plus d'une amélioration des performances, une réduction de l'espace mémoire occupé par le programme et de sa consommation électrique. Les systèmes temps réel ont apporté des contraintes supplémentaires en terme de temps d'exécution. Aujourd'hui, une part croissante des systèmes embarqués est pourvue d'une connectivité réseau. Il devient donc pertinent de prendre en compte celle-ci dans le processus d'optimisation.

Le projet auquel nous nous intéressons ici, mis en place au LSIIT dans le cadre du programme EME[1], a pour but d'augmenter les capacités d'exécution d'un ordinateur lorsqu'il est accessible par réseau en découpant un programme séquentiel destiné à cet ordinateur pour l'exécuter partiellement sur des ordinateurs distants. Cette problématique de délégation de traitements s'apparente à la parallélisation automatique de programmes et peut donc s'appliquer à tout ordinateur connecté à d'autres ordinateurs par un réseau. Un programme séquentiel, écrit dans un langage impératif, pourra être transformé automatiquement pour être exécuté sur une grille d'ordinateurs hétérogènes après avoir été lancé depuis un ordinateur quelconque (un système embarqué, un simple PC, un gros calculateur...). Grâce à ce projet, les ordinateurs pourront exécuter des programmes nécessitant plus de capacités qu'ils n'en possèdent en exploitant celles d'autres machines accessibles par réseau.

Toutefois, dans le cadre des systèmes embarqués dont l'environnement d'exécution (bande passante, énergie, temps, ...) est fortement dynamique, la délégation de traitements doit également être dynamique. L'environnement d'exécution des ordinateurs classiques en réseau dépendant d'un moins grand nombre de paramètres que celui des systèmes embarqués, la délégation de traitements sur un tel réseau se voit comme un cas particulier, moins contraint.

Ce projet s'appuie sur une analyse minutieuse du code source des programmes dont on veux déléguer certains traitements. Le découpage du programme est primordial car si l'on parvient à mettre en évidence des traitements relativement indépendants on peut espérer une amélioration considérable des performances grâce à une parallélisation des exécutions de ces traitements et une minimisation des communications.

La première partie de ce rapport est consacrée à une présentation conceptuelle du projet. Dans la seconde partie nous décrivons ce qui a été réalisé au cours du stage.

[1] EME : Environnements mobiles embarqués.

Première partie

Présentation générale du projet

Ce projet de délégation de traitements ressemble à d'autres compilateurs comme McCAT[1] et ORC/OPEN64[2] car il se base également sur une représentation haut niveau des programmes (FIRST pour McCAT et WHIRL[2] pour ORC). Une telle représentation des programmes permet :

- de bénéficier de toute l'information donnée par le programmeur ;
- de disposer de structures de contrôle de flot bien formées (contrairement à du « code spaghetti[3] »), c'est-à-dire :
  - des boucles (comme les structures `do-while`, `while` ou `for` en C) ;
  - des structures de décision à choix multiples (comme les `if-else` ou `switch-case` en C), également appelées structures conditionnelles ;
  - des appels de fonctions.
- une meilleure lisibilité et compréhension des analyses et des transformations du code, en préservant leur structure syntaxique.

Comme le montre le schéma de la figure 1, ce projet se divise en quatre modules relativement indépendants :

- la spécification du découpage du programme dont on veut déléguer des traitements ;
- une analyse de la distribution des traitements ;
- la réalisation effective du découpage ;
- la mise en place de la distribution.

Les trois premiers modules permettent d'effectuer une transformation automatique de programmes séquentiels écrits dans un langage impératif et le dernier module correspond au protocole d'exécution des programmes transformés. Dans les sections suivantes, nous exposons les concepts à mettre en oeuvre pour développer les modules du projet.

[2] WHIRL : Very HIgh Representation Level.

[3] Code spaghetti : code utilisant des labels et des instructions de sauts (`goto`).

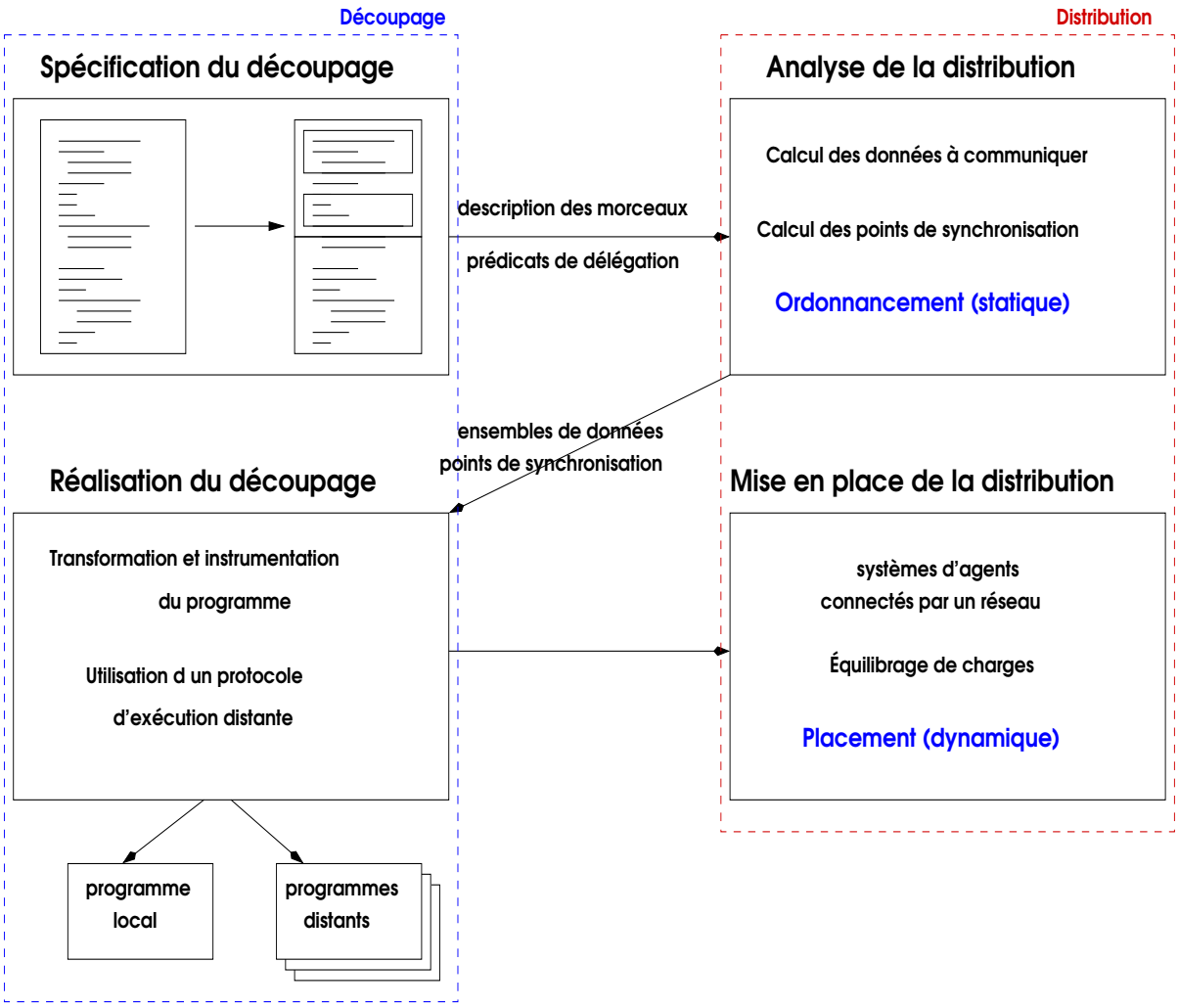

FIG. 1 – Schéma global du projet.

1 Spécification du découpage

Ce premier module consiste à découper un programme, selon les caractéristiques de la machine locale[4], afin de distribuer les traitements qui le composent. Avant de partitionner le programme, il faut collecter de nombreuses informations relatives à son code et à ceux des librairies ou packages utilisés, s'il y en a et si leur code source est disponible. Cela permet de déterminer le découpage le plus adéquat possible pour la machine locale.

Dans certains cas, le programme dont on veut déléguer des traitements doit être partiellement normalisé. En effet, dans les langages impératifs comme C, C++ ou Java, les tests conditionnels ne sont pas toujours exécutés dans leur intégralité.

[4] Machine locale : machine qui délègue l'exécution de certains traitements d'un programme.

Exemple 1 Les tests conditionnels suivants ne sont pas toujours exécutés intégralement :

```
...
int * x, y;
...
if ((x != NULL) || (y > 3))
  ... /* exécution du bloc d'instructions BLOC_1. */
...
while (x && (y > 0) && (x[y] != y))
  ...
...
```

Pour le premier test, si le pointeur `x` est différent de `NULL`, l'expression `(y > 3)` n'est pas évaluée. De même, pour le second test, l'expression `(x[y] != y)` n'est évaluée que si le pointeur `x` est différent de `NULL` et si l'expression `(y > 0)` vraie.

Une normalisation possible des précédents tests est, en introduisant une nouvelle variable de type booléen pour normaliser le test conditionnel complexe de la structure de contrôle de flot itérative :

```
...
int * x, y;
int __new_var;
...
if (x != NULL)
  ... /* exécution du bloc d'instructions BLOC_1. */
else if (y > 3)
  ... /* exécution du bloc d'instructions BLOC_1. */
...
if (x)
  {
    if (y > 0)
      __new_var = (x[y] != y);
    else
      __new_var = 0;
  }
else
  __new_var = 0;
while (__new_var)
  {
    ...
    if (x)
      {
        if (y > 0)
          __new_var = (x[y] != y);
        else
          __new_var = 0;
      }
    else
      __new_var = 0;
  }
...
```

Remarque 1 *Cette normalisation des tests conditionnels sera exploitée par le second module du projet, l'analyse de la distribution (cf section 2). Cependant, comme elle modifie le code source (introduction de structures de contrôle de flot conditionnelles et de variables) elle doit être réalisée avant toutes les analyses du programme.*

Remarque 2 *Du fait de l'insertion de structures de contrôle de flot conditionnelles et de variables, cette transformation de code est relativement lourde ; de plus, le programmeur aura du mal à reconnaître son code après cette transformation. D'autre part, ces modifications de code ne sont utiles qu'à déterminer avec exactitude quelles instructions des tests conditionnels sont réellement exécutées afin d'identifier des données à communiquer entre les tâches. Nous proposons dans la section 2.1.4 une variante des méthodes d'identification des données à communiquer qui ne nécessite pas cette transformations des tests conditionnelles.*

Comme nous voulons éviter le « code spaghetti » et disposer de structures de contrôle de flot bien formées, une deuxième normalisation s'impose. Elle concerne les structures de contrôle de flot conditionnelles à choix multiples qui sont introduites par le mot clé `switch`. Ces structures de contrôle de flot sont dites bien formées lorsque tous les cas qu'elles contiennent correspondent à l'exécution d'un seul bloc d'instructions se terminant par l'instruction `break`.

Exemple 2 Voici un exemple de structure de contrôle de flot conditionnelle introduite par le mot clé `switch` qui est *mal formée* :

```
switch (x)
  {
     case 3:
     case 5:
     case 7:
        printf ("x est un nombre premier inférieur à 10: %d\n", x);
        break;
     case 2:
        y = z * t;
     case 4:
        y = y / x;
        break;
     ...
  }
```

Cette structure de contrôle de flot conditionnelle est mal formée car lorsque `x` vaut `2`, les bloc d'instructions correspondant respectivement à `case 2` et `case 4` sont exécutés. Cela peut poser des difficultés pour positionner les mécanismes de synchronisation, il faut donc la reformuler. Voilà quel pourrait être le résultat de la normalisation de cette structure de contrôle de flot conditionnelle mal formée, en dupliquant certaines instructions :

```
switch (x)
  {
     case 3:
     case 5:
     case 7:
        printf ("x est un nombre premier inférieur à 10: %d\n", x);
        break;
     case 2:
        y = z * t;
        y = y / x;
        break;
     case 4:
        y = y / x;
        break;
     ...
  }
```

Lorsque ces normalisations, ont été effectuées, l'analyse du code source du programme dont on veut déléguer des traitements peut commencer. Il faut tout d'abord rechercher les propriétés des types des données et étudier leur vivacité afin d'évaluer la charge inhérente à la communication des données.

Remarque 3 *Il est souhaitable de favoriser le parallélisme à gros grain, par exemple :*
- *en « inlinant » les fonctions qui ne sont appelées qu'à un seul endroit du programme ;*
- *en améliorant la localité spaciale et temporelle des accès aux données.*

Ce type de transformation du code pouvant modifier l'ordre d'exécution des instructions et compliquer les tests conditionels des structures de contrôles de flot itératives, elles doivent précéder l'analyse de vivacité des données et le partitionnement du programme ainsi que les normalisations de code précédemment décrites.

Après avoir analysé en détails le programme à distribuer, toutes les informations nécessaires pour déterminer judicieusement la localisation des morceaux de programme sont disponibles.

Définition 1 *Un morceau de programme est une partie de programme issue du découpage de celui-ci.*

Chaque morceau d'un programme est caractérisé par sa position dans le programme et sa capacité à être délégué.

1.1 Position des morceaux de programmes

Nous avons défini un mécanisme d'adressage d'instructions et d'opérandes (cf annexe A pour les détails de l'adressage [4]) qui permet d'associer une adresse unique à toutes les instructions et les opérandes des instructions d'un programme en respectant l'ordre partiel d'exécution et en tenant compte des différentes structures de contrôle de flot.

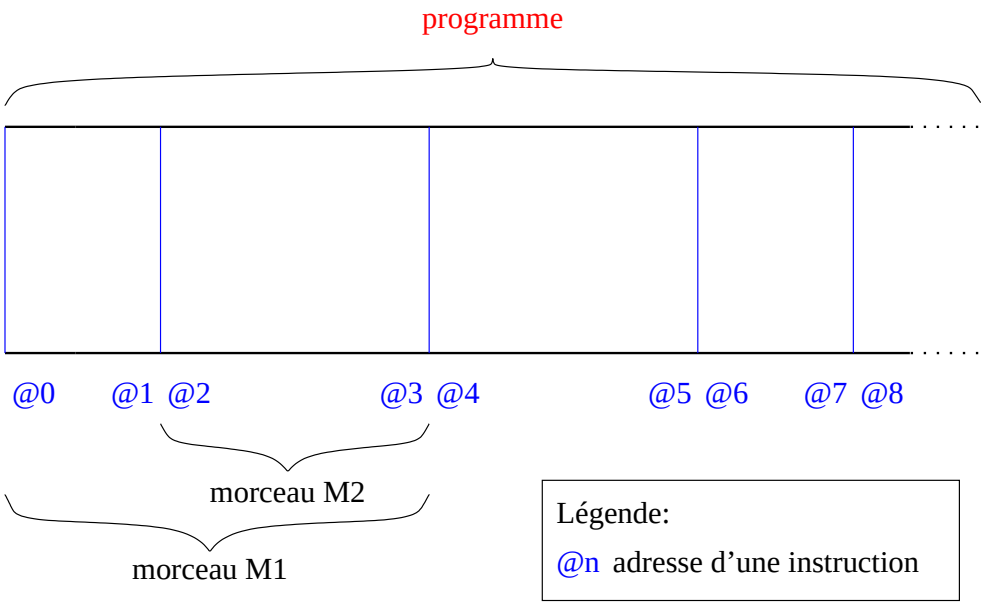

FIG. 2 – Position des morceaux de programmes.

La position des morceaux de programmes doit être définie à l'aide de notre système d'adressage. Ainsi, comme le montre l'exemple de la figure 2, la position d'un morceau dans un programme peut être déterminée par deux adresses :

- l'adresse de début du morceau, c'est à dire l'adresse de la première instruction du morceau ;
- l'adresse de fin du morceau, c'est à dire l'adresse de la dernière instruction du morceau.

Comme ce projet n'est adapté qu'aux programmes constitués de structures de contrôle de flot bien formées (i.e. ne générant pas du « code spaghetti »), toutes les instructions comprises dans un morceau de programme sont celles dont l'adresse est postérieure ou égale à l'adresse de début et antérieure ou égale à l'adresse de fin du morceau considéré.

Par exemple, les morceaux M1 et M2 de la figure 2 sont tels que, pour toute instruction I du programme :
$@0 \leq adresse(I) \leq @3 \Leftrightarrow I \in M1$,
$@2 \leq adresse(I) \leq @3 \Leftrightarrow I \in M2$, et,
$M2 \subset M1$ (où $adresse(I)$ représente l'adresse de I dans notre système d'adressage).

Les morceaux de programmes peuvent :

- soit, englober une ou plusieurs structures de contrôle de flot dans leur intégralité ;
- soit, être strictement inclus dans un des fils d'une structure de contrôle de flot.

Dans le cas des structures de contrôle de flot conditionnelles, si un morceau de programme débute avant le test conditionnel, il ne peut pas se terminer dans la structure de contrôle de flot car cela poserait quelques problèmes pour déterminer où l'exécution du morceau suivant doit commencer.

Si un morceau de programme commence avant le début d'une structure de contrôle de

flot itérative, il ne peut pas se terminer à l'intérieur de cette structure de contrôle de flot car cela impliquerait des duplications de code qui compliqueraient l'exécution sans pour autant améliorer les performances de distribution.

De cette manière, chaque morceau n'a qu'une seule adresse de début et une unique adresse de fin (celles-ci peuvent être identiques lorsqu'un morceau est réduit à une instruction). De plus, les adresses de début et de fin d'un morceau sont composées des mêmes coordonnées hormis la dernière, celle de l'adresse de début est antérieure (ou égale) à la celle de l'adresse de fin.

Pour accroître la modularité de l'exécution des programmes, le découpage peut être hiérarchique et aboutir à des morceaux contenant d'autres morceaux. Dans un premier temps, nous avons décidé que seule la machine locale pouvait déléguer des exécutions de morceaux, par la suite, cela pourrait être intéressant que des machines distantes puissent également déléguer des exécutions. D'autre part, un morceau ne peut pas être à cheval sur d'autres morceaux, cela impliquerait, au mieux, que l'exécution des parties communes soit dupliquée et nous pensons que cela ne présente pas d'intérêt.

Lorsqu'un morceau est inclus dans une ou plusieurs structures de contrôle de flot itératives, celui-ci sera éventuellement exécuté plusieurs fois. Nous lui assignons une propriété supplémentaire : un vecteur dit *d'itérations* dont la dimension est égale au nombre de structures de contrôle de flot itératives contrôlant son exécution. Ce vecteur permet de distinguer les instances d'un morceau inclus dans des boucles ou des nids de boucles.

Définition 2 *Une tâche est une instance de morceau en cours d'exécution (une tâche est pour un morceau, ce qu'est un processus pour un programme).*

Remarque 4 *Les techniques existantes d'ordonnancement parallèle dans les nids de boucles qui utilisent le formalisme des vecteurs d'itérations pourront être adaptées à notre modèle dans le futur (voir [5]).*

1.2 Capacité à être délégué

Une autre propriété des morceaux de programmes est leur capacité à être délégués. Il existe deux types de morceaux de programmes : les morceaux locaux et les morceaux délégables.

Définition 3 *Un morceau local est un morceau de programme qui sera toujours exécuté par la machine locale.*

Un morceau est dit local s'il contient des interactions avec l'utilisateur ou avec des ressources spécifiques à la machine locale, ou si son exécution locale est toujours préférable à son exécution déléguée. Par exemple si la communication inhérente à l'exécution déléguée d'un morceau a un coût global plus élevé que son exécution locale, ce morceau sera considéré comme local.

Définition 4 *Un morceau délégable est un morceau de programme dont l'exécution pourra être, au choix, déléguée à un ordinateur distant, ou effectuée par la machine locale.*

La décision de déléguer l'exécution d'un morceau délégable sera prise dynamiquement par la machine locale, selon la valeur de leur prédicat de délégation.

Définition 5 *Un prédicat de délégation est une fonction propre à un morceau délégable qui permet de décider si l'exécution de celui-ci doit être déléguée.*

Les prédicats de délégation doivent être élaborés durant la spécification du découpage du programme à partir des caractéristiques du morceau délégable auquel ils correspondent. Ils retournent une valeur booléenne et sont paramétrés par :
- l'énergie restante (lorsqu'il s'agit de systèmes dépendant d'une batterie) ;
- la charge de la machine locale ;
- la quantité de mémoire disponible ;
- l'interface réseau utilisée ;
- la bande passante vers le réseau ;
- le temps restant (lorsqu'il s'agit de programmes temps-réels).

1.3 Résultats

La spécification du découpage de programme produit les résultats suivants :
- une liste des descriptions des morceaux spécifiés ;
- une description des prédicats de délégation de chaque morceau délégable.

La description d'un morceau désigne :
- son identifiant (unique dans un programme) ;
- le nom du programme dont il fait partie ;
- ses adresses de début et de fin ;
- son type (local ou délégable) ;
- l'identifiant de son prédicat de délégation s'il s'agit d'un morceau délégable.

Ces descriptions constituent, en plus du code source du programme, les données d'entrée de l'analyse de la distribution.

2 Analyse de la distribution

L'analyse de la distribution des traitements consiste à déterminer :
- les données à communiquer entre les tâches ;
- la localisation des points de synchronisation.

2.1 Identification des données à communiquer

Soient X et Y deux morceaux différents du même programme et soient A et B des tâches instanciant respectivement X et Y. Soient i_A et i_B des instructions des morceaux X et Y respectivement, telles que l'exécution de i_B soit postérieure à l'exécution de i_A dans le programme d'origine. Une donnée doit être communiquée entre les deux tâches A et B si elle est définie (ou produite) par i_A et utilisée (ou consommée) par i_B sans qu'une instruction exécutée entre i_A et i_B dans le programme d'origine ne la produise à nouveau.

Cette problématique, qui s'apparente à la notion de dépendances de données, s'exprime en termes de vivacité de données (ou d'analyse de variables actives). La notion de base est

l'intervalle de vivacité : pour une donnée, il débute lors d'une production de variable et finit à la dernière consommation correspondant à cette production.

Plusieurs consommations pouvant correspondre à une production, sans qu'il existe de relation chronologique d'exécution entre ces consommations, nous substituons la notion de *pieuvre* à celle d'intervalle.

Définition 6 *Une pieuvre est constituée d'une production de donnée et des consommations qui lui correspondent.*

Inversement, plusieurs productions de donnée peuvent correspondre à une consommation, d'où l'introduction de la notion de *pool* de *pieuvre*s.

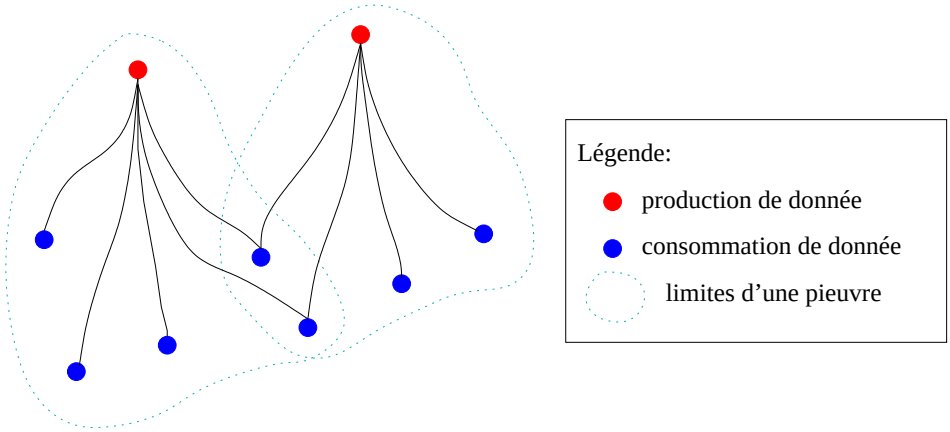

FIG. 3 – Exemple d'un *pool* constitué de deux *pieuvres*.

Définition 7 *Un pool regroupe toutes les pieuvres concernant la même donnée qui ont des consommations en commun.*

Un *pool* prend la forme d'un graphe biparti dans lequel un ensemble de productions est relié à un ensemble de consommations d'une donnée (cf l'exemple de la figure 3).

La problématique s'exprime maintenant de la façon suivante : une donnée `z` doit être communiquée d'une tâche A vers une tâche B s'il existe un arc d'un *pool* de `z` reliant une production située dans une tâche A à une consommation située dans une tâche B.

Comme la constitution des ensembles de données dépend du caractère obligatoire des productions de données, nous introduisons quelques concepts avant de la détailler.

Considérons les branches d'arbre de syntaxe abstraite qui sont des fils d'une structure de contrôle de flot. Il y a deux types de branches :

- les branches *optionnelles* : ce sont celles qui ne sont pas toujours exécutées car leur exécution dépend d'un test conditionnel ;
- les branches *obligatoires* : ce sont les autres branches. Elles sont obligatoirement exécutées lorsque le morceau auquel elles appartiennent est exécuté car elles sont incluses dans tous leurs chemins d'exécution.

Dans un premier temps, il faut étudier toutes les structures de contrôle de flot conditionnelles du programme d'origine afin de déterminer leur adresse et le nombre de branches qu'elles engendrent. De plus, afin d'identifier avec exactitude les données qui doivent être communiquées, il faut prendre en compte la symétrie des structures de contrôle de flot conditionnelles par rapport à leurs productions.

Proposition 1 *Une structure de contrôle de flot conditionnelle, incluse dans une tâche* $\boldsymbol{T}$*, est symétrique par rapport à une de ses productions si et seulement si :*

1. *la production concerne une donnée* $\boldsymbol{x}$ *dont au moins une consommation est située dans une autre tâche que la tâche* $\boldsymbol{T}$*;*
2. *toutes les branches, issues de cette structure de contrôle de flot et différentes de celle correspondant au test conditionnel, produisent au moins une fois la donnée* $\boldsymbol{x}$*.*

Toute autre structure de contrôle de flot conditionnelle est dite asymétrique par rapport à la production de $\boldsymbol{x}$*.*

Proposition 2 *Une structure de contrôle de flot conditionnelle, incluse dans une tâche* $\boldsymbol{T}$*, est symétrique par rapport à ses productions si et seulement si elle est symétrique par rapport à chaque production dont au moins une consommation associée est située dans une tâche différente de la tâche* $\boldsymbol{T}$*. Toute autre structure de contrôle de flot conditionnelle est dite asymétrique par rapport à ses productions.*

Exemple 3 Le schéma de la figure 4 représente un exemple (en C) de deux structures de contrôle de flot conditionnelles imbriquées. Supposons que ce traitement proviennent d'une tâche T. Si les données $\boldsymbol{x}$, et $\boldsymbol{y}$ sont consommées dans des tâches qui sont différentes de la tâche T et qui sont exécutées après cette dernière, tandis que la donnée $\boldsymbol{z}$ n'est consommée que dans des tâches dont l'exécution précède celle de la tâche T, alors :

- la structure de contrôle de flot conditionnelle imbriquée, celle qui est introduite par le mot clé ***if***, est symétrique par rapport à la production de $\boldsymbol{y}$. De plus, comme c'est la seule donnée produite dans cette structure de contrôle de flot conditionnelle (et consommées dans des tâches exécutées après la tâche T), celle-ci est symétrique par rapport à toutes ses productions.
- la structure de contrôle de flot conditionnelle englobante, celle qui est introduite par le mot clé ***switch***, est :
  - symétrique par rapport à la production de la donnée $\boldsymbol{x}$;
  - asymétrique par rapport à la production de la donnée $\boldsymbol{y}$.

  Par conséquent cette structure de contrôle de flot conditionnelle est asymétrique par rapport à ses productions (elle serait symétrique par rapport à ses productions si la donnée $\boldsymbol{y}$ n'était pas consommée dans des tâches exécutées après la tâche T).

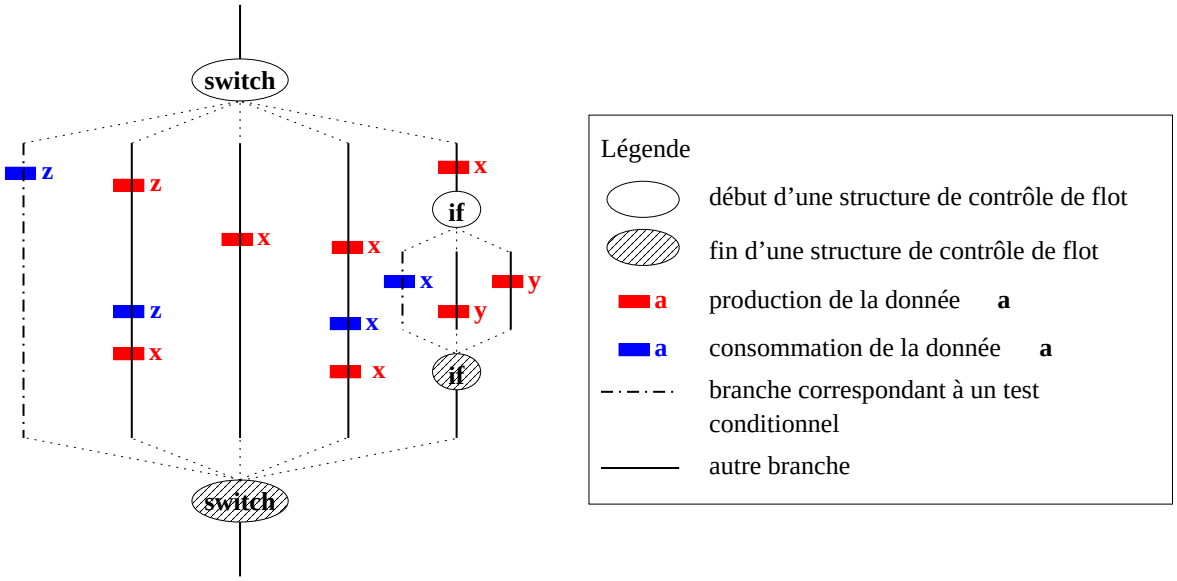

FIG. 4 – Exemple de structures de contrôle de flot conditionnelles en C.

Nous avons envisagé deux stratégies pour construire les ensembles de données à communiquer entre les tâches :

- la première consiste à créer des ensembles de données *approximatifs* contenant toutes les données potentiellement produites ;
- la seconde consiste à créer des ensembles de données *exacts* constitués des données effectivement produites.

Ces stratégies sont développées dans les deux prochaines sections puis nous détaillons un exemple de leur utilisation dans la section suivante. Enfin, nous exposons les modifications à effectuer dans ces méthodes afin de ne pas normaliser les tests conditionnels du code à analyser (cette normalisation a été décrite au début de la section 1).

2.1.1 Ensembles de données approximatifs

Ces ensembles de données sont relativement simples à constituer cependant ils peuvent augmenter le volume des communications.

Algorithme de construction du graphe approximatif des communications
Entrées :
– VIV : la liste des éléments *pool* décrivant la vivacité des données ;
– LDT : la liste des tâches.
Sortie :
– GRAPHE : le graphe approximatif des données à communiquer entre les différentes tâches du programme.

```
$(GRAPHE) := Graphe-approximatif-des-données-à-communiquer ($(VIV), $(LDT));

DEBUT_ALGO [Graphe-approximatif-des-données-à-communiquer]
initialiser $(GRAPHE);  /* $(GRAPHE) est un graphe sans arc contenant autant de
                           sommets qu'il y a de morceaux différents. */
POUR (chaque élément pool PO ∈ $(VIV)) REP
  AVEC DO: la donnée concernée par $(PO);
  POUR (chaque élément pieuvre PI ∈ $(PO)) REP
    AVEC TPROD: la tâche d'où provient la production incluse dans $(PI);
    POUR (chaque tâche TA ∈ $(LDT)) REP
      SI (l'adresse d'une des consommation incluse dans $(PI) est comprise entre
          les adresses de début et de fin de $(TA))
      ALORS
        SI (($(TPROD) et $(TA) représentent la même tâche incluse dans le corps
             d'une boucle ou d'un nid de boucles) et
            ($(TPROD) et $(TA) correspondent à des itérations différentes))
        ALORS
          ajouter le vecteur d'itérations adéquat au sommet correspondant à $(TPROD)
            dans $(GRAPHE), s'il n'y était pas déjà;
          insérer dans $(GRAPHE) un sommet correspondant à $(TA) avec son vecteur
             d'itérations respectif, s'il n'y était pas déjà;
        FIN_SI
        SI ($(TPROD) et $(TA) correspondent à des sommets différents dans $(GRAPHE))
        ALORS
          ajouter dans $(GRAPHE) un arc valué par $(DO) du sommet correspondant à
             $(TPROD) vers celui correspondant à $(TA);
        FIN_SI
      FIN_SI
    FIN_REP
  FIN_REP
FIN_REP
FIN_ALGO
```

Pour élaborer les ensembles de données approximatifs, il faut déterminer la symétrie des structures de contrôle de flot conditionnelles par rapport à chacune de leurs productions puis il faut construire un graphe, dit graphe approximatif des communications, en suivant l'algorithme

spécifié précédemment. Les sommets de ce graphe sont les tâches auquels les vecteurs d'itérations sont rajoutés lorsque les tâches correspondent à des morceaux inclus dans des boucles ou des nids de boucles. Les arcs du graphe sont orientés de la tâche représentant le producteur vers celle qui consomme la donnée concernée et sont valués par cette donnée.

De part sa construction, le graphe approximatif des données à communiquer ne peut pas contenir de boucles. Nous pouvons donc en déduire simplement des sous-ensembles de données *optionnels* et *obligatoires* grossiers ; pour cela, il faut répartir les données valuant le graphe en fonction du caractère optionnel ou obligatoire de la branche dans laquelle elles sont produites.

Proposition 3 *Un sous-ensemble de données optionnel est un ensemble de données produit dans une tâche* **T** *et consommé dans des tâches différentes de* **T**, *et dont toute donnée* `x` *est produite dans une même branche optionnelle qui n'est pas issue d'une structure de contrôle de flot conditionnelle symétrique par rapport à la production de* `x`.

Proposition 4 *Un sous-ensemble de données obligatoire est un ensemble de données produit dans une tâche* **T** *et consommé dans des tâches différentes de* **T**, *et dont toute donnée* `x` *est produite dans une branche obligatoire ou dans une branche optionnelle issue d'une structure de contrôle de flot conditionnelle symétrique par rapport à la production de* `x`.

Remarque 5 *Dans certains « cas pathologiques » (cf exemple suivant) les ensembles de données approximatifs peuvent générer des* fausses dépendances *entre des tâches.*

Exemple 4 Soient T_1, T_2 et T_3 trois tâches issues d'un même programme telles que :

1. l'ordonnancement est le suivant : T_1 précède T_2 qui précède T_3 (i.e. $T_1 \preceq T_2 \preceq T_3$) ;
2. T_1 produit la donnée `x` dans une branche obligatoire ou dans une branche optionnelle issue d'une structure de contrôle de flot conditionnelle symétrique par rapport à la production de `x` ;
3. T_2 ne consomme pas cette donnée `x`, mais, elle peut la produire dans une branche optionnelle issue d'une structure de contrôle de flot conditionnelle asymétrique par rapport à la production de `x` ;
4. T_3 consomme la donnée `x`.

Dans ce cas, la donnée `x` appartient à l'ensemble de données envoyé par T_1 à T_3 ainsi qu'à celui envoyé par T_2 à T_3. Cependant elle doit également appartenir à l'ensemble de données approximatif produit par T_1 et consommé par T_2. Sans cela, quand T_2 ne produit pas la donnée `x`, T_2 doit communiquer à T_3 la donnée `x` sans connaître sa valeur.

Par conséquent, après avoir construit le graphe approximatif des communications, il faut appliquer l'algorithme spécifié dans la suite. Celui-ci procède en deux étapes :

1. il met à jour les sous-ensembles de données optionnels et obligatoires afin de prendre en compte les *fausses* dépendances ;
2. il construit les ensembles de données approximatifs en faisant l'union des sous-ensembles de données optionnels et obligatoires adéquats.

Notation Soient T_1 et T_2 deux tâches différentes d'un même programme telles que la tâche T_1 soit exécutée avant la tâche T_2 (i.e. $T_1 \preceq T_2$).

- *SEDoptio*$[T_1 \rightarrow T_2]$ est le sous-ensemble de données optionnel envoyé par T_1 à T_2 ;
- *SEDoblig*$[T_1 \rightarrow T_2]$ est le sous-ensemble de données obligatoire envoyé par T_1 à T_2.

Algorithme pour concevoir les ensembles de données approximatifs

Entrées :

- OPTIO : la liste des sous-ensembles de données optionnels obtenus à partir du graphe approximatif des communications ;
- OBLIG : la liste des sous-ensembles de données obligatoires obtenus à partir du graphe approximatif des communications ;
- GRAPHE : le graphe approximatif des communications précédemment établi.

Sortie :

- LEDA : la liste des ensembles de données approximatifs du programme.

```
$(LEDA) := Liste-des-ensembles-de-données-approximatifs ($(OPTIO), $(OBLIG), $(GRAPHE));

DEBUT_ALGO
/* Mise-à-jour des sous-ensembles de données optionnels et obligatoires. */
AVEC SOMMETS := {S ∈ $(GRAPHE)};
     TMP := ∅;
TANT QUE ($(SOMMETS) ≠ ∅) REP
  $(TMP) := {S ∈ $(SOMMETS) | ∀ S′ ∈ $(SOMMETS), ∄ (S, S′) ∈ $(GRAPHE)};
  POUR (tout sommet X ∈ $(TMP)) REP
    AVEC PRED := {Y ∈ $(SOMMETS) | ∃ ($(Y), $(X)) ∈ $(GRAPHE)};
    POUR (tout sommet Y1 ∈ $(PRED)) REP
      POUR (tout sommet Y2 ∈ $(PRED) | $(Y1) ⪯ $(Y2)) REP
        AVEC SED1 := SEDoptio[$(Y1)→$(X)] ⋂ SEDoptio[$(Y2)→$(X)];
             SED2 := SEDoblig[$(Y1)→$(X)] ⋂ SEDoptio[$(Y2)→$(X)];
        SEDoptio[$(Y1)→$(Y2)] := SEDoptio[$(Y1)→$(Y2)] ⋃ SED1;
        SEDoblig[$(Y1)→$(Y2)] := SEDoblig[$(Y1)→$(Y2)] ⋃ SED2;
      FIN_REP
    FIN_REP
  FIN_REP
  $(SOMMETS) := $(SOMMETS) \ $(TMP);
FIN_REP
/* Création des ensembles de données approximatifs. */
$(LEDA) := NULL;  /* liste vide. */
POUR (tout sommet Y1 ∈ $(GRAPHE)) REP
  POUR (tout sommet Y2 ∈ $(GRAPHE) tel que $(Y1) ⪯ $Y2) REP
    AVEC EDA[$(Y1)→$(Y2)] = SEDoptio[$(Y1)→$(Y2)] ⋃ SEDoblig[$(Y1)→$(Y2)];
    SI ($(EDA) ≠ ∅)
    ALORS
      insérer $(EDA) dans $(LEDA);
    FIN_SI
  FIN_REP
FIN_REP
FIN_ALGO
```

2.1.2 Ensembles de données exacts

Les ensembles de données décrits précédemment comprennent l'ensemble des données potentiellement produites et consommées. L'ensemble des données effectivement produites lors de l'exécution peut dépendre du contrôle de flot. Nous présentons dans cette section une manière de procéder permettant la prise en compte du contrôle de flot dans le calcul des ensembles de données.

Afin d'identifier avec exactitude les données qui doivent être communiquées, il faut déterminer la symétrie des structures de contrôle flot conditionnelles par rapport à leurs productions. Ainsi, en traçant l'exécution des branches optionnelles qui ne sont pas issues de structures de contrôle de flot conditionnelles symétriques par rapport à leurs productions, il sera possible de déterminer dynamiquement quelles productions ont été effectivement réalisées.

Les données produites dans les branches optionnelles qui ne sont pas issues de structures de contrôle de flot conditionnelles symétriques par rapport à leurs productions sont regroupées selon les morceaux qui vont les consommer en ensembles appelés *sous-ensembles de données produits*.

Lorsque la symétrie des structures de contrôle de flot conditionnelles par rapport à leurs productions est déterminée, les sous-ensembles de données produits peuvent être constitués. Cela nécessite la construction d'un graphe comme lors de la conception des ensembles de données approximatifs. Les sommets de ce graphe, dit graphe exact des communications, sont les branches du programme et ses arcs, qui sont orientés de la branche productrice vers la branche consommatrice, sont valués par la donnée concernée. Le graphe exact des communications est construit en suivant l'algorithme spécifié dans la suite.

Comme le graphe approximatif des communications, le graphe exact des données à communiquer ne peut pas contenir de boucles. Le sous-ensemble de données qui devra être communiqué entre deux branches issues de tâches différentes est constitué des données qui valuent les arcs entre la branche productrice et la branche consommatrice.

La base de l'ensemble de données à communiquer entre deux tâches T_1 et T_2 (avec T_1 et T_2 des tâches appartenant au même chemin d'exécution, et, l'exécution de T_1 précèdant celle de T_2) est constitué des données produites dans des branches de T_1 qui ne sont pas des branches optionnelles issues de structures de contrôle de flot conditionnelles asymétriques par rapport à leurs productions. A cet ensemble de données de base s'ajouteront les sous-ensembles de données produits dans les branches de T_1 qui sont optionnelles et issues de structures de contrôle symétriques par rapport à leurs productions, en fonction de leur exécution. L'union des sous-ensembles de données effectivement produits doit être réalisée au niveau des points de synchronisation, ce qui complique un peu leur mise en oeuvre.

Algorithme de construction du graphe exact des communications

Entrées :
- VIV : la liste des éléments pool décrivant la vivacité des données ;
- LDB : la liste des branches avec leurs propriétés (leurs adresses de début et de fin ainsi que leur symétrie lorsqu'elles proviennent de structures de contrôle de flot conditionnelles).

Sortie :
- GRAPHE : le graphe exact des données à communiquer entre les différentes tâches du programme.

```
$(GRAPHE) := Graphe-exact-des-données-à-communiquer ($(VIV), $(LDB));

DEBUT_ALGO [Graphe-exact-des-données-à-communiquer]
initialiser $(GRAPHE);  /* $(GRAPHE) est un graphe sans arc contenant autant de
                           sommets qu'il y a de branches différentes. */
POUR (chaque élément pool PO ∈ $(VIV)) REP
  AVEC DO: la donnée concernée par $(PO);
  POUR (chaque élément pieuvre PI ∈ $(PO)) REP
    AVEC BPROD: la branche d'où provient la production incluse dans $(PI);
    POUR (chaque branche BR ∈ $(LDB)) REP
      SI (l'adresse d'une des consommation incluse dans $(PI) est comprise entre
          les adresses de début et de fin de $(BR))
      ALORS
        SI (($(BPROD) et $(BR) sont incluse dans le corps de la même boucle ou
             du même nid de boucles) et
            ($(BPROD) et $(BR) correspondent à des itérations différentes))
        ALORS
          ajouter le vecteur d'itérations adéquat au sommet correspondant à $(BPROD)
             dans $(GRAPHE), s'il n'y était pas déjà;
          insérer dans $(GRAPHE) un sommet correspondant à $(BR) avec son vecteur
             d'itérations respectif, s'il n'y était pas déjà;
        FIN_SI
        SI (($(BPROD) et $(BR) correspondent à des sommets différents dans $(GRAPHE))
            et ($(BPROD) et $(BR) représentent des branches issues de tâches
                différentes))
        ALORS
          ajouter dans $(GRAPHE) un arc valué par $(DO) du sommet correspondant à
             $(BPROD) vers celui correspondant à $(BR);
        FIN_SI
      FIN_SI
    FIN_REP
  FIN_REP
FIN_REP
FIN_ALGO
```

2.1.3 Utilisation de ces stratégies sur un exemple

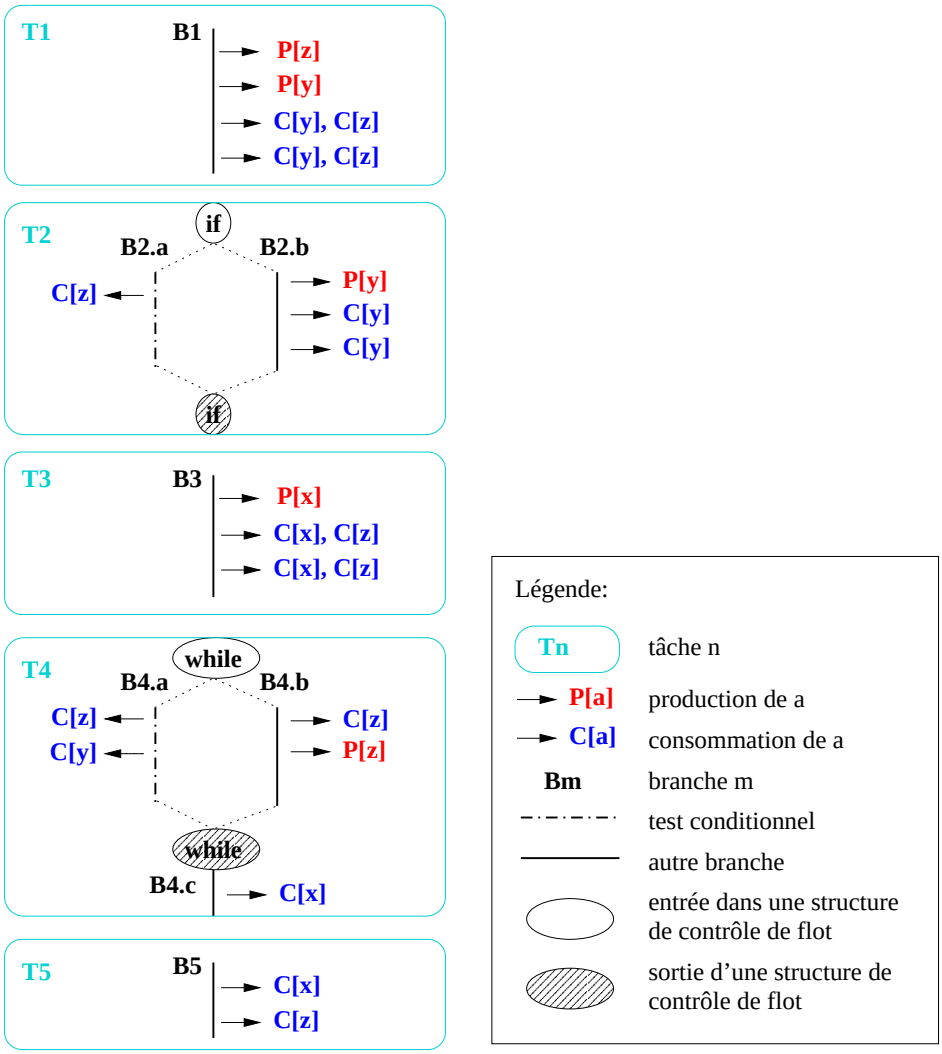

FIG. 5 – Exemple de programme partitionné.

Le schéma de la figure 5 décrit le code source d'un programme qui a été partitionné en cinq tâches. La communication des données entre les tâches peut être schématisée par des pools de pieuvres comme le montre la figure 6.

Dans un premier temps, nous allons déterminer les ensembles de données approximatifs qui doivent être communiqués entre les différentes tâches de ce programme, puis nous constituerons les ensembles de données exacts.

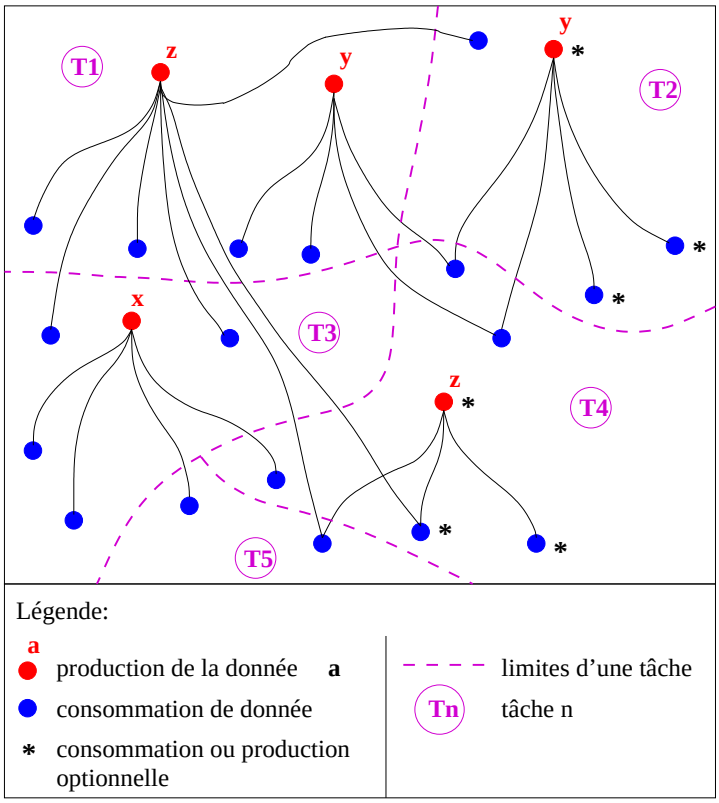

FIG. 6 – Exemple de programme partitionné.

Ensembles de données approximatifs correspondants

Nous construisons le graphe approximatif des communications en suivant l'algorithme donné dans la section 2.1.1. Le graphe obtenu pour le programme considéré est représenté sur la figure 7.

A partir de ce graphe, nous déduisons les sous-ensembles de données optionnels et obligatoires grossiers suivants :

- $SEDoblig[T_1 \rightarrow T_2] = \{z\}$
- $SEDoblig[T_1 \rightarrow T_3] = \{z\}$
- $SEDoblig[T_1 \rightarrow T_4] = \{y, z\}$
- $SEDoblig[T_1 \rightarrow T_5] = \{z\}$
- $SEDoptio[T_2 \rightarrow T_4] = \{y\}$
- $SEDoblig[T_3 \rightarrow T_4] = \{x\}$
- $SEDoblig[T_3 \rightarrow T_5] = \{x\}$
- $SEDoptio[T_4 \rightarrow T_5] = \{z\}$

De plus les sous-ensembles de données grossiers suivants sont tous vides :

- $SEDoptio[T_1 \rightarrow T_2] = \emptyset$
- $SEDoptio[T_1 \rightarrow T_3] = \emptyset$

- $SEDoptio[T_1 \rightarrow T_4] = \emptyset$
- $SEDoptio[T_1 \rightarrow T_5] = \emptyset$
- $SEDoblig[T_2 \rightarrow T_4] = \emptyset$
- $SEDoptio[T_3 \rightarrow T_4] = \emptyset$
- $SEDoptio[T_3 \rightarrow T_5] = \emptyset$
- $SEDoblig[T_4 \rightarrow T_5] = \emptyset$

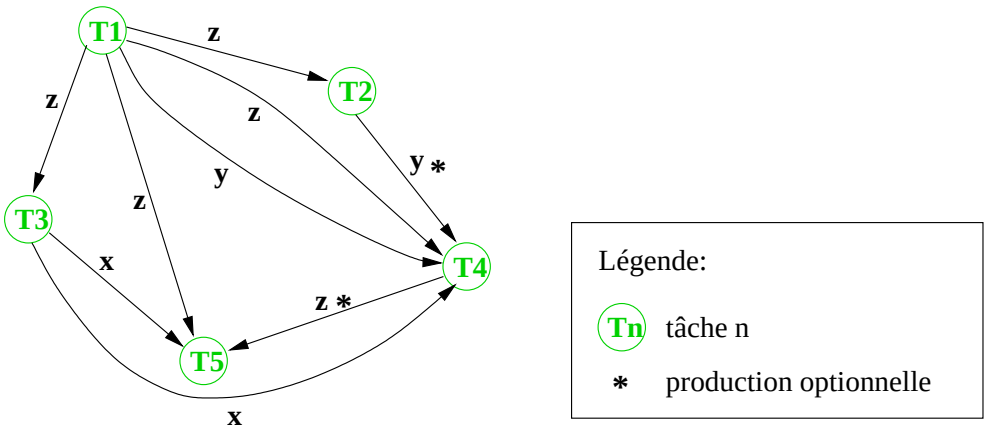

FIG. 7 – Graphe approximatif des communications correspondant au programme de la figure 5.

En appliquant l'algorithme pour concevoir les ensembles de données approximatifs spécifié dans la section 2.1.1, on obtient :

- $ED_{approx}[T_1 \rightarrow T_2] = \{y, z\}$
- $ED_{approx}[T_1 \rightarrow T_3] = \{z\}$
- $ED_{approx}[T_1 \rightarrow T_4] = \{y, z\}$
- $ED_{approx}[T_1 \rightarrow T_5] = \{z\}$
- $ED_{approx}[T_2 \rightarrow T_4] = \{y\}$
- $ED_{approx}[T_3 \rightarrow T_4] = \{x\}$
- $ED_{approx}[T_3 \rightarrow T_5] = \{x\}$
- $ED_{approx}[T_4 \rightarrow T_5] = \{z\}$

Remarque 6 *Une fausse dépendance portant sur la donnée* $\mathbf{y}$ *a été introduite entre les tâches* T_1 *et* T_2*. Cela provient du fait que la tâche* T_2 *ne consomme pas cette donnée mais elle la produit, et ceci, uniquement dans une branche optionnelle qui n'est pas issue d'une structure de contrôle de flot conditionnelle symétrique.*

Ensembles de données exacts correspondants

Nous construisons le graphe exact des communications en suivant l'algorithme donné dans la section 2.1.2. Le graphe obtenu pour le programme considéré est représenté sur la figure 8.

A partir de ce graphe, nous déduisons les sous-ensembles de données optionnels et obligatoires grossiers suivants :

- $SED[B_1 \rightarrow B_{2.a}] = \{z\}$

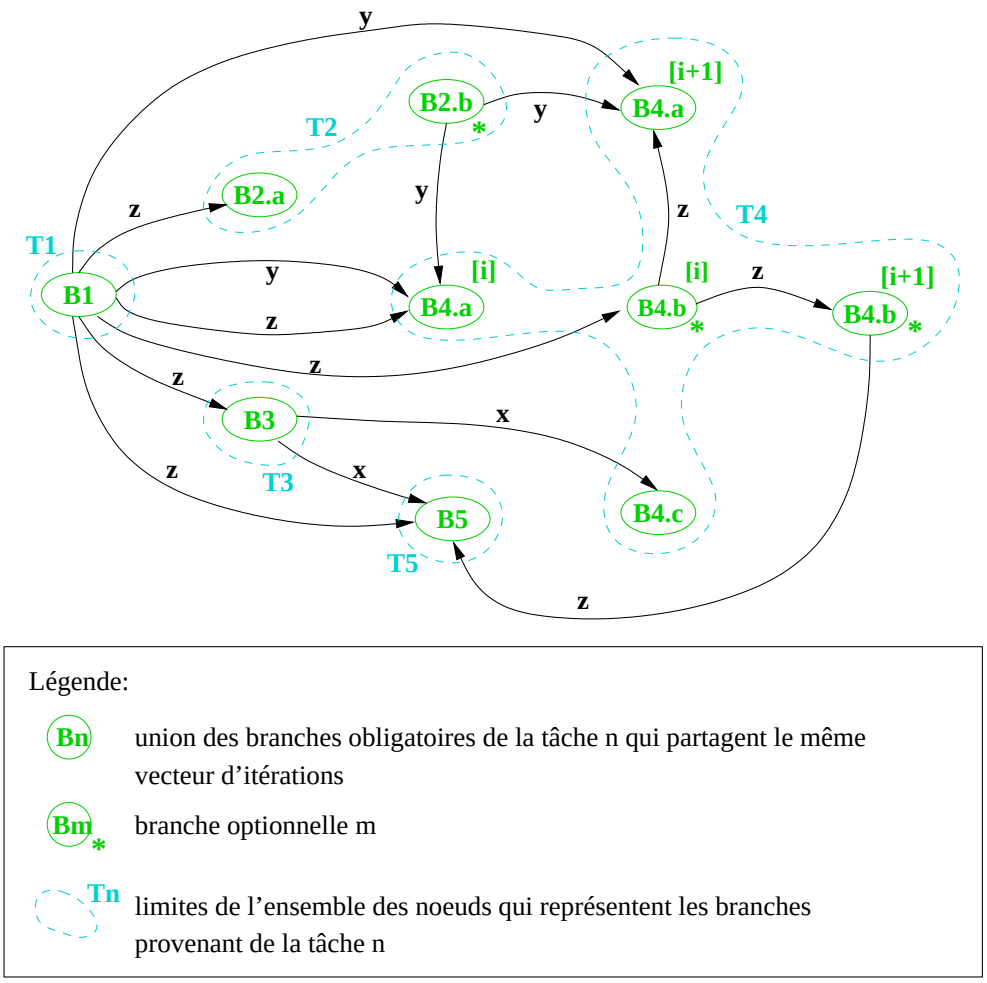

FIG. 8 – Graphe exact des communications correspondant au programme de la figure 5.

- $SED[B_1 \rightarrow B_3] = \{z\}$
- $SED[B_1 \rightarrow {B_{4.a}}^{[i]}] = \{y, z\}$
- $SED[B_1 \rightarrow {B_{4.b}}^{[i]}] = \{z\}$
- $SED[B_1 \rightarrow {B_{4.a}}^{[i+1]}] = \{y\}$
- $SED[B_1 \rightarrow B_5] = \{z\}$
- $SED[B_{2.b} \rightarrow {B_{4.a}}^{[i]}] = \{y\}$
- $SED[B_{2.b} \rightarrow {B_{4.a}}^{[i+1]}] = \{y\}$
- $SED[B_3 \rightarrow B_{4.c}] = \{x\}$
- $SED[B_3 \rightarrow B_5] = \{x\}$
- $SED[{B_{4.b}}^{[i+1]} \rightarrow B_5] = \{z\}$

Lorsque tous ces sous-ensembles de données sont déterminés, nous pouvons envisager toutes

les unions de ces sous-ensembles de données qui peuvent être effectivement produites. Voici la liste de ces unions :

- $ED_{exact}[T_1 \rightarrow T_2] = \{z\}$
- $ED_{exact}[T_1 \rightarrow T_3] = \{z\}$
- $ED_{exact}[T_1 \rightarrow T_4] = \{y, z\}$
- $ED_{exact}[T_1 \rightarrow T_5] = \{z\}$
- $ED_{exact}[T_2 \rightarrow T_4] = \begin{cases} \emptyset \\ \{y\} \end{cases}$ en fonction de l'exécution de la branche $B_{2.b}$.
- $ED_{exact}[T_3 \rightarrow T_4] = \{x\}$
- $ED_{exact}[T_3 \rightarrow T_5] = \{x\}$
- $ED_{exact}[T_4 \rightarrow T_5] = \{z\}$

2.1.4 Alternative à la normalisation des tests conditionnels

Il est possible de modifier légèrement les deux méthodes d'identification des données à communiquer précédemment décrites afin de ne pas transformer les tests conditionnels du code à analyser (cf début de la section 1).

Le fait de ne pas normaliser les tests conditionnels implique l'introduction d'un nouveau concept : le caractère obligatoire ou optionnel d'une instruction.

Définition 8 *Une instruction est dite obligatoire si elle appartient à tous les chemins d'exécution du morceau dont elle provient. Dans le cas contraire, elle est dite optionnelle.*

Par conséquent, toute instruction incluse dans une branche optionnelle est optionnelle, et, toute instruction incluse dans une branche obligatoire qui n'est pas un test conditionnel est une instruction obligatoire. Quant aux tests conditionnels, toutes les intructions qui les composent sont optionnelles hormis celle qui est atteignable en parcourant en profondeur et à gauche l'arbre syntaxique correspondant.

Exemple 5 Voici un exemple de test conditionnel en C :

```
...
if (((x = foo(y)) && (y > x) && (y = foo(x))) || ((x == y) && (x > 0)))
  ...
...
```

L'arbre syntaxique correspondant, en adoptant l'associativité à gauche pour les opérations booléennes binaires, est représenté sur la figure 9. La seule instruction obligatoire du précédent test conditionnel est celle qui coïncide avec la sous-expression *se_1*.

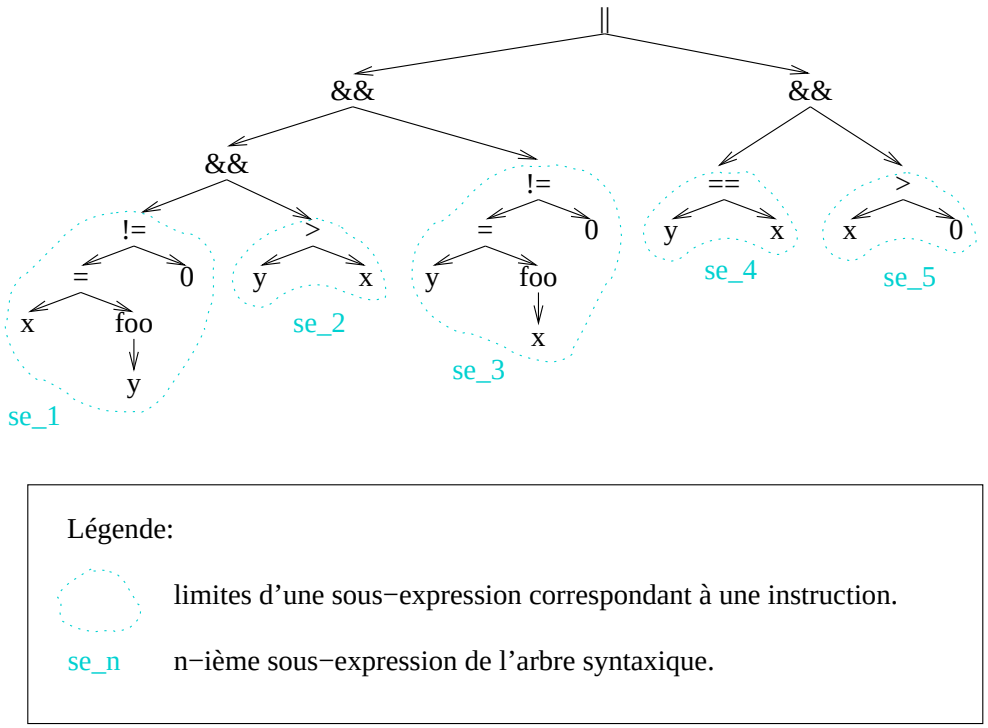

FIG. 9 – Arbre syntaxique du précédent test conditionnel.

La conception des ensembles de données approximatifs ne diffère de la méthode décrite dans la section 2.1.1 que par la manière de répartir les données valuant le graphe approximatif des communications. Ces dernières doivent être réparties en sous-ensembles de données optionnels et obligatoires en fonction de leur propre caractère optionnel ou obligatoire (et non plus selon le caractère optionnel ou obligatoire de la branche qui les contient).

En ce qui concerne l'élaboration des ensembles de données exacts, les différences par rapport à la méthode exposée dans la section 2.1.2 sont beaucoup plus significatives. Nous avons envisagé deux solutions :

1. soit, tracer l'exécution des instructions optionnelles qui sont incluses dans des tests conditionnels et qui produisent des données, afin de ne communiquer que les données effectivement produites ;
2. soit, introduire des fausses dépendances afin de limiter l'*overhead*, en particulier au niveau de la machine locale.

2.2 Localisation des points de synchronisation

Pour répartir les ensembles de données en respectant les dépendances entre les instances de morceaux d'un programme, il est nécessaire d'utiliser des mécanismes de synchronisation.

Les mécanisme de synchronisation choisis sont des *points de rendez-vous*. Ces derniers sont

de deux types :

- les *points de rendez-vous actifs* qui sont placés à la fin de la production d'un ensemble de données produit. Dès que ce type de point de rendez-vous est rencontré, l'ensemble de données produit auquel il est associé est envoyé aux machines qui exécutent des morceaux exploitant ces données.
- les *points de rendez-vous passifs* qui sont placés avant la première consommation d'une des données d'un ensemble de données consommé. Ces points de rendez-vous sont bloquants, il faut attendre de recevoir les ensembles de données qui leur sont associés pour pouvoir poursuivre l'exécution du morceau.

3 Réalisation du découpage

La réalisation effective du découpage exploite les résultats obtenus lors de la spécification du découpage et de l'analyse de la distribution. Elle consiste à :

- instrumenter le programme ;
- transformer le programme pour en produire deux versions :
  - l'une est spécifique à la machine locale ;
  - l'autre est destinée aux machines distantes.

3.1 Instrumentation du programme

L'instrumentation du programme d'origine est commune aux deux transformations ultérieures.

La localisation des points de rendez-vous a été déterminée lors de l'analyse de la distribution. Il suffit donc de positionner les points de rendez-vous actifs aux adresses correspondantes. En ce qui concerne les points de rendez-vous passifs, il faut les positionner aux adresses calculées en prenant garde de suivre l'ordonnancement des tâches afin que les éventuels écrasements de données, qui peuvent se produire lorsqu'une donnée appartient à plusieurs ensembles de données, soit réalisés dans l'ordre adéquat.

D'autre part, si les ensembles de données exacts sont utilisés, les points de rendez-vous actifs doivent tenir compte du traçage de l'exécution des branches optionnelles qui ne sont pas issues de structures de contrôle de flot conditionnelles symétriques par rapport à leurs productions afin de déterminer dynamiquement quels sous-ensembles de données ont été effectivement produits et donc quelles données doivent être communiquées.

3.2 Transformation spécifique à la machine locale

La transformation effectuée pour la machine locale est schématisée sur la figure 10. C'est la dernière opération avant la compilation qui aboutit à un programme exécutable uniquement par la machine locale.

Afin de tester la valeur du prédicat de délégation de chaque morceau délégable, il faut insérer, juste avant l'adresse de début du morceau considéré, une structure de contrôle de flot conditionnelle du type : `if(cond) bloc1 else bloc2;` (`cond` représente le prédicat en

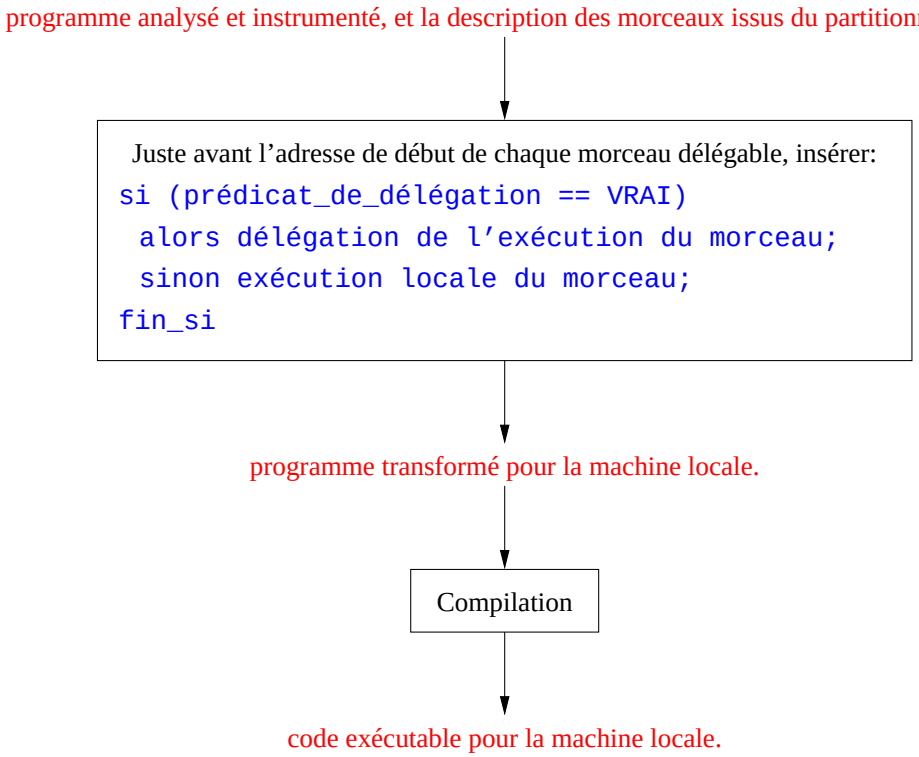

FIG. 10 – Transformation du code pour la machine locale.

question). Si la valeur du prédicat est vrai, alors l'exécution du morceau est déléguée par l'intermédiaire d'un agent — la délégation de traitement sera détaillée par la suite — (cela correspond à `bloc1`), sinon la machine locale exécute le morceau (elle exécute le code d'origine jusqu'à l'adresse de fin du morceau en tenant compte de l'instrumentation ; cela correspond à `bloc2`).

3.3 Transformation spécifique aux machines distantes

La possibilité de déléguer l'exécution de certains morceaux de programme implique l'utilisation d'un ensemble de machines distantes. Celles-ci doivent être accessibles par réseau depuis la machine locale (et l'utilisateur doit avoir un droit d'accès à ces ordinateurs). Il faut également que chacun de ces ordinateurs distants dispose du code correspondant aux morceaux délégables d'un programme. Ce code est élaboré par une machine ayant à sa disposition :

- un compilateur-croisé ;
- le code source correspondant au programme instrumenté ;
- les résultats des précédents modules.

La figure 11 montre la transformation de code effectuée pour tous les ordinateurs distants. Elle consiste à construire une fonction pour chacun des morceaux délégables d'un programme. L'identifiant de ces fonctions est identique sur tous les ordinateurs pour faciliter les appels dis-

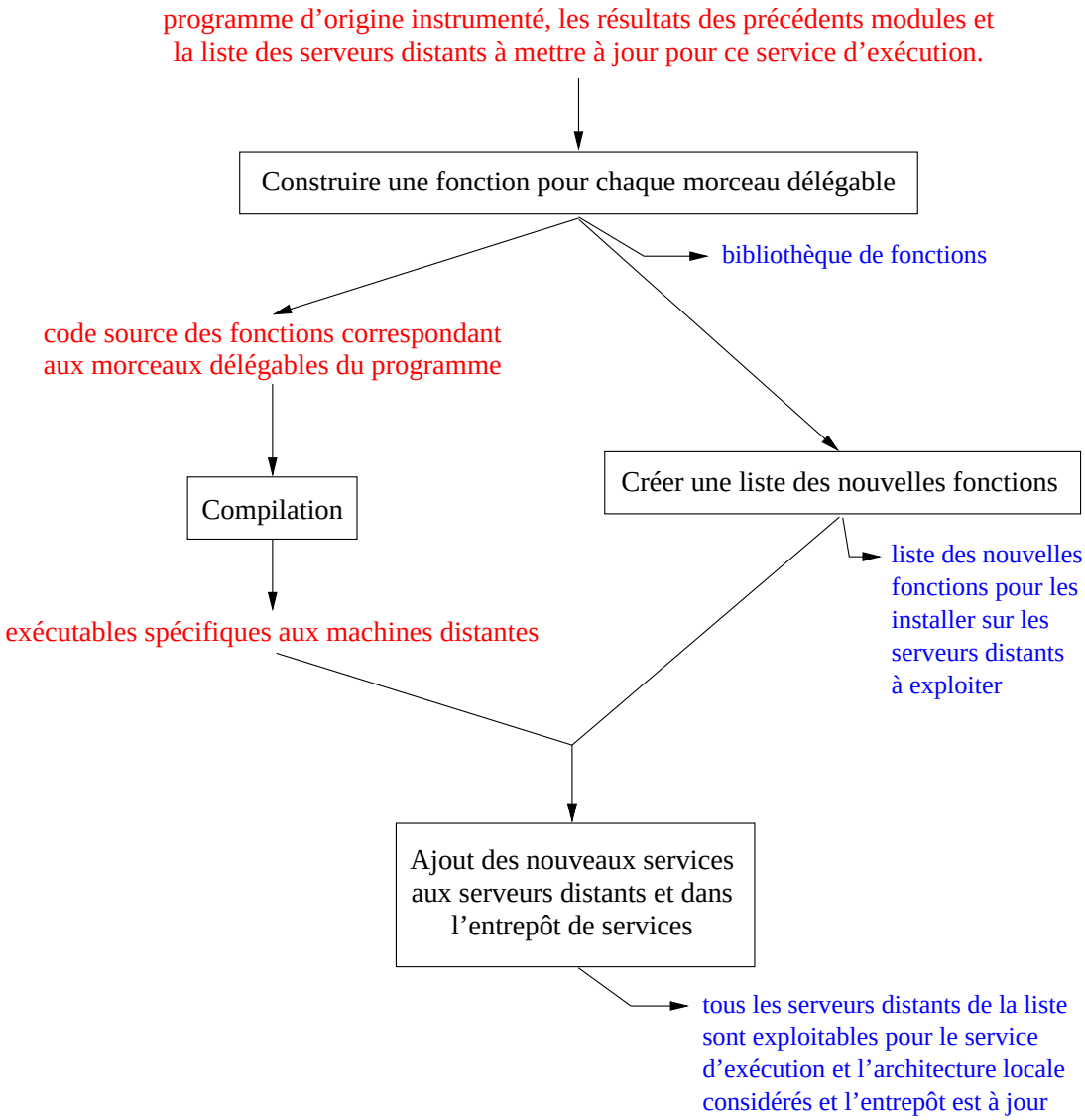

FIG. 11 – Transformation du code pour les machines distantes.

tants. Chaque ordinateur distant propose des services d'exécution[5], ils exécutent des morceaux de programmmes à la demande.

Au cours de cette transformation, une liste des services d'exécution existants est mise à jour ; cette liste est stockée dans un entrepôt[6]. Chaque ordinateur distant possède une liste similaire qui lui est propre et qui décrit les services d'exécutions connus. Ces listes permettent de savoir quels ordinateurs distants sont exploitables pour un service d'exécution donné. L'entrepôt

[5]Un service d'exécution correspond à l'ensemble des morceaux délégables d'un programme obtenus avec certaines caractéristiques de machine locale.

[6]Un entrepôt est en fait une base de données située sur un serveur accessible par réseau.

est mis à jour afin de stocker toutes les informations nécessaires à l'insertion de nouveaux interlocuteurs.

En adoptant cette méthode, on évite à la machine locale de devoir envoyer, à chaque exécution, le code source de chaque morceau délégué qui devrait alors être compilé à la volée (cela dégraderait des performances, d'autant que plusieurs appels à la même fonction nécessiteraient autant de communications et de compilations...).

4 Mise en place de la distribution

Afin de ne pas surcharger la *machine locale*, la multi-localisation du code à exécuter et de ses données est gérée par des agents [3].

Définition 9 *Un agent est une entité logicielle capable d'observer son environnement et de réagir en fonction des évênements perçus.*

Les agents sont des processus essentiels de notre mécanisme de délégation de traitements. Ils centralisent les informations nécessaires et choisissent, parmi un ensemble de machines exploitables, les machines distantes utilisées pour les délégations. Ce sont des agents intelligents et mobiles.

Définition 10 *Un agent intelligent utilise l'intelligence artificielle et ses connaissances pour raisonner et agir sur son environnement.*

Définition 11 *Un agent mobile peut se déplacer d'un système à un autre afin de tirer partie de la topologie du réseau et de se rapprocher des ressources qu'il contrôle.*

D'autre part, chaque *machine distante*, doit être considérée comme un seul interlocuteur, même si elle exécute simultanément plusieurs tâches, celles-ci pouvant provenir de différents services d'exécution. Par conséquent un processus dédié exclusivement aux communications, appelé ***gestionnaire de services***, est lancé sur chaque *machine distante* utilisée. Les *machines locales* doivent également être dotées de gestionnaire de services pour uniformiser les communications de données.

Les agents utilisés pour réaliser notre protocole de délégation d'exécutions sont les suivants :
- un *super-agent* ;
- des *agents de placement*.

4.1 Le super-agent

Il est la clé de voûte de notre système de délégation de traitements, il administre les agents de placement et les interlocuteurs, quand il y en a. Il gére tous les services d'exécutions d'un ensemble de machines. Tout nouvel interlocuteur (soit une *machine locale*, soit un *ordinateur distant*) doit se présenter au super-agent pour initier le mécanisme de délégation. Si le nouvel interlocuteur est une *machine locale*, le super-agent le met en relation avec l'agent de placement

adéquat, le service d'exécution voulu est alors lancé. Si le nouvel interlocuteur est un *ordinateur distant*, tous les agents de placement sont informés de son arrivée et des services d'exécution qu'il propose, ensuite il est exploitable par toutes les *machines locales* déclarées.

D'autre part, le super-agent est également chargé d'instancier des agents de placement au gré des besoins, c'est à dire lorsque les limites de leurs capacités sont atteintes (avant que des goulets d'étranglement ne se forment et ne viennent ralentir les délégations), et de les supprimer lorsqu'ils ne sont plus utilisés. Le super-agent, de par son statut particulier, ne doit jamais s'arrêter, ce doit être un processus démon et, pour prendre des précautions vis-à-vis des pannes, il faudrait conserver une copie de ses informations (par exemple en utilisant deux super-agents — s'exécutant sur des machines différentes — comme les serveurs DNS).

Les informations mémorisées par le super-agent sont les suivantes :

- la liste des *machines locales* déclarées avec, pour chacune d'elles, la liste des *machines distantes* exploitées et la liste des services d'exécution connus ;
- la liste des *ordinateurs distants* déclarés avec, pour chacun d'eux, la liste des services d'exécution connus ;
- une liste exhaustive des agents de placement ;
- une liste de machines pouvant héberger des instances d'agent de placement ou de super-agent.

4.2 Les agents de placement

Ils gèrent la multilocalisation du code et des données. Pour ne pas provoquer de goulet d'étranglement à leur niveau, la capacité des agents de placement est bornée en termes de nombre et de taille de services d'exécution actifs.

Le rôle des agents de placements est de stocker, de manière centralisée, toutes les informations nécessaires pour diriger les flux de données entre les morceaux des services d'exécution qu'ils supervisent. C'est essentiel car lorsqu'une exécution déléguée est lancée, on ne sait pas encore quels ordinateurs vont exécuter les morceaux de programme qui vont consommer les données produites pendant la délégation. Par conséquent, chaque agent de placement doit :

- conserver une table montrant les communications des ensembles de données entre les morceaux de chaque service d'exécution et donnant le nombre de consommateurs de chaque ensemble de données produit ;
- maintenir une table décrivant les *ordinateurs distants* exploitables et pour chacun d'eux les services d'exécution proposés et ceux en cours d'exécution ;
- stocker la liste des *machines locales* supervisées avec pour chacune d'elles les identifiants des services d'exécution en cours ;
- mémoriser diverses informations relatives aux morceaux en cours d'exécution afin de gérer les points de rendez-vous. Ces informations sont :
  - l'identifiant du morceau en cours d'exécution ;
  - l'identifiant du service d'exécution dont le morceau provient ;
  - l'adresse IP de l'ordinateur sélectionné pour exécuter ce morceau.
- garder à jour une liste d'attente des producteurs contenant :
  - l'adresse IP de l'ordinateur,

– la liste des identifiants des ensembles de données et le nombre de consommateurs restant à satisfaire pour chacun d'eux.

De plus, comme le montre la figure 12, l'agent de placement est également chargé du placement des morceaux de programme délégables. C'est lui qui sélectionne dynamiquement un ordinateur pour chaque délégation de traitement et non la *machine locale* afin de ne pas la surcharger.

L'intelligence de nos agents de placement devra leur permettre de déterminer la machine distante la plus adéquate à exécuter une tâche parmi un ensemble de machines accessibles. Ces agents seront également mobiles afin de migrer vers l'une des machines disponibles les plus proches de leurs interlocuteurs respectifs.

4.3 Synchronisation grâce aux points de rendez-vous

4.3.1 Points de rendez-vous actifs

Dès qu'un point de rendez-vous actif (après la production de la dernière donnée d'un ensemble de données produit) est rencontré au cours de l'exécution d'un morceau de programme, l'ordinateur qui exécute ce morceau envoie à l'agent de placement une requête contenant l'identifiant de l'ensemble de données produit pour savoir où l'envoyer.

L'agent de placement consulte ses tables et renvoie les adresses IP des machines qui vont consommer cet ensemble de données ; le producteur peut alors faire parvenir l'ensemble de données produit aux destinataires déjà connus. Si le nombre de ces identifiants est inférieur au nombre total de consommateurs de cet ensemble de données, alors l'agent de placement ajoute le producteur à la liste d'attente pour cet ensemble de données. Lorsqu'il est mis sur une liste d'attente, le gestionnaire de service du producteur, qui est prévenu par l'agent de placement, doit stocker l'ensemble de données concerné tant qu'il n'a pas été envoyé à tous ses consommateurs respectifs.

L'agent de placement consulte la liste d'attente des producteurs dès qu'il enregistre le lancement de l'exécution d'un morceau délégable afin de trouver les consommateurs manquant pour renseigner les producteurs en attente au plus vite. Dès que tous les consommateurs ont été statisfaits, l'agent de placement supprime le producteur de la liste d'attente pour l'ensemble de données concerné et prévient le producteur qui n'a plus besoin de stocker l'ensemble de données produit. Les point de rendez-vous actifs terminent au fûr et à mesure que les points de rendez-vous passifs qui leurs sont associés terminent.

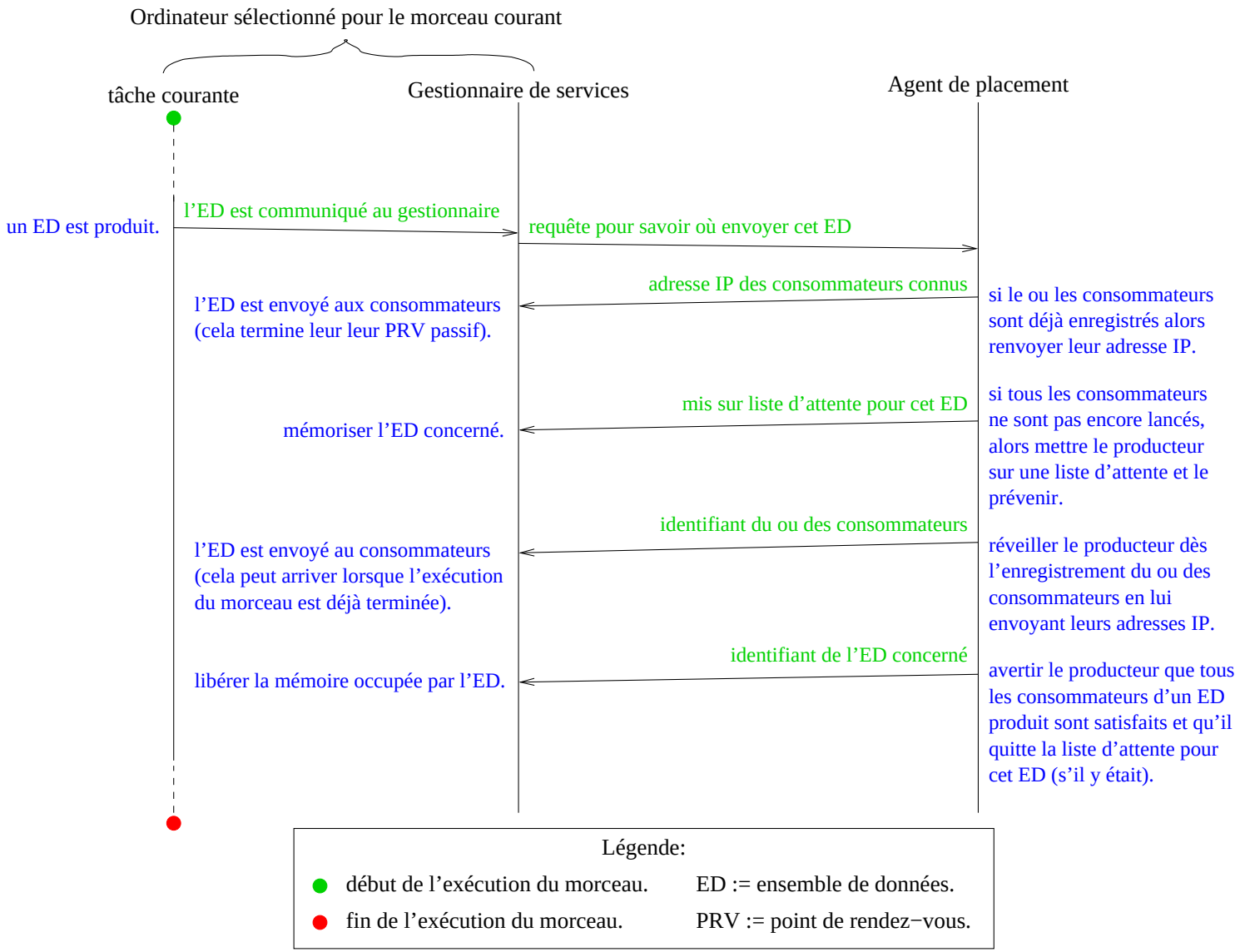

FIG. 12 – Gestion des points de rendez-vous par l'agent de placement.

4.3.2 Points de rendez-vous passifs

Lorsqu'un point de rendez-vous passif (avant la première utilisation d'une donnée appartenant à un ensemble de données pas encore consommé) est rencontré au cours de l'exécution (locale ou déléguée) d'un morceau, celle-ci est interrompue jusqu'à la réception de l'ensemble de données correspondant.

4.4 Exécution des morceaux de programmes

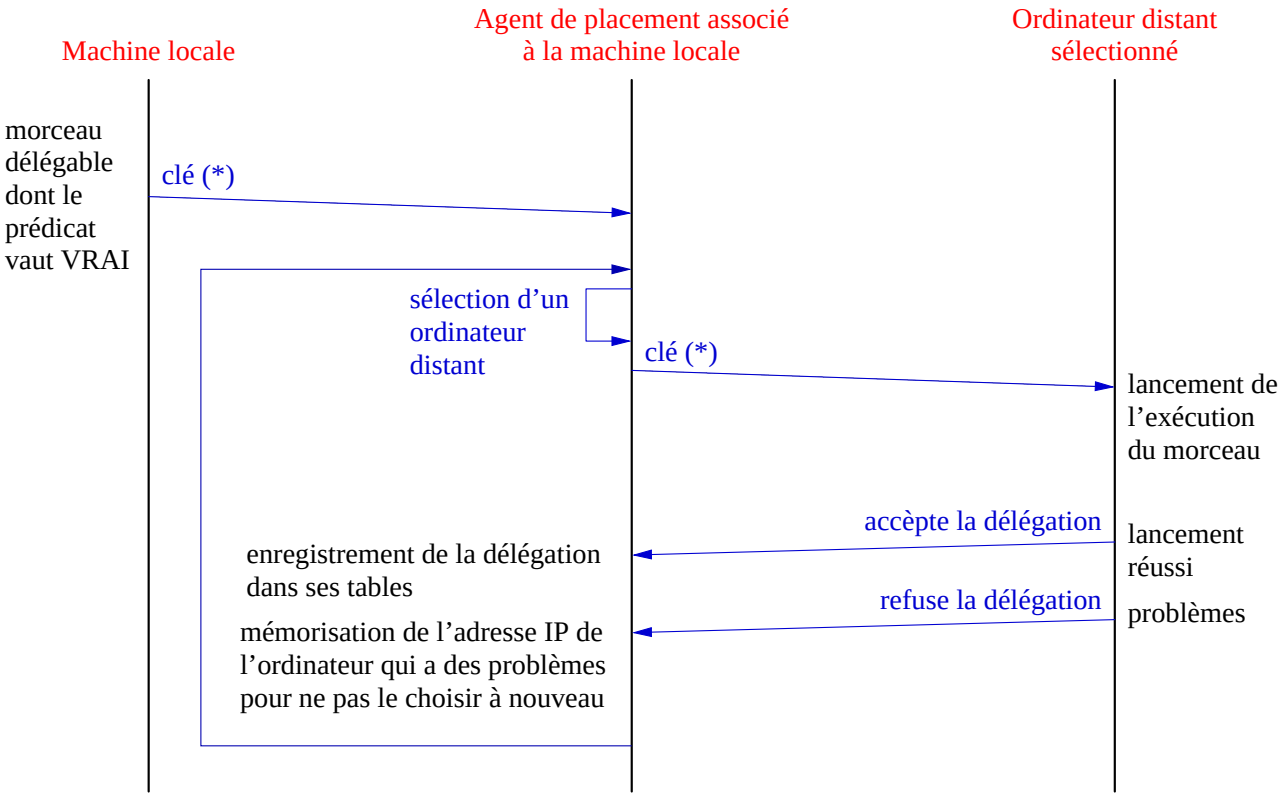

FIG. 13 – Lancement d'une exécution déléguée.

L'ordonnancement des morceaux d'un programme est réalisé statiquement lors de la phase d'analyse de la distribution en déterminant les points de rendez-vous qui identifient les dépendances entre les morceaux. La décision de déléguer un morceau délégable est dynamique, elle est prise par la machine locale selon la valeur du prédicat de délégation du morceau au moment où il doit être exécuté.

Tant qu'il s'agit d'un morceau délégable et que la valeur de son prédicat de délégation est ***vrai***, la machine locale envoie à son agent de placement une requête contenant son identifiant,

celui du morceau à déléguer et celui du service d'exécution dont il provient, avant de passer au morceau suivant. C'est l'agent de placement qui réalise le placement effectif du morceau délégué, comme le montre la figure 13. Il sélectionne un ordinateur distant exploitable et lui envoie les identifiants reçus pour lancer l'exécution. Quand un ordinateur distant reçoit une telle requête de l'agent de placement, il lance l'exécution de la tâche correspondante puis, si le lancement s'est déroulé normalement, il renvoie un message d'acceptation à l'agent de placement. Dans le cas contraire, il lui envoie un message de refus.

Lorsque l'agent de placement reçoit un message d'acceptation d'un ordinateur distant, il l'enregistre dans ses tables et prévient les producteurs concernés dans la file d'attente, s'il y en a, qu'ils peuvent envoyer leur productions. Par contre quand l'agent de placement reçoit un message de refus (cela signifie que l'ordinateur initialement sélectionné a des problèmes) ou s'il ne reçoit aucun message après un certain délai, il mémorise l'identifiant de l'ordinateur précédemment sélectionné, afin de ne pas le choisir à nouveau, puis il choisit un autre ordinateur à qui il soumet la requête.

Une fois lancée, l'exécution déléguée d'un morceau se synchronise avec les autres exécutions grâce aux points de rendez-vous.

Lorsqu'il s'agit d'un morceau local ou d'un morceau délégable dont la valeur du prédicat de délégation est ***faux***, le morceau est exécuté par la machine locale et le placement des morceaux suivants est suspendu jusqu'à la fin de l'exécution locale.

Remarque 7 *Lors d'une exécution locale, la machine locale ne cherche pas à placer les morceaux délégables suivants car les paramètres de leur prédicat de délégation (i.e. le contexte d'exécution futur) ne sont pas encore connus. De plus, les valeurs de ces paramètres pouvant varier rapidement, l'utilisation d'une estimation diminuerait le degré de réalisme du procédé et cela surchargerait la machine locale...*

Deuxième partie

Implémentation

Étant donnée l'ampleur du projet par rapport à la durée d'un stage de DEA, sa réalisation est loin d'être terminée (par exemple le compilateur McCAT a été développé activement pendant presque dix ans par de nombreux chercheurs...). La plupart des étapes de notre projet de délégation de traitements décrites dans la première partie n'ont pas été implémentées mais nous les avons globalement spécifiées. Nous nous sommes surtout intéressés à l'analyse du code source qui est indispensable à notre mécanisme de délégation de traitements.

1 Choix d'implémentation

Dans cette section, nous présentons les principaux choix que nous avons fait pour réaliser l'implémentation ainsi que les technologies existantes que nous avons décidé d'utiliser.

1.1 Langage de programmation considéré

Pour débuter, nous avons décidé de nous intéresser uniquement à la délégation de traitements issus de programmes écrits en langage C[6] et respectant la norme ANSI. Par la suite nous pensons étendre notre mécanisme de délégation aux programmes codés avec des langages orientés objets comme C++, objective C (C#) ou Java.

D'autre part, nous avons quelque peu réduit le langage C pour simplifier le processus d'analyse. Les restrictions que nous imposons sont les suivantes :

- l'exécution d'un programme ne doit lancer qu'un seul processus, i.e. les threads et les processus fils ne sont pas permis ;
- l'usage des labels et du mot clé `goto` est proscrit tant que nous ne procédons pas à une élimination des `goto` afin d'éviter les « codes spaghettis » ;
- l'utilisation des fonctions `setjmp()`, `longjmp()`, `_setjmp()`, `_longjmp()`, `sigsetjmp()`, `siglongjmp()` et `__sigsetjmp()`, provenant de la librairie standard ***setjmp.h*** qui permettent d'éviter la séquence normale d'appel et de retour de fonction pour simuler le comportement des exceptions en C, n'est pas tolérée ;
- les appels de fonctions via des pointeurs de fonctions ne sont pas gérés ;
- la manipulation de fonctions à nombre variable d'arguments différentes des fonctions `scanf()` et `printf()` n'est pas autorisée ;
- l'insertion d'instructions assembleur (via les mots clé `asm` ou `__asm__` dans le programme est interdite, de même un programme ne peut pas contenir des fichiers directement écrits en assembleur ;
- la manipulation d'attributs introduits par le mot clé `__attribute__` n'est pas supportée ;
- les références à des types de données en utilisant les mots clés `typeof` ou `__typeof__` sont interdites ;
- aucune extension GNU, quelle qu'elle soit, n'est admise.

Nous envisageons d'autoriser par la suite certaines des restrictions que nous avons citées, comme l'utilisation et la définition de fonctions à nombre variable d'arguments ainsi que la manipulation de pointeurs de fonctions.

1.2 Analyse lexico-syntaxique des programmes

Afin de ne pas « réinventer la roue », nous avons décidé de nous appuyer sur un compilateur existant pour implémenter notre analyse de programme plutôt que de devoir écrire un analyseur lexico-syntaxique [7]. Nous avons naturellement choisi le compilateur ***GCC***[7][8].

Description de GCC

GCC présente de nombreux avantages dont les principaux sont les suivants :

- il est sous licence GPL (disponible gratuitement et librement modifiable) ;
- c'est un compilateur C, C++ (***G++***), Java (***GCJ***), Objective C (appelé aussi C#), Fortran (***G77***) et Ada (***GNAT***) entre autres ;
- son compilateur C respecte la norme C99 (même s'il y ajoute de nombreuses extensions GNU) ;
- il est très portable ;
- il est possible d'en faire un compilateur croisé pour de nombreuses architectures ;
- quasiment tous les programmes sont à sa portée (il est l'un des rares compilateurs à pouvoir compiler un noyau Linux).

Le principal inconvénient de GCC est le manque de documentation technique concernant son implémentation. La plupart des pages de documentation disponibles ne sont pas à jour, il faut donc se plonger dans son code pour comprendre son fonctionnement.

Les compilateurs qui composent GCC fonctionnent tous de manière quasiment identique hormis GCJ (le compilateur Java), du fait des particularités du langage Java (il peut tout de même produire des fichiers objets et du code exécutable sans machine virtuelle Java, mais uniquement à partir de ***bytecode***[8]).

Déroulement d'une compilation

Après l'appel au préprocesseur (***CPP***), le compilateur lance une analyse lexicale du code source qui produit un arbre de syntaxe abstraite (AST : Abstract Syntax Tree) dont la représentation est particulière au ***front-end***[9] utilisé. L'AST est alors mis sous la forme dite ***generic*** ; cette représentation permet d'harmoniser les ASTs produits par les différents front-ends de GCC et de n'utiliser qu'un seul back-end pour achever la compilation, quel que soit le langage. Ensuite l'AST est analysé syntaxiquement et certaines fonctions peuvent être « inlinées ». Après cela, l'AST est transcrit dans la représentation appelée ***gimple*** ; celle-ci correspond à la représentation ***generic*** en utilisant un code à trois adresses (par exemple l'expression : `a = b + c - d ;` est transformée en : `t1 = b + c ; a = t1 - d ;`). L'AST subit alors des optimisations

[7] GCC : GNU Compiler Collection.

[8] Bytecode : code binaire portable obtenu en compilant un fichier source Java pouvant être chargé et exécuté par une machine virtuelle Java.

[9] Un front-end est la partie du compilateur qui dépend d'un langage de programmation.

selon les options de compilation. Certaines optimisations utilisent un graphe de contrôle de flot ou d'autres représentations qui leurs sont propres comme ***SSA***[10], utilisée par exemple pour effectuer la propagation des constantes. Ensuite l'AST est mis sous forme ***RTL***[11] sur laquelle d'autres optimisations peuvent s'effectuer. A la fin des optimisations, le compilateur transforme le RTL en du code assembleur puis en fichiers binaires, dits fichiers objets. Ces fichiers sont transmis à l'éditeur de liens (***LD***) qui produit enfin un fichier exécutable.

1.3 Stockage et manipulation des résultats

En ce qui concerne le stockage des résultats intermédiaires de chaque étape des modules de notre projet, nous nous sommes orientés vers le métalangage XML[12][9]. Cela se justifie par le fait que ce langage est :

- extensible à volonté (comme son nom l'indique), en définissant de nouvelles balises par le biais de DTDs[13] ou de schémas XML[10] ;
- utilisable sans difficulté sur internet, il peut même servir à communiquer les données via une sur-couche applicative comme ***SOAP***[14][12] ou ***XML-RPC***[13] utilisée conjointement à un protocole TCP/IP tel HTTP ou SMTP par exemple ;
- très concis, il ne contient que des données brutes et des balises (un document XML est relativement clair et accessoirement lisible par l'homme).

De plus, il existe de nombreux outils permettant de manipuler des documents XML à partir d'autres langages de programmation. Nous utilisons ***libxml2***[11], qui est l'une des meilleures API[15] XML tant de par ses performances que du point de vue du nombre de standards XML (analyseur SAX[16] et DOM[17], schémas XML, création et validation d'arbres XML, signature, cryptage, ...) qu'elle implémente, pour exploiter les fichiers XML contenant nos divers résultats. Ainsi nous avons réalisé des analyseurs XML pour réutiliser nos résultats intermédiaires. Dans un soucis d'efficacité, nos analyseurs utilisent l'interface SAX de la libxml2 (elle est conforme à la norme SAX2 du consortium World Wide Web — W3C). Ils sont orientés évènements (ces derniers sont : ouverture de balise, fermeture de balise, contenu textuel...) et requièrent moins de mémoire que des analyseurs utilisant la norme DOM qui sont plus simples à mettre en oeuvre mais dont les performances sont moins bonnes.

[10] SSA : Static Single Assignment.
[11] RTL : Register Transfer Language.
[12] XML : eXtensible Markup Langage.
[13] DTD : Document Type Definition.
[14] SOAP : Simple Object Access Protocol.
[15] API : Application Programming Interface.
[16] SAX : Simple API for XML.
[17] DOM : Document Object Model.

2 Implémentation partielle de l'analyse

2.1 Embedded2grid

Nous avons conçu un processus, que nous avons appelé ***embedded2grid***, pour garder une vue d'ensemble du programme et des librairies ou packages dont il dépend. Cet outil prend en entrée un fichier de configuration (cf annexe D) qu'il génère lorsqu'il est appelé avec l'option *-g* et qui doit être complété par l'utilisateur. Un fichier de configuration correspond à une analyse. Par conséquent chaque librairie ou package, dont dépend un programme à transformer et dont le code source est disponible, doit être décrit par un fichier de configuration pour être analysé. Les résultats de ces analyses intermédiaires (i.e. les noms des fichiers XML décrivant leur graphe d'appels de fonctions et la vivacité de leurs données, ainsi que les chemins absolus jusqu'à ces fichiers) doivent être spécifiés dans le fichier de configuration du programme à transformer.

Lorsqu'embedded2grid est lancé avec un fichier de configuration valide, il détermine quelles unités de compilation doivent être analysées. Pour ce faire, il procède comme un ***makefile*** : il compare leurs dates de dernière modification avec celles des éventuels fichiers XML de résultats (ceux-ci n'existent que si le projet a déjà été analysé, ils sont conservés pour permettre une *analyse séparée*[18]). Ensuite, il initie l'analyse des unités de compilation qui le nécessitent en créant un processus fils qui appelle ***GCC*** pour chacune d'elles avec les options adéquates (analyse sans compilation réelle, aucun fichier objet n'est produit). Quand tous les fichiers source du projet ont été analysés, il récupère les résultats obtenus et les exploite pour terminer l'analyse de vivacité en suivant le graphe d'appels de fonctions du projet complet.

Remarque 8 *L'utilisation de GCC pour implémenter notre analyse de programme interdit la délégation de traitements issus de programmes dont la compilation est effectuée par des méta-compilateurs comme* ***MOC***[19] *ou qui sont implémentés dans un langage de programmation étendu par une librairie ou un package dont ils dépendent (par exemple, les programmes utilisant l'API QT[14] sont codés en C++ mais QT ajoute plusieurs mots clés au langage comme* `slot` *ou* `signal`*).*

Remarque 9 *Embedded2grid ne peut pas lancer plusieurs analyses de fichiers source du même projet en parallèle. La parallélisation de l'analyse nécessiterait que les processus fils, qui l'exécutent via GCC, communiquent entre eux et utilisent des verrous sur les fichiers de résultats accédés lors de l'analyse de chaque unité de compilation.*

Embedded2grid transmet certaines informations à GCC par l'intermédiaire des variables d'environnements qu'il crée pour l'analyse :

- `E2G_FIRST` : elle est uniquement créée lors de l'analyse de la première unité de compilation d'un projet qui n'a pas encore été analysé ;

[18] Nous entendons par analyse séparée, une analyse des fichiers source d'un projet effectuée par étapes, cela peut arriver quand on modifie le code d'un projet alors qu'il a déjà été analysé

[19] MOC : Meta Object Compiler — Ce méta-compilateur provient de l'API QT[14], il est utilisé lorsqu'on implémente de nouveaux widgets pour générer leur code source à partir d'une description sommaire dans un header.

– `E2G_OUTPUT` : elle donne le chemin absolu vers le répertoire où les principaux résultats du projet sont stockés ;
– `E2G_NAME` : elle contient la concaténation du nom et de la version du projet (elle sert de base pour le nom des fichiers dans lesquels sont enregistrés les principaux résultats du projet comme la table des variables globales et la description des types de données utilisés).

Si elles existent déjà lors du lancement d'embedded2grid, cela signifie qu'elles sont potentiellement utilisées par une autre application (mais ceci ne pose aucun problème, sauf, peut-être, si cette application est exécutée en même temps qu'embedded2grid) alors l'utilisateur en est informé mais l'exécution se poursuit.

Quand la vivacité des données du projet est calculée, embedded2grid dispose de toutes les informations requises pour découper le projet considéré et élaborer les prédicats de délégation des morceaux délégables. Cependant ni le partitionnement de programme ni la conception des prédicats de délégation n'ont été implémentés car les heuristiques à mettre en oeuvre ne sont pas encore élaborées. Nous avons toutefois spécifié la représentation des descriptions de morceaux dans la DTD ***parts_definition.dtd*** (cf annexe C).

Une fois le partitionnement effectué, embedded2grid peut concevoir les ensembles de données à communiquer puis déterminer le positionnement des points de rendez-vous. L'analyse du projet est alors achevée, celui-ci peut donc être transformé et compilé.

2.2 Analyse d'une unité de compilation

Nous avons convenu d'implémenter une nouvelle passe de compilation dans GCC pour réaliser le début de notre analyse de programme. Elle est controlée par la nouvelle option `-fcompute-static-data-liveness` et requiert la définition des variables d'environnement `E2G_OUTPUT`, `E2G_NAME` et `E2G_FIRST` (cette dernière n'étant définie que si le fichier en cours d'analyse est le premier fichier source d'un projet à être analysé) qui est gérée par embedded2grid. Notre passe de compilation s'effectue fonction par fonction comme GCC l'impose puisque son principe est de ne garder en mémoire que l'AST de la fonction en cours de compilation.

Comme les fichiers source analysés doivent encore subir des transformations, nous stoppons la compilation en utilisant conjointement à `-fcompute-static-data-liveness` l'option `-fsyntaxe-only`, qui commande uniquement une vérification syntaxique de l'unité de compilation et termine l'exécution du compilateur sans produire de fichiers objets.

Remarque 10 *Nous avons également créé l'option* `-fno-compute-static-data-liveness` *afin de respecter le formalisme de GCC. Cette option est appliquée par défaut et n'entraine l'exécution d'aucune passe de compilation supplémentaire.*

L'idéal aurait été que notre passe de compilation s'exécute sur un AST représenté sous forme ***generic*** pour pouvoir s'appliquer à tous les langages pour lesquels GCC implémente un front-end et pour bénéfécier de toutes les optimisations proposées par GCC. Cependant les ASTs produits par le front-end C sont directement transformés (sans passer par la représentation dite

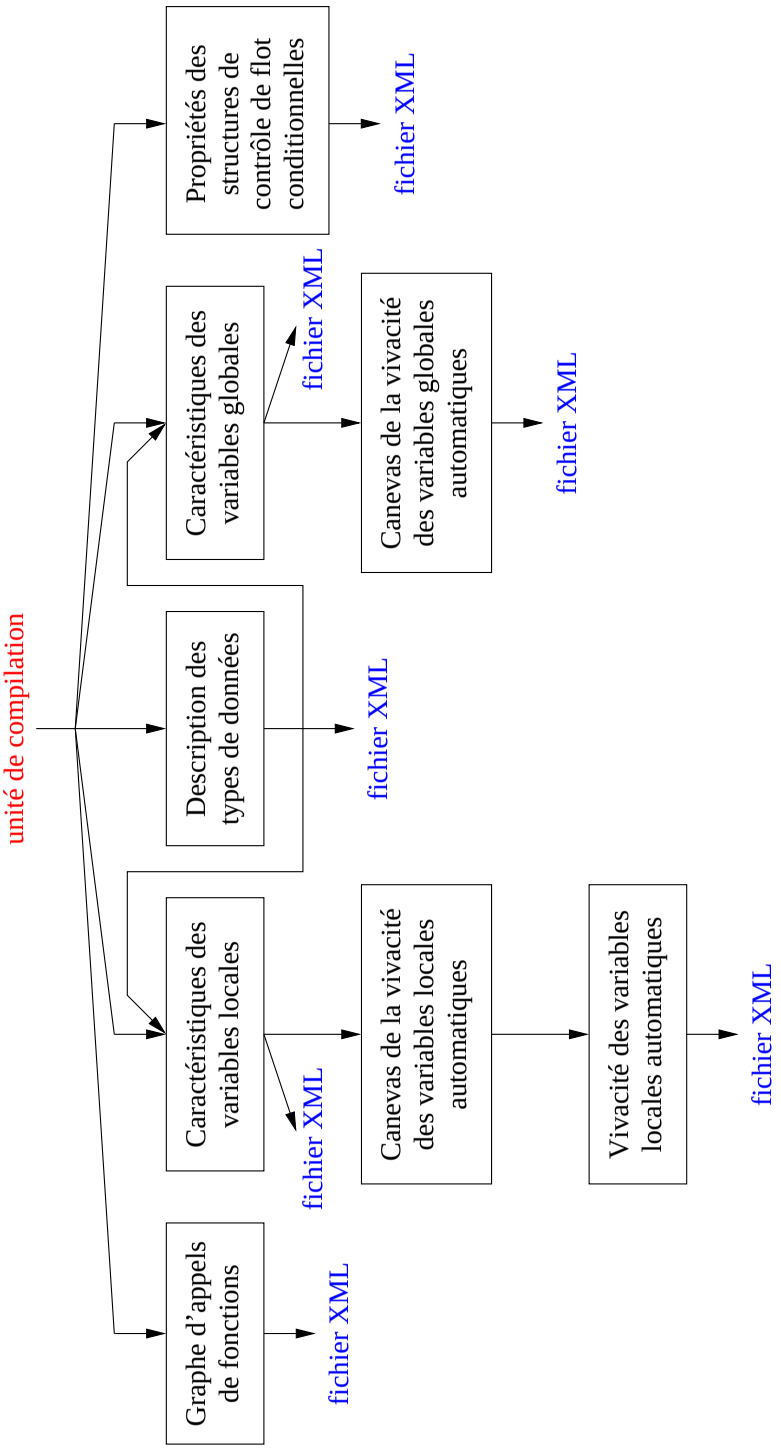

FIG. 14 – Opérations réalisées simultanément par notre passe de compilation.

generic) en ASTs ***gimple***. Comme cette représentation est basée sur un code à trois adresses, elle introduit des variables temporaires internes au compilateur et de nombreuses instructions sont fractionnées en plusieurs intructions. De ce fait, notre système d'adressage d'instructions, qui a été conçu pour correspondre le plus possible au code source des programmes, aurait été plus ou moins faussé selon la complexité des instructions d'origine et il aurait fallut différencier les variables du programme d'origine de celles générées par le compilateur pour ne pas fausser l'analyse de vivacité et la composition des ensembles de données à communiquer. Par conséquent, notre passe de compilation s'exécute sur des ASTs particuliers au front-end C, ce qui interdit quasiment toutes les optimisations proposées par GCC. En effet, si les ASTs sont optimisés après avoir été analysés, ils sont potentiellement modifiés et les résultats de l'analyse risquent de ne plus être valables. Pour utiliser les optimisations de GCC qui s'effectuent après notre passe de compilation sans pour autant annihiler ses résultats, il faudrait que les optimisations tiennent compte du partitionnement du programme et de la localisation des points de rendez-vous.

Comme le montre l'organigramme de la figure 14, notre nouvelle passe de compilation de GCC réalise simultanément :

- l'analyse des propriétés des structures de contrôle de flot conditionnelles ;
- la construction du graphe d'appels de fonctions ;
- l'étude des types de données ;
- la constitution des tables des variables locales et globales ;
- la génération d'un canevas de l'analyse de la vivacité des données globales automatiques et l'analyse de la vivacité des données locales automatiques.

Nous avons regroupé ces cinq étapes de l'analyse en une seule passe de compilation afin de ne pas parcourir à outrance les ASTs des fichiers source du projet. Ces étapes sont détaillées dans les sections suivantes.

2.2.1 Analyse des propriétés des structures de contrôle de flot conditionnelles

Dans le langage C, les structures de contrôle de flot conditionnelles sont de deux types :

- celles qui testent la nullité de la valeur de l'expression du test conditionnel, elles respectent l'une des trois syntaxes suivantes :
  - `if (cond) instr1` si la valeur de `cond` est différente de zéro alors `instr1` est exécuté ;
  - `if (cond) instr1 else instr2` si la valeur de `cond` est différente de zéro alors c'est `instr1` qui est exécuté sinon c'est `instr2` ;
  - `cond ? expr1 : expr2` cette expression prend la valeur de l'expression `expr1` si la valeur de `cond` est non nulle, sinon elle prend la valeur de l'expression `expr2`.

  Où :
  `cond` est une expression de type arithmétique ou pointeur servant de test conditionnel ;
  `instr1 instr2` sont des instructions ou des blocs d'instructions ;
  `expr1 expr2` sont des expressions quelconques.
- celles qui gèrent plusieurs alternatives pour la valeur de l'expression du test conditionnel, elles suivent la syntaxe suivante :

```
switch (cond)
  {
     case expr_cst_1 : instr_1
     case expr_cst_2 : instr_2
             ...
     case expr_cst_N : instr_N
     [ default : instr ]
  }
```

`cond` est évaluée et fournit une valeur `V`. Si `V` est une des valeurs `expr_cst_1` ou `expr_cst_2` ou ... ou `expr_cst_N` alors l'exécution se poursuit par les instructions du `case` correspondant et ce jusqu'à la prochaine instruction `break` ou la dernière instruction de `instr`. Si `V` n'est pas trouvé et que `default` est présent, alors `instr` est exécuté.
Où :
`cond` est une expression de type entier ou caractère servant de test conditionnel ;
`expr_cst_1 ...expr_cst_n` sont des expressions constantes de type entier ou caractère ;
`instr_1 ...instr_N et instr` sont des instructions (peut-être l'instruction vide) ou des blocs d'instructions dont la dernière instruction peut être le mot clé `break`.

Dès que nous rencontrons une structure de contrôle de flot conditionnelle en parcourant l'AST d'une fonction, nous enregistrons l'adresse à laquelle elle est située et le nombre de fils qu'elle engendre. La représentation de ces propriétés est spécifiée par la DTD ***conditional_branches.dtd*** (cf annexe E).

Ces propriétés permettront de déterminer la symétrie des structures de contrôle conditionnelles du programme par rapport aux productions et les ensembles de données à communiquer entre les différents morceaux de programme. Elles sont enregistrées dans des fichiers XML (d'extension `cond_stmts.xml`) — un fichier XML par unité de compilation — qui seront concaténés à la fin de l'analyse par embedded2grid.

2.2.2 Graphe d'appels de fonctions

Le graphe d'appels de fonctions est essentiel pour déterminer la vivacité des données avec exactitude. Il est également utilisé pour identifier les fonctions qui ne sont appelées qu'une seule fois ; ainsi, dans ces fonctions, le positionnement des points de rendez-vous pourra être affiné.

En parcourant l'AST de la fonction en cours d'analyse, nous relevons tous les appels de fonctions ainsi que l'adresse à laquelle ils ont lieu et les propriétés (son nom et un marqueur pour savoir s'il s'agit d'une fonction ***built-in***[20]) de la fonction appelée. A la fin de l'analyse de l'unité de compilation, nous créons un fichier XML (dont l'extension est `call_graph.xml`) dans lequel nous enregistrons la description des appels de fonctions en respectant notre DTD ***call_graph.dtd*** (cf annexe F).

[20] Les fonctions built-in sont celles qui appartiennent au standard du langage et qui sont implémentées par le compilateur.

2.2.3 Étude des types de données

Les caractéristiques des types de données du projet sont nécessaires pour estimer la charge des communications de données lors du partitionnement du programme et pour effectuer les communications lors de l'exécution. Nous avons spécifié leur représentation dans la DTD ***types_definition.dtd*** (cf annexe G).

A chaque nouvelle déclaration de variable rencontrée, nous décodons les caractéristiques de son type de données. Si ce n'est pas un type de base, alors, s'il s'agit d'un type aggrégé nous le décomposons pour étudier le type de tous les champs, sinon, s'il s'agit d'un pointeur nous nous intéressons au type pointé, sinon s'il s'agit d'un *tableau statique*[21] nous considérons le type et le nombre des éléments, sinon, s'il s'agit d'une énumération nous relevons le nom et la valeur des constantes. Lorsque nous avons toutes les caractéristiques d'un type de données et de ses éventuels composants, nous déterminons si nous les avons déjà enregistrés (i.e. si nous l'avons déjà rencontré dans une précédente unité de compilation du même programme). Si ce n'est pas le cas, nous assignons un identifiant unique à chaque nouveau type de données — cela nous permettra de les référencer simplement, sans répéter toutes leurs caractéristiques, dans nos tables de variables — puis nous les enregistrons.

Remarque 11 *Dans une même unité de compilation, GCC construit un sous-arbre correspondant à chaque type de données utilisé. Nous utilisons donc l'adresse de ce sous-arbre afin d'identifier un type de données dans une unité de compilation.*

Exemple 6

```
/* Fichier "point.c": */
...
struct Point
{
   double x, y;
   unsigned long int col;
};
...
static struct Point p = {0., 0., 0};  /* Variable globale au fichier "point.c". */
...
```

Les descriptions des types de données correspondants à la déclaration de la variable `p` sont les suivantes :

```
<data_type key="0" memory="160" const="no" volatile="no">
  <struct name="Point">
    <field type="1" name="x"/>
    <field type="1" name="y"/>
    <field type="2" name="col"/>
  </struct>
</data_type>
```

[21] Nous appelons tableau statique un tableau dont le nombre d'éléments est connu à la compilation.

```
<data_type key="1" memory="64" const="no" volatile="no">
  <simple name="double" signed="yes" size="normal"/>
</data_type>
<data_type key="2" memory="32" const="no" volatile="no">
  <simple name="int" signed="no" size="long"/>
</data_type>
```

Nous sauvegardons la description de tous les types de données du programme dans un fichier XML qui est mis à jour par GCC au fur et à mesure de l'analyse. Ce fichier est lu avant de commencer l'analyse d'une unité de compilation lorsque ce n'est pas la première du projet à être analysée (i.e. lorsque la variable d'environnement `E2G_FIRST` n'a pas été définie par embedded2grid). De plus, afin d'être capables d'effectuer une analyse séparée des fichiers sources d'un programme, nous créons un fichier XML (d'extension `data_types.xml`) par unité de compilation analysée pour y stocker la description des types de données utilisés dans cette unité.

2.2.4 Conception des tables de variables

Lors de l'analyse de vivacité des données, nous allons souvent référencer les variables du projet et nous avons besoin de connaître leur portée et leur classe de stockage car la vivacité des variables déclarées ***static*** est particulière. Nous attribuons donc aux variables un identifiant unique et nous enregistrons leurs propriétés dans des tables en fonction de leur portée.

Pour chaque unité de compilation, nous créons une table contenant les caractéristiques des variables locales et une autre pour celles des variables globales. De plus, nous rassemblons dans une table les descriptions des variables globales de tous les fichiers source du projet qui ont déjà été analysés. Cette table est chargée en mémoire avant de commencer l'analyse d'un fichier source du projet lorsque d'autres ont déjà été analysés.

Ces tables sont mises à jour au fur et à mesure que des déclarations de variables sont rencontrées en traversant les ASTs des fonctions du fichier source courant. Lorsqu'une variable est de type composé (structure ou union), nous enregistrons, dans les tables adéquates, la description de chaque champ afin d'affiner l'analyse de la vivacité des données (chaque champ d'une donnée composée a le même identifiant que la donnée considérée et la même classe de stockage).

Nous avons défini la représentation des variables locales dans la DTD ***local_variables.dtd*** (cf annexe H) et celle des variables globales dans la DTD ***global_variables.dtd*** (cf annexe I). Les tables des variables locales et globales sont enregistrées dans des fichiers XML, dont les extensions sont respectivement `local_vars.xml` et `global_vars.xml`, en se conformant aux DTDs précédemment citées.

Exemple 7 Voici les enregistrements correspondants à la déclaration de variable globale de l'exemple précédent :

```
<file name="point.c">
  <variable name="p" key="0" type="0" qual="static" register="no" init="yes"/>
  <variable name="p.x" key="0" type="1" qual="static" register="no" init="yes"/>
  <variable name="p.y" key="0" type="1" qual="static" register="no" init="yes"/>
```

```
  <variable name="p.col" key="0" type="2" qual="static" register="no" init="yes"/>
  ...
</file>
```

2.2.5 Analyse de la vivacité des données automatiques

Les données d'un projet se répartissent en deux sortes :
- les données automatiques ou statiques : leur occupation mémoire est connue à la compilation, elles sont instanciées dans la pile du programme ;
- les données dynamiques : ce sont les pointeurs ; les zones mémoires pointées, dont la taille n'est pas connue à la compilation, seront allouées dans le tas du programme au cours de son exécution.

Nous ne nous sommes intéressés qu'à la vivacité des données automatiques pour le moment. L'analyse de la vivacité des données dynamiques, qui nécessite une analyse des ***alias***[22], est beaucoup plus complexe à réaliser. Nous procédons en deux étapes pour constituer un canevas de la vivacité des données automatiques dans une fonction.

Nous commençons par construire, pour chaque variable rencontrée, un arbre décrivant les structures de contrôle de flot de la fonction. Nous stockons temporairement dans ces arbres toutes les productions et consommations des variables correspondantes. Nous y enregistrons également les instructions de saut (***break***, ***continue*** et ***return***), qu'elles soient gardées[23] par des structures de contrôle de flot conditionnelles ou non, cela nous permet d'éliminer le code qui n'est jamais exécuté et de rassembler avec exactitude les productions et les consommations des variables locales même dans les boucles et les nids de boucles.

Lorsque l'AST de la fonction a été totalement traversé, nous passons à la deuxième étape qui consiste à réunir les productions et les consommations correspondantes dans des pieuvres, quand c'est possible. Comme nous analysons les fonctions dans l'ordre dans lequel elles sont définies dans les fichiers source sans tenir compte du graphe d'appels de fonctions, nous ne pouvons pas encore, à de stade, associer les productions et les consommations des variables globales. Le canevas de l'analyse de la vivacité des données globales se résume donc aux arbres précédemment construits. Nous enregistrons dans un fichier XML (dont l'extension est `global_live.xml`) toutes les productions et les consommations contenues dans les arbres des variables globales en respectant la DTD ***tmp_global_static_data_liveness.dtd*** que nous avons établie (cf annexe J). Quant au canevas de la vivacité des données locales, nous l'établissons en associant les productions et les consommations adéquates dans des pieuvres, que nous regroupons par pools lorsqu'elles partagent au moins une consommation. Ensuite nous enregistrons ce canevas de la vivacité des données locales automatiques dans un fichier XML (dont l'extension est `local_live.xml`) en se conformant à la DTD ***local_static_data_liveness.dtd*** (cf annexe K).

Remarque 12 *Tels que nous construisons les arbres dans lesquels nous stockons les productions et les consommations des variables, la première « manipulation » de variable incluse dans*

[22] Deux variables sont des alias si elles désignent la même zone en mémoire ; dans l'exemple suivant, (*p) et i sont des alias : int i, *p = &i ;

[23] Nous disons d'une instruction qu'elle est gardée si son exécution dépend d'un test conditionnel.

les arbres concernant les données locales est une production de variable. Cette propriété n'est pas vérifiée pour les arbres décrivant une partie de la vivacité des données globales ; c'est pourquoi ces arbres constituent le canevas de la vivacité des données globales.

Exemple 8 Le petit programme suivant calcule la factorielle des entiers saisis par l'utilisateur tant que celui-ci veut poursuivre les calculs.

```
1   #include <stdio.h>

    int main (void)
    {
5     char s[64];
      int res;
      do
        {
          int n, i = 1;
10        res = 1;
          do
            {
              printf ("\nNombre dont il faut calculer la factorielle?\n\t");
              scanf ("%d", &n);
15            if (n < 0)
                printf ("La factorielle n'est définie que pour des entiers positifs!\n");
            }
          while (n < 0);
          while (i <= n)
20          {
              res *= i;
              ++i;
            }
          printf ("La factorielle de %d vaut %d.\n", n, res);
25        printf ("Un nouveau calcul? ('o' ou 'O' pour continuer)\n\t");
          scanf ("%s", &s);
        }
      while (!strcmp (s, "o") || !strcmp (s, "O"));
      return res;
30  }
```

Les figures 15, 16, 17 et 18 présentent les arbres qui décrivent les manipulation des variables du programme, tels qu'ils sont construits lors de notre passe de compilation, avant d'affiner les canevas de la vivacité des variables locales. La légende de ces figures est précisée sur la figure 19.

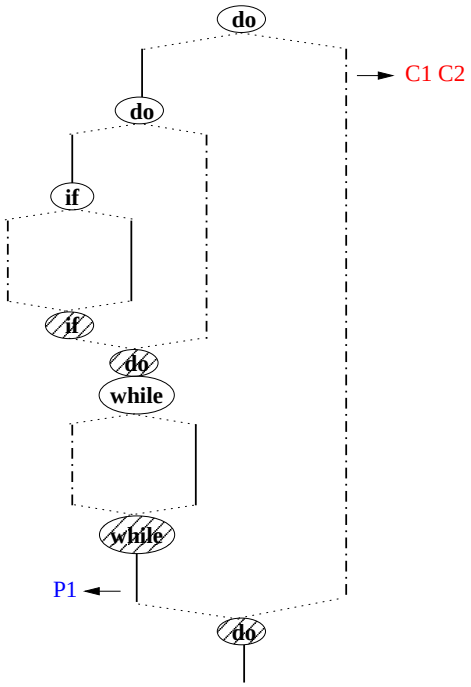

FIG. 15 – Arbre relatif à la variable `s`.

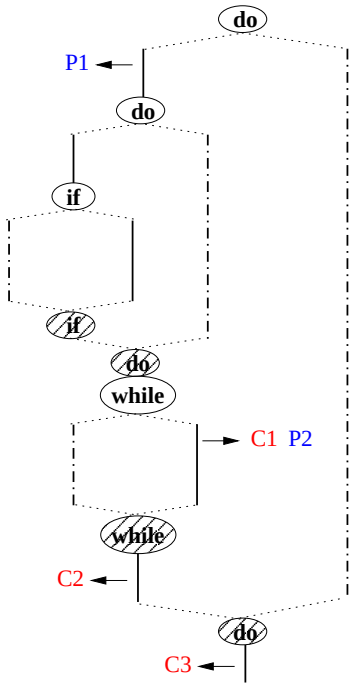

FIG. 16 – Arbre relatif à la variable `res`.

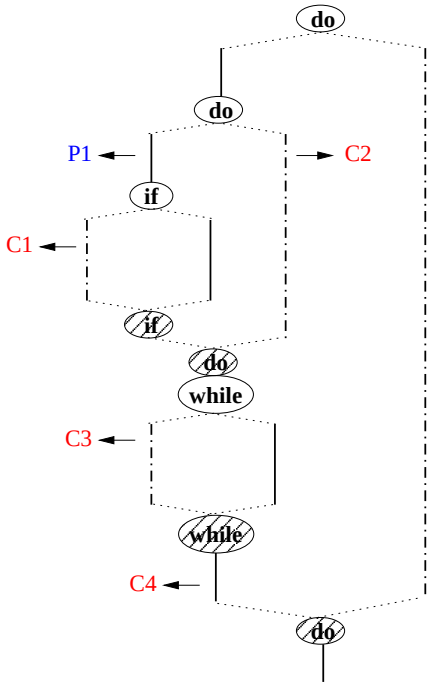

FIG. 17 – Arbre relatif à la variable `n`.

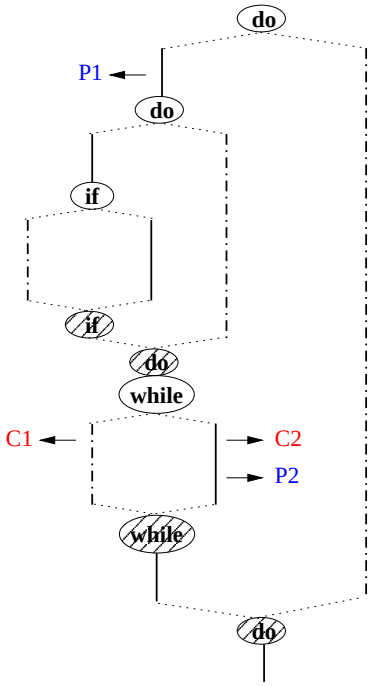

FIG. 18 – Arbre relatif à la variable `i`.

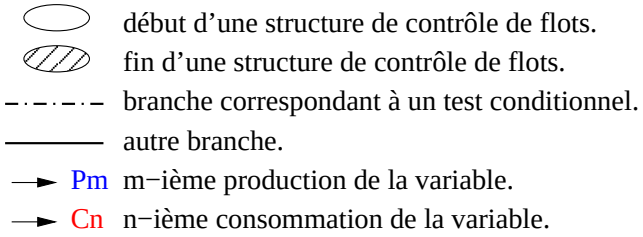

FIG. 19 – Légende des figures 15, 16, 17 et 18.

2.3 Poursuite de l'analyse par embedded2grid

Les résultats obtenus après l'exécution de notre nouvelle passe de compilation de GCC sur tous les fichiers source du projet ne sont que partiels. Embedded2grid doit les rassembler et achever l'analyse du projet comme cela est présenté sur le schéma de la figure 20.

Le graphe d'appels de fonctions du projet est réalisé en concaténant les fichiers XML décrivant le graphe d'appels de fonctions de chaque unité de compilation du projet et ceux des librairies ou packages dont le projet dépend. Ensuite embedded2grid termine l'analyse du projet en trois étapes :

- il finalise l'analyse de vivacité des données automatiques ;
- il détermine la symétrie des structures de contrôle de flot conditionnelles du projet ;
- il constitue les ensembles de données à communiquer, puis il localise les points de rendez-vous correspondants.

2.3.1 Finalisation de l'analyse de vivacité des données automatiques

La finalisation l'analyse de vivacité des données locales automatiques est triviale. Il suffit à embedded2grid de concaténer les fichiers XML qui décrivent la vivacité des données locales automatiques de chaque unité de compilation du projet.

Remarque 13 *Lorsqu'il s'agira de déterminer la vivacité des données locales dynamiques du programme, il faudra vérifier que les appels de fonctions qui ont lieu entre la production et une des consommations correspondantes (de la même* pieuvre*) ne produisent pas les données passées en arguments.*

En ce qui concerne la vivacité des données globales automatiques du projet, elle est calculée à partir des canevas conçus lors de l'exécution de notre passe de compilation sur les fichiers source du projet. Comme toutes les informations nécessaires sont disponibles (le graphe complet des appels de fonctions et le canevas de la vivacité des données globales automatiques), elle est déterminée en suivant le graphe d'appels de fonction à partir de la fonction `main`.

2.3.2 Calcul de la symétrie des structures de contrôle de flot conditionnelles

L'analyse de la vivacité des données du projet et les propriétés des structures de contrôle de flot conditionnelles permettent à embedded2grid de déterminer la symétrie de ces dernières par rapport à leurs productions. Pour ce faire il compare les productions de données qui ont lieu dans chaque branche d'une même structure de contrôle de flot conditionnelle.

Remarque 14 *Nous avons décidé de manipuler les ensembles de données dits exacts, c'est pourquoi nous avons besoin de connaître la symétrie des structures de contrôle de flot conditionnelles du programme par rapport à leurs productions. De plus l'instrumentation du programme devra tracer l'exécution des branches optionnelles qui ne sont pas issues de structures de contrôle de flot conditionnelles symétriques par rapport à leurs productions.*

2.3.3 Production des ensembles de données et localisation des points de rendez-vous

Embedded2grid constitue les ensembles de données à comuniquer selon l'un des deux algorithmes présentés dans la première partie en construisant un graphe. Dans l'optique d'extraire du programme d'origine le plus de parallélisme possible, nous nous sommes orientés vers l'algorithme qui permet de déterminer les ensembles de données exacts afin de n'envoyer que les données effectivement produites et de ne pas générer de fausses dépendances entre les tâches. Au fur et à mesure que les ensembles de données du projet sont établis, embedded2grid calcule la localisation des points de rendez-vous correspondants. Ceux-ci se répartissent de la manière suivante : à un ensemble de données, correspondent un point de rendez-vous actif et un ou plusieurs points de rendez-vous passifs (pouvant être situés dans des tâches différentes).

Lorsque les ensembles de données du projet sont produits et que la localisation des points de rendez-vous correspondants est déterminée, nous stockons ces informations dans un fichier XML en se conformant à la DTD ***datasets_appointements.dtd*** (cf annexe L) que nous avons conçue.

Remarque 15 *Si l'utilisation des ensembles de données exacts s'avérait trop lourde pour la machine locale du fait des traçages que cela implique (ce qui pourrait survenir si la machine locale est un système embarqué), il est envisageable que le choix du type des ensembles de données (exacts ou approximatifs) soit laissé à l'utilisateur via un nouveau paramètre dans le fichier de configuration de embedded2grid.*

FIG. 20 – Finalisation de l'analyse par embedded2grid.

3 Communication de données et appels distants

Dans cette section, nous exposons quelques idées et suggestions de techniques à utiliser pour implémenter la délégation de tâches et la communication de données entre celles-ci.

3.1 Systèmes distribués

Étant données les spécificités de l'application que nous développons, nous avons exclu l'usage de systèmes distribués comme CORBA[24] et Java/RMI[25] (cf respectivement [24] et [25] pour de plus amples informations). La lourdeur de ces systèmes pourrait poser des problèmes pour certaines architectures et notamment pour les systèmes embarqués (il existe des implémentations de ces systèmes distribués destinées à des systèmes embarqués mais, en général, elles ne sont pas sous licence GPL). De plus ces systèmes sont basés sur un langage IDL[26] pour faciliter le développement des applications. Ce langage intermédiaire compliquerait leur utilisation : nous devrions générer automatiquement du code IDL, compiler celui-ci puis exploiter les squelettes de code cible ainsi obtenus. Toutefois, de nombreuses fonctionnalités de CORBA pourraient s'avérer intéressantes pour notre projet, par exemple :

- la découverte de services ;
- l'installation de nouveau services sans avoir besoin de recompiler le code des serveurs (cette fonctionnalité est spécifiée dans la norme CORBA3 mais pas encore réalisée) ;
- l'utilisation d'un *entrepôt* pour stocker la description des services et objets distribués ;
- la transparence des communications de données.

3.2 Outils de programmation parallèle

Nous nous sommes tout d'abord intéressé aux librairies de fonctions de la famille MPI[27] (comme MPICH, cf [15]), basées sur la transmission de messages. Nous pensions les utiliser pour communiquer nos données et effectuer nos appels distants comme dans de nombreuses applications destinées à des grilles d'ordinateurs. Cela s'est avéré impossible car dans tout programme MPI, les nombres de processus à générer et de processeurs exploitables doivent être connus au lancement du programme, ce qui n'est pas notre cas puisque nous réalisons un placement dynamique des morceaux délégués sur des ordinateurs distants dont le nombre peut varier au cours du temps.

Ensuite, nous avons étudié le fonctionnement de PVM[28] (cf [16] pour plus de détails). C'est une bibliothèque de communication et un environnement de programmation pour machines parallèles et réseaux de stations (hétérogènes ou pas) qui permet à un réseau de calculateurs d'apparaître comme une seule machine (concept de machine virtuelle). PVM est composé de deux parties : un « daemon » (processus démon), qui réside sur chacun des noeuds de l'ensemble,

[24] CORBA : Common Object Request Broker Architecture.
[25] RMI : Remote Procedure Invocation.
[26] IDL : Interface Description Language.
[27] MPI : Message Passing Interface.
[28] PVM : Parallel Virtual Machine.

et une bibliothèque de sous-programmes qui permettent l'initialisation des programmes sur les noeuds, la communication entre les processus et la modification de la configuration des machines.

L'utilisation de MPI ou de PVM risquerait de rendre notre application trop lourde pour être exploitée par n'importe quel système embarqué (ce qui serait domage, étant donné que c'était le but premier du projet) et surtout, la communication des données entre les divers machines ne serait pas aisée car ces deux systèmes s'appuient sur les RPC (cf [18] pour la spécification des RPC). L'usage des RPC implique une encapsulation/décapsulation des données pour les sérialiser/désérialiser et s'abstraire des caractéristiques d'architecture (« little endian » et « big endian ») des interlocuteurs. Il nous aurait donc fallut générer de manière automatique des fonctions de codage/décodage (à partir des fonctions XDR de base) correspondant à chaque type de données utilisé qui ne soit pas un type de base du protocole. Même si l'usage de PVM peut être simplifié en utilisant CPPvm (cf [17] pour une présentation complète), une interface C++ de PVM, qui permet de transmettre des objets simples (certaines classes définies par l'utilisateur et quelques unes de la librairie standard C++, STL[29], peuvent être transférées par des fonctions standards de l'interface CPPvm mais il y a beaucoup de restrictions), de nombreuses lacunes demeurent. Nous n'utiliserons donc pas ces technologies de programmation parallèle qui s'avèrent peu intéressantes pour réaliser la parallélisation et distribution automatique que nous désirons effectuer.

3.3 Protocoles RPC

Après avoir étudié divers outils de programmation parallèle, nous nous sommes orientés vers les protocoles RPC en commençant par les GridRPC (cf [19] pour une présentation complète). Ces RPC sont exploitées au-dessus de systèmes de distribution de calculs comme Netsolve (cf [21]) ou Ninf/Ninf-G (cf [22]) que nous voulons éviter car ils sont trop lourds pour des systèmes embarqués. Elles sont destinées aux grilles d'ordinateurs et utilisent, en interne, un langage IDL (comme CORBA[24] ou Java/RMI[25] par exemple). Nous avons décidé de ne pas les employer, car réutiliser de manière automatique les fichiers obtenus en compilant du code IDL (lui-même généré automatiquement) nous semblait trop compliqué à réaliser par rapport aux avantages que nous pouvions en tirer, sans parler des inconvénients propres aux RPCs en règle générale (génération automatique des fonctions de sérialisation/désérialisation des données).

Par la suite, nous nous sommes documentés à propos des XML-RPCs (cf [20] pour consulter la spécification et d'autres informations). Il s'agit d'une norme issue des travaux de Dave Winer. C'est un moyen d'accéder à des services web en décrivant les échanges sous un format XML et en utilisant HTTP comme protocole de transfert. La simplicité de XML-RPC est son plus grand avantage (sa spécification avec les exemples et la FAQ tiennent en sept pages !), c'est un protocole extrêmement facile à comprendre et à mettre en application. Ses principaux inconvénients sont les suivants :

– tout type de données composé (structure, tableau hétérogène) ou définis par l'utilisateur

[29]STL :Standard Templates Library.

est forcément anonyme,
- il est impossible de spécifier à quelle partie de l'application réceptrice sont destinées les données envoyées, et,
- on ne peut pas passer un objet en argument à une fonction.

Après les XML-RPCs, nous nous sommes intéressés au protocole SOAP (cf [12] pour en avoir une vue complète). Ce protocole peut être considéré comme une extension de XML-RPC : il gère des échange d'informations (textuelles) ou de commandes WSDL en utilisant XML pour formater les messages et HTML (ou SMTP ou POP) pour les véhiculer. SOAP est plus approfondi que XML-RPC, et donc aussi plus compliqué à utiliser mais plus précis. Un message SOAP respecte l'encodage déterminé par un schéma XML et est généralement constitué de deux parties :
- une déclaration XML (optionnelle), et,
- une enveloppe SOAP (l'élément racine), elle-même composée :
  - d'un entête SOAP (optionnel) pour transmettre des données d'authentification ou de gestion de session, et,
  - d'un corps SOAP qui encapsule une unique balise de méthode portant le nom de la méthode appelée suivi du mot "Response" dans le cas d'un message de réponse. Les arguments ou le résultat sont inclus et minutieusement décrits dans le tag de méthode.

SOAP crée donc des messages qui se décrivent eux-mêmes, c'est à dire que chaque message SOAP transporte avec lui assez d'informations pour qu'une application le reçoive et sache comment le traiter. Les principaux avantages de SOAP sont :
- la possibilité de nommer explicitement tout type de données, y compris les plus complexes, et de personnaliser chaque détail d'un message,
- l'interopérabilité avec tous les sytèmes d'exploitations et les langages de programmation (en effet, SOAP n'est tenu par aucun système d'exploitation, langage de programmation ou modèle objet), et,
- l'utilisation des infrastructures existantes (firewalls, routeurs, proxy...) par l'usage de HTTP.

Cependant plus le message est personnalisé, plus il nécessite de travail de la part de l'expéditeur et du destinataire pour l'empaquetage et le dépaquetage des données. Par ailleurs, l'utilisation du langage XML pour coder les données, qui pourrait être un inconvénient pour certaines applications, est un avantage pour nous. En effet, au cours de l'analyse des programmes à déporter, nous étudions en détails les données qui doivent être communiquées entre différents morceaux du programme et nous sauvegardons leurs caractéristiques sous forme de fichiers XML (la description des données à inclure dans les messages SOAP existe donc déjà).

Finalement nous avons étudié un MOM[30] qui semble être la technologie la plus appropriée pour répondre à nos besoins : xmlBlaster[23]. Ses principaux avantages sont :
- il est multiplateforme ;
- il est implémenté dans de nombreux langages (il y a des clients C/C++, Java, Python, PHP, ...) ;

[30]MOM : Message Oriented Middleware.

– les messages sont écrits en XML ;
– il supporte de multiples protocoles (CORBA, RMI, XmlRpc, SOAP, sockets réseau...) ;
– il propose des systèmes de sécurité et d'authentification ainsi qu'un plugin LDAP[31] ;
– il permet facilement la réalisation d'agents.

[31]LADP : Lightweight Directory Access Protocol — protocole de gestion d'annuaires de réseau.

Conclusion

Ce projet de délégation de traitements, que nous avons initié, nécessite encore beaucoup de développement ; son envergure serait comparable à celle du compilateur McCAT dont la réalisation a demandé près de dix ans de travail à une équipe de chercheurs canadienne. Nous avons globalement spécifié son fonctionnement mais de nombreux points restent à approfondir.

Le module *spécification du découpage* nécessite des heuristiques complexes, qui restent à déterminer, pour élaborer les prédicats de délégation des morceaux de programmes délégables et déterminer une partition judicieuse des programmes afin d'extraire le plus de parallélisme en minimisant les communications.

L'analyse de programme, que nous avons partiellement implémentée et qui correspond au module *analyse de la distribution*, est un outil essentiel du projet, elle peut être intéressante pour d'autres applications ou optimisations de code. La réutilisation de cette analyse est facilitée par le fait que les résultats intermédiaires de chacune de ses étapes sont stockés dans des fichiers XML et sont donc aisément exploitables, d'autant que nous avons déjà implémenté (en C) les analyseurs correspondants aux DTDs utilisées.

Le module *réalisation du découpage* n'a pas été réalisé. Nous l'avons spécifié dans sa globalité, cependant les détails de la transformation de code spécifique aux machines *locales* et celle destinée aux machines *distantes* restent à préciser car ces transformations automatiques dépendent fortement des technologies qui seront mises en oeuvre pour réaliser les communications de données ainsi que les lancements et la synchronisation des différents types d'exécutions.

Quant au dernier module du projet, *mise en place de la distribution*, nous avons étudié les différentes technologies qui pourraient être utilisées pour le réaliser. De plus nous avons spécifié un protocole utilisant des agents pour centraliser les informations et gérer la multilocalisation du code à exécuter et de ses données.

La poursuite du développement de ce projet pourra se baser sur les commentaires dans l'implémentation réalisée et grâce à la documentation générée à l'aide de *doxygen* [26] . De plus, si *dot*, un outil de réalisation de graphes issu de *graphviz* [27], est utilisé conjointement à *doxygen*, la documentation peut contenir les graphes d'appels de fonctions propres à chaque fonction, simplifiant la compréhension du code existant.

D'autre part, nous avons réalisé une application calculant les racines réelles d'un polynôme par la méthode de Newton-Raphson. Cette application est codée en C, elle utilise uniquement des variables automatiques dans sa partie algorithmique et dispose d'une interface graphique réalisée avec GTK2. Elle a pour vocation de servir de programme de test pour le développement de notre projet de délégation de traitementsen en déterminant manuellement la localisation des morceaux et en faisant en sorte que les morceaux contenant des données dynamiques (concernant uniquement l'interface graphique) soient locaux.

Références

[1] Université McGill de Montreal au Canada, 1990 – 1999.
`http://www-acaps.cs.mcgill.ca/info/McCAT/McCAT.html`

[2] Intel et l'académie des sciences de Chine, ORC : Open Research Compiler, dernière version : 2.1, 25 Juillet 2003. `http://ipf-orc.sourceforge.net`

[3] J. Ferber, *Les Systèmes Multi-Agents* InterEditions, Paris, 1995–1997.

[4] Benoît Meister et Gilles Bitran, Rapport de recherches, *Adressage des instructions dans notre système de découpage*, 2003.

[5] A. Dante, Y. Robert et F. Vivien, *Scheduling and Automatic Parallelization*, 2000, Bukhaüser Boston.

[6] B.W. Kernigham et D.M. Ritchie, *Le langage C*, 2^{e} édition, 1994.

[7] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers : Principles, Techniques and Tools.* Addison Wesley, 1986.

[8] GCC : GNU Compiler Collection, dernière version : 3.3.1, `http://gcc.gnu.org`.

[9] W3C, Spécification du langage XML (eXtensible Markup Language) version 1.0 deuxième édition et des DTDs (Document Type Definition), 6 Octobre 2000.
`http://www.w3.org/TR/REC-xml`

[10] W3C, Spécification des schémas XML, version 1.0, 2 Mai 2001 :
partie 0 — `http://www.w3.org/TR/xmlschema-0`
partie 1 : structures — `http://www.w3.org/TR/xmlschema-1`
partie 2 : types de données — `http://www.w3.org/TR/xmlschema-2`

[11] Daniel Veillard, *libxml2* : une librairie XML en C disposant, entre autres, des interfaces SAX2 et DOM2, dernière version : 2.5.10, 2002–2003, `http://xmlsoft.org/`.

[12] W3C, Recommendation version 1.2 de SOAP (Simple Object Access Protocol), 24 juin 2003, `http://www.w3.org/TR/2003/REC-soap12-part0-20030624/`.

[13] UserLand Software, XML-RPC, `http://www.xmlrpc.com`, 15 juin 1999.

[14] Trolltech, QT : API pour concevoir des interfaces graphiques en C++, dernière version : 3.2.1, `http://www.trolltech.com`

[15] Bill Gropp et Rusty Lusk du laboratoire national d'Argonne, et, Tony Skjellum et Nathan Doss de l'université de l'état du Mississippi, MPICH : une implémentation portable de MPI version 1.2.2, Septembre 2003.
`http://www.psc.edu/general/software/packages/mpich/mpich.html`,

[16] PVM : Parallel Virtual Machine, `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[17] Univeristé de Stuttgart, CPPvm : Une interface C++ de PVM, 1999–2002.
`http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm/`

[18] SUN Microsystems, RPC : Remote Procedure Call.
rfc 1831 avril 1988 — `http://www.faqs.org/rfcs/rfc1831.html`
rfc 1050 août 1995 — `http://www.faqs.org/rfcs/rfc1050.html`

[19] SOI Asia Project, GridRPC, juillet 2002.
http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf

[20] UserLand Software, XML-RPC, 15 juin 1999. http://www.xmlrpc.com

[21] Université du Tennessee, Netsolve version 1.4.1. http://www.icl.cs.utk.edu/netsolve

[22] Le projet Ninf, Ninf, 2001. http://ninf.apgrid.org

[23] MOM (Message-Oriented Middleware) xmlBlaster, 2000–2003.
http://www.xmlblaster.org

[24] OMG (Object Management Group), CORBA (Common Object Request Broker Architecture). http://www.corba.org.

[25] SUN Microsystems, Java RMI (Remote Method Invocation).
http://java.sun.com/products/jdk/rmi/

[26] Dimitri Van Heesh, doxygen version 1.3, 22 Avril 2003.
http://www.stack.nl/~dimitri/doxygen

[27] D. Dobkin, J. Ellson, E. Gansner, E. Koutsofios, S North, K.P. Vo et G. Woodhull, graphviz version 1.10, 2003.
http://www.research.att.com/sw/tools/graphviz

A Adressage des instructions dans notre système de découpage

Auteurs Benoît Meister et Gilles Bitran.

Ce rapport technique décrit une instanciation au langage C des normes d'adressage d'instructions dans un arbre de syntaxe abstrait utilisées dans notre outil de découpage de programmes. L'adressage décrit ici définit un ordre total entre les instructions permettant de les localiser dans le flot de contrôle du programme. Celui-ci permet de retrouver l'ordre (chronologique) d'exécution des instructions (qui est un ordre partiel).

Mots-clés Analyse de programmes, adressage d'instructions, Arbre syntaxique (AST)

A.1 Introduction

Ce rapport technique se place dans le cadre d'un outil de découpage de programmes développé dans notre laboratoire. Le programme considéré peut être écrit dans un langage impératif dont le flot de contrôle doit être déterminé par des structures de haut niveau (de type if, case, while, for) uniquement. Dans tous les cas, son analyse syntaxique [7] produit son arbre de syntaxe abstrait, qui décrit indirectement son flot de contrôle. Le découpage de programmes consiste à partitionner un programme en ensembles d'instructions successives appelés *morceaux* de programmes. Pour analyser les instructions d'un programme, il est alors important de savoir dans quel morceau celles-ci se trouvent. Il faut donc être capable de localiser une instruction, et être capable de retrouver l'ordre d'exécution d'un ensemble d'instructions d'après cette localisation. De manière générale, une analyse globale des instructions d'un programme peut nécessiter l'*adressage* de ces instructions. Cet adressage doit :

1. permettre de *localiser* facilement et lisiblement une instruction,
2. rendre compte de l'ordre d'exécution de ces instructions,
3. et prendre en compte la structuration du contrôle de flot existant dans les langages actuels.

La section A.2 explicite la norme d'adressage qui sera utilisée pour nos travaux sur le découpage de programmes, dans le cas du langage C. Ensuite, les algorithmes permettant la comparaison chronologique de deux instructions (point 2) sont décrits dans la section A.5.

A.2 Norme d'adressage des intructions

La construction d'un adressage répondant à la condition 1 est triviale : il suffit pour cela de prendre l'ordre dans lequel les instructions ont été écrites par le programmeur (leur numéro de ligne si celui-ci a écrit une instruction par ligne). Cet ordre trivial ne satisfait cependant pas la condition 2 : par exemple, il ferait apparaître les branches *then* et *else* d'un *if* comme successives dans le temps, alors que leurs exécutions sont exclusives. Il ne satisfait pas non plus la condition 3 car il ne permet pas de savoir facilement (sans reparcourir tout le programme) à l'intérieur de quel bloc de contrôle (ou *bloc de base)* donné se situe une instruction. Cette

section présente notre norme d'adressage en traitant *par difficulté croissante* les différents cas de contrôles de flots existant dans les langages actuels. Notons d'abord que notre adressage d'instructions comprend le nom du programme (ou du fichier objet) et de la fonction à laquelle l'instruction appartient. La première partie de l'adresse d'une instruciton est donc son couple (programme, fonction). Sa seconde partie est l'adresse d'une instruction à l'intérieur d'une fonction donnée.

Les structures de contrôle dans les langages classiques (par ex. *if/then/else, case, while, for, do... while*) fonctionnent par *imbrication* : une structure de contrôle peut être vue comme une instruction particulière qui détermine l'exécution de blocs d'instructions successives. Les instructions d'un bloc sont ainsi *imbriquées* par la structure de contrôle, elle-même faisant partie d'un autre bloc d'instructions, lui-même pouvant être imbriqué par une structure de contrôle ou contenu dans une fonction. L'imbrication des blocs d'instructions d'un programme, qui caractérise le flot de contrôle d'un programme, est représentée par son arbre de syntaxe abstrait (AST).

On définit le *niveau d'imbrication* d'une instruction i par le nombre de structures de contrôle imbriquant transitivement le bloc comportant i. Il existe un chemin unique dans l'AST entre l'instruction de passage de paramètres d'une fonction et toute instruction de cette fonction. L'idée de base de notre adressage est d'adresser une instruction par le parcours de ce chemin. Un tel chemin peut être exprimé par un vecteur de $2n+1$ indices, n étant le niveau d'imbrication de l'instruction (le niveau le plus bas étant le niveau 0). À chaque structure de contrôle imbriquant transitivement l'instruction correspondra deux indices, et le dernier indice est le rang séquentiel de l'instruction dans son bloc. Passons en revue les diverses structures de contrôle de flot.

A.3 séquence d'instructions

Les instructions structurées en une liste (*plate*) d'instructions peuvent s'adresser (à l'intérieur de cette liste) par leur ordre séquentiel dans le programme. Pour indiquer que l'instruction appartient à une séquence plate d'instructions, une balise <seq> est attachée à l'indice de séquence.

Exemple 9

```
int fou(int a, int b) {   // instruction (<seq> 0)
  int x;
  x = a = 2 * b;          // instruction (<seq> 1)
  b/=a+x;                 // instruction (<seq> 2)
  return x;               // instruction (<seq> 3)
}
```

□

L'entête de fonction doit être comptée comme la première instruction (numéro 0), car elle correspond à l'affectation de variables (par le mécanisme de passage de paramètres ou celui des valeurs par défaut en C++). Nous prenons l'hypothèse (cohérente dans les langages de notre connaissance) que les paramètres sont indépendants les uns des autres. Cette hypothèse nous permet de considérer le passage de paramètres comme une seule instruction : nous n'interdisons pas les instructions affectant plusieurs variables distinctes et indépendantes.

A.4 contrôle de type *boucle*

Un noeud *while* possède deux fils dans l'AST : son test et le corps (bloc d'instructions). Dans le cas du while, le test est exécuté en premier. La balise <while> est associée à l'indice indiquant le numéro de bloc. Les noeuds sont donc numérotés selon la figure 21a.

Exemple 10 Voyons un exemple d'adressage des instructions dans le cas d'un while.

```
int barre(int a, int b) {   // instruction (<seq>0)
  int x;
  x = a > b ? a : b;        // (<seq> 1)
  while (x > 3) {           // (<seq> 2, <while> 0, <seq> 0)
    x -= 3;                 // (<seq> 2, <while> 1, <seq> 0)
    printf("x = %d\n",x);   // (<seq> 2, <while> 1, <seq> 1)
  }
  printf("Et voilà\n");     // (<seq> 3)
  return x;                 // (<seq> 4)
}
```

Les adresses d'instructions contenues dans le while comportent 3 indices. Regardons l'instruction `x-=3`. Le premier indice, <seq> 2, indique qu'elles appartiennent au bloc contrôlé par l'instruction while dont l'adresse est 2 et appartient à une suite d'instructions. Son second indice indique qu'elle est imbriquée dans une structure *while*, et dans quel fils (ici le corps) elle se situe. Son troisième indice indique qu'elle est la première instruction (numéro 0) du corps. □

Un noeud *do..while* possède les deux mêmes fils dans l'AST qu'un noeud *while*. Mais cette fois, le test est exécuté en dernier. Le fils correspondant au corps porte donc le numéro 0, et le test est numéroté 1 (voir figure 21b). La balise associée est <do>

Exemple 11 Voici un exemple d'adressage des instructions dans le cas d'un do ... while.

```
int barre(int a, int b) {   // instruction (<seq>0)
  int x;
  x = a > b ? a : b;        // (<seq> 1)
  do {
    x -= 3;                 // (<seq> 2, <do> 0, <seq> 0)
    printf("x = %d\n",x);   // (<seq> 2, <do> 0, <seq> 1)
  } while (x > 3);          // (<seq> 2, <do> 1, <seq> 0)
  printf("Et voilà\n");     // (<seq> 3)
  return x;                 // (<seq> 4)
}
```

□

Un noeud *for* possède quatre fils : une initialisation, un test de sortie, un corps et un incrément. Leur ordre dans notre adressage, présenté en figure 21c, reflète leur ordre d'exécution. La balise associée aux fils d'un noeud *for* est <for>.

Exemple 12 Voici un exemple d'adressage des instructions dans le cas d'un for.

```
int barre(int a, int b) {  // instruction (<seq>0)
  int x;
  x = a > b ? a : b;       // (<seq> 1)
```

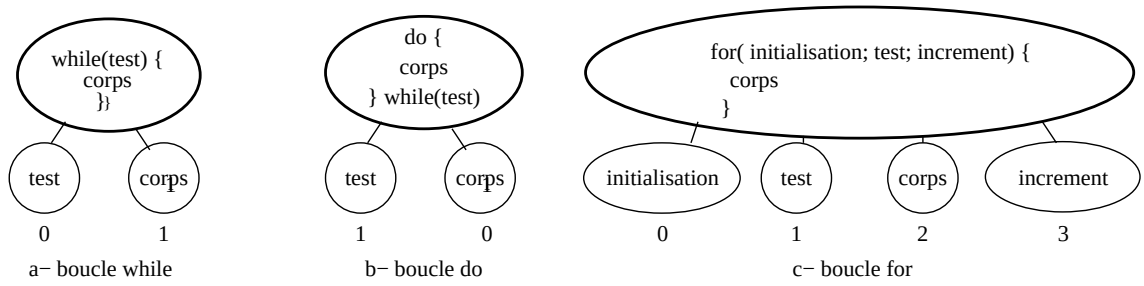

FIG. 21 – Ordre sur les fils des structures de contrôle de type boucle

```
  for (int i = 3,          // (<seq> 2, <for> 0, seq 0)
       j = 5;              // (<seq> 2, <for> 0, seq 1)
           i < x;          // (<seq> 2, <for> 1, seq 0)
           i++,            // (<seq> 2, <for> 3, seq 0)
           j++;) {         // (<seq> 2, <for> 3, seq 1)
     x = a-i;              // (<seq> 2, <for> 2, seq 0)
     b-=j;                 // (<seq> 2, <for> 2, seq 1)
  }
  printf("Et voilà\n");    // (<seq> 3)
}
```

□

A.4.1 contrôle de type branchement conditionnel

Les branchements de type *if/then/else* ou *switch/case* contrôlent l'exécution d'un ensemble de blocs de façon qu'un de ces blocs seulement soit exécuté, à l'exclusion de tous les autres. Les deux types de branchement se distinguent par leur nombre de fils dans l'AST :

- les branchements *if/then/else* on deux ou trois fils : le test, la branche *then* et la branche *else* ;
- les branchements *switch/case* ont $n + 1$ fils, avec $n \geq 1$: l'expression testée et les n cas possibles.

On considèrera donc le branchement *if/then/else* comme un cas particulier du *switch/case*. Leur balise commune est <if/case>. L'exécution des branches étant exclusive, on ne peut pas ordonner les fils selon un ordre d'exécution. Seule l'expression testée est antérieure aux branches. Toutefois, il est nécessaire de distinguer les différentes branches afin d'avoir une adresse d'instructions non-ambiguë. Pour cela, on numérote les branches dans l'ordre dans lequel elles sont écrites par le programmeur.

Exemple 13 Voici un exemple d'adressage des instructions où l'on imbrique des structures if/then/else et switch/case.

```
int barre(int a, int b) {  // instruction (<seq>0)
  int x;
  printf("Coucou\n");      // (<seq> 1)
  if (a < b) {             // (<seq> 2, <if/case> 0, <seq> 0)
    x = a;                 // (<seq> 2, <if/case> 1, <seq> 0)
    b -= x;                // (<seq> 2, <if/case> 1, <seq> 1)
  }
  else {
    x=b;                   // (<seq> 2, <if/case> 2, <seq> 0)
    switch(x) {            // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 0, <seq> 0)
      case 3: {
        printf("x = 3\n"); // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 1, <seq> 0)
        break;             // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 1, <seq> 1)
        }
      case 7: {
        printf("x = 7\n"); // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 2, <seq> 0)
        break;             // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 2, <seq> 1)
        }
      default: continue;   // (<seq> 2, <if/case> 2, <seq> 1, <if/case> 3, <seq> 0)
    }
    a -= x;                // (<seq> 2, <if/case> 2, <seq> 1)
  }
  printf(``niet'');        // (<seq> 3)
}
```

□

A.5 Préordre d'exécution des instructions

Un algorithme simple permet de retrouver un préordre reflétant l'ordre chronologique d'exécution possible des instructions d'après leur adresse. Le préordre *inférieur* respecte la proposition suivante : d non inférieur à b entraîne : pour toute exécution possible du programme, si l'instruction d est exécutée à un moment t, et si l'instruction b est exécutée à un moment t' alors $t > t'$. On peut en extraire une relation *ordre_chronologique* définie de la manière suivante. Soient A = (préfixe(A), milieu(A), suffixe(A)) et B = (préfixe(B), milieu(B), suffixe(B)) deux adresses d'instructions, telles que préfixe(A)=préfixe(B) soit le plus grand préfixe commun entre A et B et où milieu() est un indice (avec sa balise associée). Notons que par construction, si préfixe(A) = préfixe(B) alors tag(milieu(A)) = tag(milieu(B)). Alors on a :

```
ordre_chronologique (A, B) =
   si milieu(A) est vide alors A = B
   sinon, en fonction de tag(milieu(A)):
   - si tag(milieu(A)) = <if/case>
        si milieu(A) = 0 alors A < B
        sinon si milieu(B) = 0 alors B < A
              sinon pas d'ordre chronologique entre A et B
   - si tag(milieu(A)) = <for>
        si milieu(A) = 0 alors A < B
```

```
     sinon si milieu(B) = 0 alors B < A
           sinon A < B et A > B
- si tag(milieu(A)) = <do> ou <while>
     A < B et A > B
- si tag(milieu(A)) = <seq>
     si milieu(A) < milieu (B) alors A < B
        sinon A > B
```

B DTD : fichier address.dtd

Cette DTD, qui spécifie la représentation des adresses des intructions et des opérandes des instructions dans un programme, est incluse par plusieurs de nos DTDs qui sont présentées dans la suite.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*****************************************************************************-->
<!--                                                                             -->
<!--                Statement and operand addressing DTD                         -->
<!--                                                                             -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                         -->
<!--           Benoit Meister <meister@icps.u-strasbg.fr>                        -->
<!--                                                                             -->
<!--*****************************************************************************-->


<!-- address of a statement. -->
<!ELEMENT address (coordinate+)>
<!ATTLIST address file CDATA #REQUIRED
                  function CDATA #IMPLIED>
     <!-- Attributes description:
             - "file", "function": name of the file and function which contain
               the statement.
             Note that function can be "" if the instruction is global. -->
<!--*****************************************************************************-->

<!-- Definition of the element "coordinate" which represents an address'
     coordinate as they are defined in our system of statements addressing. -->
<!ELEMENT coordinate EMPTY>
<!ATTLIST coordinate flag (if_switch|for|while|do|seq|ope) #REQUIRED
                     number CDATA #REQUIRED>
     <!-- Attributes description:
             - the attribute "flag" attached to the coordinates of the data's
               address is used to represent the flow control structure of the
               statement:
                 + 'if' or 'switch ... case ...' ==> flag = if_switch
                 + 'for'                          ==> flag = for
                 + 'while'                        ==> flag = while
                 + 'do ... while'                 ==> flag = do
                 + sequential statement           ==> flag = seq
                 + operand's number               ==> flag = ope
             - the attribute "number" is the statement's number. -->
<!--*****************************************************************************-->
```

C DTD : fichier parts_definition.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*****************************************************************************-->
<!--                                                                             -->
<!--                        DTD for program parts                                -->
<!--                                                                             -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                         -->
<!--           Benoît Meister <meister@icps.u-strasbg.fr>                        -->
<!--                                                                             -->
<!--*****************************************************************************-->


<!-- Importing our statement addressing system. -->
<!ENTITY % statement_addressing SYSTEM "address.dtd">
%statement_addressing;

<!-- Root of the list of parts' definition: "parts_definition" -->
<!ELEMENT parts_definition (part+)>
<!ATTLIST parts_definition project CDATA #REQUIRED>
     <!-- project's name -->
<!--*****************************************************************************-->

<!-- Program Part where the data is -->
<!ELEMENT part (address, address, vector?)>
     <!-- address of beginning and endding -->
<!ATTLIST part id CDATA #REQUIRED
               type (delegable|local) #REQUIRED
               in_loop (yes|no) #REQUIRED>
     <!-- Attributes' description:
          * "id" is the part's identifier.
          * "type" describes the part's type (local or delegable).
          * "in_loop" tells if the part is enclosed in a (maybe nested) loop. -->
<!--*****************************************************************************-->

<!-- Dependence vector between the different instances of a part -->
<!ELEMENT vector (#PCDATA)>
<!ATTLIST vector nb_index CDATA #REQUIRED>
     <!-- the number of loops which enclose the part. -->
<!--*****************************************************************************-->
```

D Fichier de configuration généré par embedded2grid

```
# Embedded2Grid 0.0.0 __ Sep  3 2003

# This file describes the settings to be used by the transformation and
# distribution system, embedded2grid.
#
# All text after a hash (#) is considered as a comment and will be ignored.
# The format is:
#       TAG = value [value ...]
# For lists items, you can use a backslash (\) to continue on the next line
# as in a Makefile:
#       TAG = value ... value \
#             value [value ...]
#
# N.B.:
#   -1- The maximum length of a line of this configuration file is 126
# characters (but you can use some backslashes (\) to define a tag on
# several lines).
#   -2- The values used for the definition can't contain neither the
# characters '\' or '#' or ':' (':' is used as a separator for some
# personalized tags).
#   -3- The tags are sorted and must stay as they are in order to be
# correctly parsed.
#   -4- Don't mix comments with some lines of a multi-lines definition.
#   -5- Don't use '~' to define the path (it is not always well supported).


#------------------------------------------------------------------------------
# General configuration.
#------------------------------------------------------------------------------

# The COMPILER tag can be used to specify the full name of the compiler.
# If it is not defined the compiler is supposed to be "gcc".

COMPILER =

# The LINKER tag can be used to specify the full name of the linker. If
# it is not defined the linker is supposed to be "gcc".

LINKER =

# The TRANSFORM tag is used to tell if the project must be transformed.
# The transformation is in fact a decomposition of the initial project in
```

```
# several tasks. Some of these tasks may be executed by some other computers
# reachable via network. The possible values for this tag are YES and NO (set
# NO to do only the analysis as for the libraries).

TRANSFORM =


#---------------------------------------------------------------------------
# Project configuration.
#---------------------------------------------------------------------------

# The PROJECT_NAME tag is a single word that should identify the project.

PROJECT_NAME =

# The PROJECT_NUMBER tag can be used to enter a project or revision number.
# This could be handy for archiving the extracted informations.

PROJECT_NUMBER =


#---------------------------------------------------------------------------
# Configuration options related to the input files of the project.
#---------------------------------------------------------------------------

# The HOMEDIR tag is used to specify the path to the main directory of
# the project.

HOMEDIR =

# The SRCDIRS tag can be used to specify the directories that contain
# the source files of the project (the names specified here must describe
# the relative path from the main directory of the project to the source
# directory). Even if the main directory contains some source files it must
# not be specified in this tag.

SRCDIRS =

# The SRCFILES_<src-dir-name> tags are some lists of words that identify
# the source files of the project (the headers have not to be specified).
# <src-dir-name> represent a source directory different from this specified
# by the tag "HOMEDIR". If the main directory contains some source files,
# use the tag "SRCFILES_" to specify them.


#---------------------------------------------------------------------------
# Configuration options related to the project compilation.
#---------------------------------------------------------------------------
```

```
# The COMPILE_FLAGS tag can be used to enter options of compilation (for
# example: -Wall -fwritable-strings -I<dir> -B<dir>).

COMPILE_FLAGS =

# If you want to specify special flags of compilation for some source files,
# you can create the corresponding tags SPECFLAGS_<src-dir>:<src-file>.
# These flags will be used in addition of them defining the tag COMPILE_FLAGS
# which are shared by all the source files of the project. If you want to
# specify special flags of compilation for some source files enclosed in the
# main directory of the project, use the tag "SPECFLAGS_:<src-file>".


# The LINK_FLAGS tag can be used to enter special options used to link the
# object files (for example: -lm -lxml2 -lz -L<dir>).

LINK_FLAGS =


#---------------------------------------------------------------------------
# configuration options related to the output files of the project.
#---------------------------------------------------------------------------

# The OUTPUT tag can be used to specify the directory (with its relative
# path from the main directory) that will contain the executables resulting
# of our transformations and the final results of the analysis. By default,
# they will be stored in the main directory of the project. If the specified
# output directory doesn't exist, it will be created.

OUTPUT =


#---------------------------------------------------------------------------
# Dependencies configuration.
#---------------------------------------------------------------------------

# The LIBRARIES tag can be used to give the full name of the libraries
# whose some functions are called by the project.

LIBRARIES =

# If you want to have the best data liveness analysis of your project, you
# must have the data liveness analysis of each "foreign" function called in
# the project and their function call graph. If you have not already the
# corresponding XML files (two per library), then if you have the sources of
# the libraries you should analyze them by creating one configuration file
# for each of them (else the analysis of the project will contain some
# approximations). If you have the XML files relative to the libraries, you
# should specify them by creating two new tags per library:
```

```
#    - CGRAPHFILE_<full-lib-name> which contains the name of the XML file
#      describing the function call graph of the library <full-lib-name> (the
#      name must represent the absolute path to this file).
#    - LIVEFILE_<full-lib-name> which contains the name of the XML file
#      describing the data liveness of the library <full-lib-name> (the name
#      must represent the absolute path to this file).
```

E DTD : fichier conditional_branches.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*************************************************************************-->
<!--                                                                         -->
<!--                      DTD for conditional branches                       -->
<!--                                                                         -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                     -->
<!--           Benoît Meister <meister@icps.u-strasbg.fr>                    -->
<!--                                                                         -->
<!--*************************************************************************-->


<!-- Importing our statement addressing system. -->
<!ENTITY % statement_addressing SYSTEM "address.dtd">
%statement_addressing;

<!-- Root of the list of conditional branches: "conditional_branches" -->
<!ELEMENT conditional_branches (nest*)>
<!ATTLIST conditional_branches project CDATA #REQUIRED>
     <!-- project's name -->
<!--*************************************************************************-->

<!-- Definition of the element "file" which describes the conditional nests
     enclosed in a file. -->
<!ELEMENT file (function*)>
<!ATTLIST file name CDATA #REQUIRED>
     <!-- it is the file's name. -->
<!--*************************************************************************-->

<!-- Definition of the element "function" which describes the conditional nests
     enclosed in a function. -->
<!ELEMENT function (nest*)>
<!ATTLIST function name CDATA #REQUIRED>
     <!-- it is the function's name. -->
<!--*************************************************************************-->

<!-- Description of a confitional nest -->
<!ELEMENT nest (coordinate+)>
    <!-- the coordinates of a conditional test -->
<!ATTLIST nest nb_branches CDATA #REQUIRED>
    <!-- "nb_branches" is the number of branches issued from the conditional
         nest -->
<!--*************************************************************************-->
```

F DTD : fichier call_graph.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--**************************************************************************-->
<!--                                                                          -->
<!--                      Functions Calls graph DTD                           -->
<!--                                                                          -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                      -->
<!--           Benoit Meister <meister@icps.u-strasbg.fr>                     -->
<!--                                                                          -->
<!--**************************************************************************-->


<!-- Importing our statement addressing system. -->
<!ENTITY % statement_addressing SYSTEM "address.dtd">
%statement_addressing;

<!-- Root of the tree of the functions calls:  "call_graph" -->
<!ELEMENT call_graph (file*)>
<!ATTLIST call_graph project CDATA #REQUIRED>
     <!-- it is the project's name. -->
<!--**************************************************************************-->

<!-- Definition of the element "file" which describes the functions calls
     enclosed in a file. -->
<!ELEMENT file (function*)>
<!ATTLIST file name CDATA #REQUIRED>
     <!-- it is the file's name. -->
<!--**************************************************************************-->

<!-- Definition of the element "function" which describes the functions called
     by another function. -->
<!ELEMENT function (call*)>
<!ATTLIST function name CDATA #REQUIRED
                   static (yes|no) #REQUIRED>
     <!-- Attributes' description:
          * "name" is the function's name.
          * "static" is a flag to tell if the function is declared 'static'. -->
<!--**************************************************************************-->
```

```
    <!-- Definition of the element "call" which represents a function call. -->
    <!ELEMENT call (coordinate+)>
         <!-- it contains the address' coordinates of the function call. -->
<!ATTLIST call function CDATA #REQUIRED
              built_in (yes|no) #REQUIRED>
         <!-- Attributes' description:
              * "function" is the name of the callee function.
              * "built_in" is a flag to tell if the callee function is built-in. -->
<!--*************************************************************************-->
```

G DTD : fichier types_definition.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--**************************************************************************-->
<!--                                                                          -->
<!--                    DTD for the variables' data types                     -->
<!--                                                                          -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                      -->
<!--           Benoit Meister <meister@icps.u-strasbg.fr>                     -->
<!--                                                                          -->
<!--**************************************************************************-->


<!-- Root of the tree of data types' definition: "data_types_definition". -->
<!ELEMENT data_types_definition (data_type*)>
<!ATTLIST data_types_definition project CDATA #REQUIRED>
     <!-- project's name -->
<!--**************************************************************************-->

<!-- Definition of a data type: "data_type".
     It can be a simple type, an union, an enumeration, a structure, an array
     a pointer or a function (in order to represent the pointers to a
     function). -->
<!ELEMENT data_type (simple|enum|union|struct|array|pointer|function)>
<!ATTLIST data_type key CDATA #REQUIRED
                    memory CDATA #REQUIRED
                    const (yes|no) #REQUIRED
                    volatile (yes|no) #REQUIRED
                    name CDATA #IMPLIED>
     <!-- Attributes' description:
          * "key" is an uniquifier for the data type.
          * "memory" gives the needed memory size in bits.
          * "const" and "volatile" are used to store the qualifiers of the
            data type.
          * "name" is the name of the data type given by a typedef, if any. -->
<!--**************************************************************************-->

<!-- The element "simple" is used to describe a simple data type. -->
<!ELEMENT simple EMPTY>
<!ATTLIST simple name CDATA #REQUIRED
                 signed (yes|no) #REQUIRED
                 size (short|normal|long|longlong) #REQUIRED>
     <!-- Attributes' description:
          * "name" is the data type's name.
```

```
          * "signed" is a flag to know if the simple data type is signed.
          * "size" is a flag to know the data type's size. -->
    <!--************************************************************************-->

    <!-- The element "union" is used to describe a union and its fields (each of
         them is described by an element "field". -->
    <!ELEMENT union (field+)>
    <!ATTLIST union name CDATA #IMPLIED>
         <!-- The union's name, if any. -->
    <!--************************************************************************-->

    <!-- The element "enum" is used to describe an enumeration. -->
    <!ELEMENT enum (constant+)>
    <!ATTLIST enum name CDATA #IMPLIED>
         <!-- The enumeration's name, if any. -->
    <!--************************************************************************-->

    <!-- The element "constant" can describe a constant of an enumeration. -->
    <!ELEMENT constant EMPTY>
    <!ATTLIST constant name CDATA #REQUIRED
                       value CDATA #REQUIRED>
         <!-- Attributes' description:
              * "name" is the constant's name,
              * "value" is the constant's value. -->
    <!--************************************************************************-->

    <!-- The element "struct" is used to describe a structure and its fields (each
         of them is described by an element "field". -->
    <!ELEMENT struct (field+)>
    <!ATTLIST struct name CDATA #IMPLIED>
         <!-- The structure's name, if any. -->
    <!--************************************************************************-->

    <!-- The element "field" can describe a field of an union or a structure. -->
    <!ELEMENT field EMPTY>
    <!ATTLIST field type CDATA #REQUIRED
                    name CDATA #REQUIRED
                    bit_field CDATA #IMPLIED>
         <!-- Attributes' description:
              * "type" is a data type's identifier.
              * "name" is the name of the field.
              * "bit_field" is the number of bits used when the field is a bit
                field.-->
    <!--************************************************************************-->

    <!-- The element "array" is used to represent a static array. -->
    <!ELEMENT array EMPTY>
    <!ATTLIST array number CDATA #REQUIRED
                    type CDATA #REQUIRED>
```

```
95        <!-- Attributes' description:
               * "number" is the number of elements in the static array.
               * "type" is a data type's identifier describing the type of the
                 elements. -->
     <!--************************************************************************-->
100
     <!-- The element "pointer" is used to represent a pointer. -->
     <!ELEMENT pointer EMPTY>
     <!ATTLIST pointer type CDATA #REQUIRED
                       restrict (yes|no) #REQUIRED>
105       <!-- Attributes' description:
               * "type" is the key of the type of the pointed data.
               * "restrict" is a flag to tell if the pointer has been declared
                 'restrict'.  -->
     <!--************************************************************************-->
110
     <!-- The element "function" is used to represent a function (in order to be
          able to represent the pointers to a function). -->
     <!ELEMENT function (argument*)>
     <!ATTLIST function result CDATA #REQUIRED>
115       <!-- "result" is the key of the returned data type.  -->
     <!--************************************************************************-->

     <!-- The element "argument" is used to represent a function's argument. -->
     <!ELEMENT argument EMPTY>
120  <!ATTLIST argument type CDATA #REQUIRED>
          <!-- "type" is the key of the argument's data type.  -->
     <!--************************************************************************-->
```

H DTD : fichier local_variables.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*************************************************************************-->
<!--                                                                         -->
<!--                  Local variables definition DTD                         -->
<!--                                                                         -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                     -->
<!--           Benoit Meister <meister@icps.u-strasbg.fr>                    -->
<!--                                                                         -->
<!--*************************************************************************-->


<!-- Root of the tree of local variables definition:  "local_variables".
     This element may contain some elements file. -->
<!ELEMENT local_variables (file*)>
<!ATTLIST local_variables project CDATA #REQUIRED>
     <!-- It's the project's name -->
<!--*************************************************************************-->

<!-- Definition of the element "file".
     This element may contain some elements function. -->
<!ELEMENT file (function*)>
<!ATTLIST file name CDATA #REQUIRED>
     <!-- It is the file's name. -->
<!--*************************************************************************-->

<!-- Definition of the element "function".
     This element contains some elements variable (representing the variables
     which are local to this function). -->
<!ELEMENT function (variable*)>
<!ATTLIST function name CDATA #REQUIRED>
     <!-- It is the function's name. -->
<!--*************************************************************************-->

<!-- Definition of the element "variable". It describes a local variable
     declaration -->
<!ELEMENT variable EMPTY>
<!ATTLIST variable name CDATA #REQUIRED
                   key CDATA #REQUIRED
                   type CDATA #REQUIRED
                   qual (auto|static) #REQUIRED
                   register (yes|no) #REQUIRED
                   init (yes|no) #IMPLIED>
```

```
        <!-- Attibutes' description:
             * "name" is the variable's name.
             * "key" is a number which is one of the two parts of the multiple
               primary key of the variables table (the second part of the key is
               the variable's name).
             * "type" is the key of the variable's data type.
             * "qual" is a flag to tell if the variable has been declared
               'auto' (by default) or 'static'.
             * "register" is a flag to tell if the variable has been declared
               'register'.
             * "init" is a flag to tell if the variable has been initialized
               during its declaration, it's only used for the variables whose the
               storage class is 'static'. -->
    <!--*************************************************************************-->
```

I DTD : fichier global_variables.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*****************************************************************************-->
<!--                                                                             -->
<!--                 Global variables definition DTD                             -->
<!--                                                                             -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                         -->
<!--           Benoit Meister <meister@icps.u-strasbg.fr>                        -->
<!--                                                                             -->
<!--*****************************************************************************-->


<!-- Root of the tree of local variables definition:  "local_variables".
     This element may contain some elements file. -->
<!ELEMENT global_variables (file*)>
<!ATTLIST global_variables project CDATA #REQUIRED>
     <!-- It's the project's name -->
<!--*****************************************************************************-->

<!-- Definition of the element "file".
     This element may contain some elements "variable" -->
<!ELEMENT file (variable*)>
<!ATTLIST file name CDATA #REQUIRED>
     <!-- It is the file's name. -->
<!--*****************************************************************************-->

<!-- Definition of the element "variable". It describes a global variable
     declaration. -->
<!ELEMENT variable EMPTY>
<!ATTLIST variable name CDATA #REQUIRED
                   key CDATA #REQUIRED
                   type CDATA #REQUIRED
                   qual (auto|static|extern) #REQUIRED
                   register (yes|no) #REQUIRED
                   init (yes|no) #IMPLIED>
     <!-- Attibutes' description:
          * "name" is the variable's name.
          * "key" is an uniquifier for the variable in the project.
          * "type" is the key of the variable's data type.
          * "qual" is a flag to tell if the variable has been declared
            'auto' (by default) or 'static' or 'extern'.
          * "register" is a flag to tell if the variable has been declared
            'register'.
```

```
          * "init" is a flag to tell if the variable has been initialized
            during its declaration, it's only used for the variables whose the
            storage class is 'static'. -->
<!--*****************************************************************************-->
```

J DTD : fichier tmp_global_static_data_liveness.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*****************************************************************************-->
<!--                                                                             -->
<!--            Global Variables Static Data Liveness' DTD                       -->
<!--                                                                             -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                         -->
<!--           Benoît Meister <meister@icps.u-strasbg.fr>                        -->
<!--                                                                             -->
<!--*****************************************************************************-->


<!-- Importing our statement addressing system. -->
<!ENTITY % statement_addressing SYSTEM "address.dtd">
%statement_addressing;

<!-- Root of the tree of static data liveness' definition:  "liveness" -->
<!ELEMENT liveness ((production|consumption)*)>
<!ATTLIST liveness project CDATA #REQUIRED>
     <!-- project's name -->
<!--*****************************************************************************-->

<!-- Definition of a "production". -->
<!ELEMENT production (address)>
   <!-- The address of the statement in which the variable is produced. -->
<!ATTLIST production name CDATA #REQUIRED
                     key CDATA #REQUIRED>
     <!-- Attributes' description:
          * "name" is the variable's name.
          * "key" is the variable's key.
           The couple (name, key) is an identifier for the variable. -->
<!--*****************************************************************************-->

<!-- Definition of a "consumption". -->
<!ELEMENT consumption (address)>
   <!-- The address of the statement in which the variable is consumed. -->
<!ATTLIST consumption name CDATA #REQUIRED
                      key CDATA #REQUIRED
                      state (tmp|sure|final) #REQUIRED>
     <!-- Attributes' description:
          * "name" is the variable's name.
          * "key" is the variable's key.
            The couple (name, key) is an identifier for the variable.
```

```
            * "state" is a flag to know if the consumption is in the adequate
              element octopus:
                + "state" = "tmp":
                  Some functions called between this consumption and the
                  corresponding productions might produce some datas which are
                  concerned by the octopus (the data liveness of these functions
                  is not yet known...).
                + "state" = "sure":
                  Some functions are called between this consumption and the
                  corresponding productions but they don't produce any concerned
                  datas (the data liveness of the callee functions is already
                  known).
                + "state" = "final":
                  There is none function call between this consumption and the
                  corresponding productions. -->
    <!--**************************************************************************-->
```

K DTD : fichier local_static_data_liveness.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--**************************************************************************-->
<!--                                                                          -->
<!--            Local Variables Static Data Liveness' DTD                     -->
<!--                                                                          -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                      -->
<!--           Benoît Meister <meister@icps.u-strasbg.fr>                     -->
<!--                                                                          -->
<!--**************************************************************************-->


<!-- Importing our statement addressing system. -->
<!ENTITY % statement_addressing SYSTEM "address.dtd">
%statement_addressing;

<!-- Root of the tree of data liveness' definition:  "liveness" -->
<!ELEMENT liveness (file*)>
<!ATTLIST liveness project CDATA #REQUIRED>
     <!-- project's name -->
<!--**************************************************************************-->

<!-- Definition of the element "file" which describes the static data liveness
     of the local variables enclosed in a file. -->
<!ELEMENT file (function*)>
<!ATTLIST file name CDATA #REQUIRED>
     <!-- it is the file's name. -->
<!--**************************************************************************-->

<!-- Definition of the element "function" which describes the static data
     liveness of the local variables enclosed in a function. -->
<!ELEMENT function (pool*)>
<!ATTLIST function name CDATA #REQUIRED>
     <!-- it is the function's name. -->
<!--**************************************************************************-->

<!-- Definition of a "pool":
     The elements "octopus" stored in the same element "pool" share some
     elements "consumption". -->
<!ELEMENT pool (octopus+)>
<!--**************************************************************************-->

<!-- Definition of an "octopus":
```

```
     An element "octopus" contains one element "production" and all the
     corresponding elements "consumption". -->
<!ELEMENT octopus (production, consumption*)>
<!--*****************************************************************************-->

<!-- Definition of a "production". -->
<!ELEMENT production (coordinate+)>
   <!-- The address of the statement in which the variable is produced. -->
<!ATTLIST production name CDATA #REQUIRED
                     key CDATA #REQUIRED>
     <!-- Attributes' description:
          * "name" is the variable's name.
          * "key" is the variable's key.
            The couple (name, key) is an identifier for the variable. -->
<!--*****************************************************************************-->

<!-- Definition of a "consumption". -->
<!ELEMENT consumption (coordinate+)>
   <!-- The address of the statement in which the variable is consumed. -->
<!ATTLIST consumption name CDATA #REQUIRED
                      key CDATA #REQUIRED
                      state (tmp|sure|final) #REQUIRED>
     <!-- Attributes' description:
          * "name" is the variable's name.
          * "key" is the variable's key.
            The couple (name, key) is an identifier for the variable.
          * "state" is a flag to know if the consumption is in the adequate
            element octopus:
              + "state" = "tmp":
                Some functions called between this consumption and the
                corresponding productions might produce some datas which are
                concerned by the octopus (the data liveness of these functions
                is not yet known...).
              + "state" = "sure":
                Some functions are called between this consumption and the
                corresponding productions but they don't produce any concerned
                datas (the data liveness of the callee functions is already
                known).
              + "state" = "final":
                There is none function call between this consumption and the
                corresponding productions. -->
<!--*****************************************************************************-->
```

L DTD : fichier datasets_appointements.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Encoding "UTF-8" exclusively -->


<!--*****************************************************************************-->
<!--                                                                             -->
<!--            DTD for Datasets and communicating points                        -->
<!--                                                                             -->
<!--           Gilles Bitran <bitran.gilles@laposte.net>                         -->
<!--           Benoît Meister <meister@icps.u-strasbg.fr>                        -->
<!--                                                                             -->
<!--*****************************************************************************-->


<!-- Root of the tree of datasets and communicating points definition:
     "communication" -->
<!ELEMENT communication (association*)>
<!ATTLIST communication project CDATA #REQUIRED>
     <!-- project's name -->
<!--*****************************************************************************-->

<!-- Definition of an "association" :
        + a list of data sets,
        + a list of receiving points. -->
<!ELEMENT association (dataset+, receive+, subdataset+)>
<!ATTLIST association id CDATA #REQUIRED>
     <!-- association's identifier -->
<!ATTLIST association sending_part CDATA #IMPLIED>
     <!-- identifier of the part of production and sending -->
<!--*****************************************************************************-->

<!-- Definition of a "dataset" :
        + a list of sub-data sets,
        + a sending point. -->
<!ELEMENT dataset (subdataset_id+, send)>
<!ATTLIST dataset id CDATA #REQUIRED>
     <!-- data set's identifier -->
<!--*****************************************************************************-->

<!-- Definition of a 'subdataset_id' -->
<!ELEMENT subdataset_id (#PCDATA)>
     <!-- it is only the identifier of a sub-data set which is enclosed in a
data set. -->
<!--*****************************************************************************-->

```

```
    <!-- Definition of a sub-set of datas "subdataset" -->
    <!ELEMENT subdataset (data+)>
    <!ATTLIST subdataset id CDATA #REQUIRED>
         <!-- sub-data set's identifier -->
<!--*****************************************************************************-->

    <!-- Definition of a sending point : "send" -->
    <!ELEMENT send (address)>
    <!ATTLIST send id CDATA #REQUIRED>
         <!-- send's identifier -->
    <!--*****************************************************************************-->

    <!-- Definition of a receiving point : "receive" -->
    <!ELEMENT receive (address, dataset_id+)>
<!ATTLIST receive id CDATA #REQUIRED>
         <!-- receive's identifier -->
    <!ATTLIST receive rcv_part CDATA #REQUIRED>
         <!-- identifier of the part of consumption and receiving -->
    <!--*****************************************************************************-->

    <!-- Definition of a 'dataset_id' -->
    <!ELEMENT dataset_id (#PCDATA)>
         <!-- it is only the identifier of a data set which is concerned by a
    receiving point. -->
<!--*****************************************************************************-->

    <!-- Definition of a data:
             + the name of the variable,
             + the type of the variable,
             + the scope of the variable,
     (the production and the consumption of a data are done in different parts -->
    <!ELEMENT data (name, type, scope)>
    <!ATTLIST data rank CDATA #REQUIRED>
         <!-- data's rank in the data set -->
<!--*****************************************************************************-->
```