# Why3 a dit : gardez le contrôle en toute situation

Jean-Christophe Léchenet[1,2], Nikolai Kosmatov[1], and Pascale Le Gall[2]

[1] CEA, LIST, Laboratoire de Sûreté des Logiciels
PC 174, 91191 Gif-sur-Yvette Cedex
prenom.nom@cea.fr
[2] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
prenom.nom@centralesupelec.fr

**Résumé**

Le slicing est une technique permettant d'extraire, à partir d'un programme donné, un programme plus petit, appelé tranche ou slice, tel que le programme et sa slice aient un comportement identique vis-à-vis d'un critère donné (appelé critère de slicing). Le calcul de la slice est classiquement basé sur des dépendances : dépendances de contrôle et dépendances de donnée. Notre objectif actuel est de construire un outil de slicing générique, qui abstraie les spécificités des langages et qui soit réutilisable, en évitant de reconstruire un outil de slicing pour chaque nouveau langage. Une étape majeure dans cette direction est de traiter les flots de contrôle non structurés.

Une méthode de calcul des dépendances de contrôle pour tout type de flot de contrôle a été proposée et prouvée (sur papier) en 2011 par Danicic et al pour un graphe orienté (fini) quelconque. Dans ce travail, nous avons réalisé une formalisation de cet algorithme dans Coq. Nous proposons également un nouvel algorithme pour le calcul des dépendances de contrôle qui optimise la technique de Danicic. L'optimisation consiste à enregistrer des informations intermédiaires sur les chemins du graphe et s'appuyer sur ces informations pour accélérer les itérations suivantes. Cela rend l'algorithme et la preuve plus complexes, et nécessite des invariants plus subtils. Nous formalisons et prouvons le nouvel algorithme dans l'outil Why3. Afin de comparer notre algorithme à celui de Danicic, nous faisons une évaluation expérimentale des deux algorithmes sur des milliers de graphes générés aléatoirement et allant jusqu'à plusieurs milliers de nœuds. Les résultats montrent que la nouvelle technique est nettement plus efficace et peut s'appliquer à des graphes (et, donc, des CFGs de programmes) réels.

## 1 Introduction

Le *slicing* est une technique proposée par Mark Weiser en 1979 [14, 15] pour simplifier un programme vis-à-vis d'un point d'intérêt appelé critère de slicing. L'objectif du slicing est de créer un programme plus petit que le programme initial mais ayant le même comportement que ce dernier vis-à-vis du critère de slicing. Le calcul de la slice se décompose généralement en deux phases : d'abord, le calcul de l'ensemble d'instructions à préserver, le *slice set*, basé sur les dépendances de contrôle et de données ; puis la construction de la slice elle-même à partir du programme initial et du slice set. Depuis 1979, plusieurs variantes de slicing ont été proposées. Celle de Weiser est désormais nommée *slicing statique arrière.* Cette variante est considérée aussi dans cet article, dans sa version intra-procédurale.

Selon Weiser, la slice est obtenue à partir du programme initial en supprimant une ou plusieurs instructions qui n'ont pas d'impact sur le critère de slicing. Il convient de préciser qu'une instruction est toujours supprimée avec toutes les instructions qu'elle contient le cas échéant (pour des boucles ou des branches de conditions). On ne peut, par exemple, supprimer la condition d'une boucle sans supprimer son corps. Lorsqu'une slice peut ainsi être construite par simples suppressions à partir du programme initial, on dit qu'elle est un *quotient* de ce dernier,

```
1   if (x >= 0) {
2     while (x > 0) {
3       x--;
4     }
5     x++;
6   } else {
7     while (x < 0) {
8       x++;
9     }
10    x--;
11  }
```

(a) Programme original

```
1   if (x >= 0) {
2
3
4
5
6   } else {
7     while (x < 0) {
8       x++;
9     }
10
11  }
```

(b) Slice par rapport à la ligne 8

FIGURE 1 – Programme jouet et sa slice

selon la dénomination utilisée dans [3]. Une limitation de cette définition, déjà identifiée par Weiser [14] et reprise par d'autres travaux [13, 8, 4], vient du fait que, pour certains langages, la suppression d'une instruction peut conduire à un programme mal formé. Une autre limitation apparaît lorsqu'on souhaite optimiser le slicing grâce à une analyse statique préalable. Par exemple, si l'analyse de valeurs garantit que la condition d'une conditionnelle est toujours fausse, on pourrait vouloir garder le contenu de la branche `else` et supprimer la branche `then` et la conditionnelle elle-même dans la slice, qui ne serait alors pas un quotient. D'autres limitations en présence de sauts (`goto`) sont illustrées ci-dessous.

Dans un travail précédent [9, 11, 2], nous avons formalisé le slicing et prouvé sa correction en Coq [12, 5] pour un langage structuré simple WHILE. La figure 1 montre l'exemple d'un programme structuré et de sa slice. Le programme de la figure 1(a) est un programme jouet écrit dans un langage semblable au C. Il incrémente ou décrémente la variable `x` suivant qu'elle est positive ou strictement négative au début de l'exécution du programme, et refait un pas en arrière, de telle sorte que la variable `x` est mise à `1` si `x >= 0`, `-1` sinon. La figure 1(b) contient la slice de ce programme par rapport à l'instruction de la ligne 8. On peut remarquer que c'est bien un quotient du programme initial puisqu'elle peut être obtenue à partir de ce dernier en supprimant les instructions 2–5. Si une analyse statique préliminaire montre que la condition de la ligne 1 est toujours fausse dans cet exemple, on pourrait vouloir garder dans la slice uniquement les lignes 7–9, mais cette slice ne serait pas un quotient.

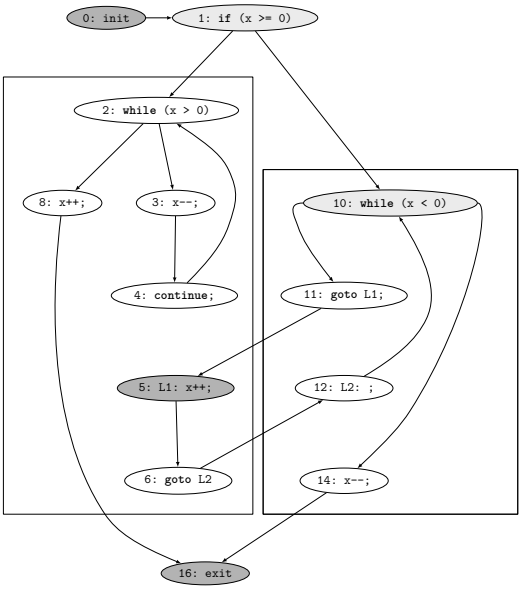
Dans un langage plus riche, avec des opérateurs moins contraints comme `goto`, on peut aussi vouloir s'affranchir de cette contrainte du quotient. Le programme 1(a) a été transformé pour donner un programme au même comportement présenté à la figure 2(a). Il est identique au premier programme si ce n'est que le corps de la deuxième boucle est inséré dans le corps de la première grâce à des `goto`. Plus précisément, les `goto` aux lignes 11 et 6 permettent l'exécution de la ligne 5 lorsque le corps de la deuxième boucle est exécuté, tandis que le `continue` de la ligne 4 fait en sorte que les lignes 5–6 soient contournées lorsque le corps de la première boucle est exécuté.

Intuitivement, on souhaiterait que la slice par rapport à la ligne 5 soit similaire à celle de la figure 1(b). Il faudrait notamment supprimer la boucle ligne 2. Mais évidemment la ligne 5, faisant partie du critère de slicing, doit être préservée (cf. fig. 2(b)). Cependant, la règle stricte imposant que la slice soit un quotient interdit de supprimer un `while` (ici, ligne 2) sans supprimer aussi le corps de la boucle.

Pour s'affranchir de cette règle, on souhaite construire un *slicer* abstrayant les spécificités

```
1   if (x >= 0) {
2     while (x > 0) {
3       x--;
4       continue;
5       L1: x++;
6       goto L2;
7     }
8     x++;
9   } else {
10    while (x < 0) {
11      goto L1;
12      L2: ;
13    }
14    x--;
15  }
```

(a) Programme réécrit avec des `goto`




(b) CFG, et le slice-set désiré $W = \{0, 1, 5, 10, 16\}$

FIGURE 2 – Programme jouet modifié et son CFG. Le critère de slicing est $V_0' = \{0, 5, 16\}$.

du langage d'entrée. On souhaite de plus qu'il soit indépendant du langage, pour pouvoir le réutiliser quel que soit le langage considéré. Une étape majeure dans l'écriture d'un tel slicer est un algorithme générique calculant les dépendances de contrôle y compris en présence de flots de contrôle non structurés.

En 2011, Danicic et al [6] ont proposé une théorie des dépendances de contrôle qui cherche à capturer l'essence de ce type de dépendance. Leur formalisation représente les programmes manipulés par le slicing sous la forme de graphes de flot de contrôle (CFG) et les critères de slicing sous la forme d'ensembles de sommets. La figure 2(b) représente le programme de la figure 2(a) sous la forme d'un CFG. Le critère de slicing $V_0' = \{0, 5, 16\}$ (en gris foncé) correspond à celui utilisé pour la figure 1. Il contient en plus les nœuds d'entrée et de sortie. La slice est alors calculée sous forme d'un CFG (et n'a pas forcément de représentation canonique par un programme). De plus, dans cette approche, si la condition de la ligne 1 est prouvée toujours fausse, on peut simplement retirer l'arête (1,2) pour ne plus devoir garder cette condition dans le slice set.

**Contributions.** [1] Nous avons implémenté en Coq la théorie des dépendances de contrôle proposée par Danicic et al qui généralise les autres relations de ce type (cf. [6]). Leur formalisation contient notamment une notion de dépendance de contrôle applicable à des graphes orientés arbitraires, un algorithme calculant des ensembles stables par rapport à celle-ci, ainsi que la preuve de correction (sur papier) de cet algorithme. Ces trois concepts font partie de notre formalisation en Coq qui permet d'extraire une implémentation certifiée. Pour pallier une faible efficacité de cet algorithme (dont l'amélioration a été supposée possible par ses auteurs [6]), nous avons également proposé un nouvel algorithme, bénéficiant d'un stockage de résultats de calcul intermédiaires et leur réutilisation dans les itérations suivantes. Nous avons prouvé

1. La formalisation de l'algorithme de Danicic en Coq, la formalisation du nouvel algorithme dans Why3 et nos implémentations des deux algorithmes en OCaml sont disponibles sur http://perso.ecp.fr/~lechenetjc/control/.

cet algorithme dans l'outil Why3 [1, 7] et démontré par des expérimentations ses meilleures performances par rapport à l'algorithme initial de Danicic.

Cet article court est basé sur les résultats d'un papier soumis [10]. La section 2 présente la notion de dépendance de contrôle et le principe de l'algorithme introduits par Danicic et al. Puis, la section 3 présente notre algorithme optimisé, discute de sa vérification et des expérimentations. Enfin, la section 4 conclut l'article.

# 2 Stabilité par dépendances de contrôle faibles

Cette section présente la théorie des dépendances de contrôle faibles développée dans [6]. Dans cette section, $G$ est un graphe orienté fini et $V'$ un sous-ensemble des sommets de $G$. On illustrera les définitions sur le CFG de la figure 2(b). Pour les exemples, on utilisera le critère de slicing $V'_0 = \{0, 5, 16\}$.

## 2.1 Définitions

Le concept d'ensemble stable par dépendances de contrôle faibles (SDCF, *weakly control-closed* dans [6]) généralise la notion de dépendance de contrôle à des graphes orientés finis arbitraires. En ne considérant que les dépendances de contrôle, l'ensemble de sommets à préserver dans la slice (le slice set) est le plus petit ensemble SDCF contenant l'ensemble de sommets formant le critère de slicing.

**Définition 2.1** (Image d'un sommet, *first observable element*)**.** *Soit $u$ et $v$ deux sommets de $G$ et $v \in V'$. On dit que $v$ est une* image *de $u$ dans $V'$ s'il existe un chemin de $u$ à $v$ dont l'intersection avec $V'$ est égale à $v$. En d'autres termes, il existe un chemin depuis $u$ vers $v$ sur lequel le premier élément de $V'$ atteint est $v$. En particulier, les sommets dans $V'$ ont pour seule image eux-mêmes.*

Par exemple, l'ensemble des images du sommet 16 dans $V'_0$ est $\{16\}$, celui de 11 est $\{5\}$, celui de 10 est $\{5, 16\}$, celui de 1 est également $\{5, 16\}$.

**Définition 2.2** (Attracteur faible, *weakly committing vertex*)**.** *Un sommet $u$ est un* attracteur faible *vers $V'$ dans $G$ s'il possède au plus une image dans $V'$.*

D'après ci-dessus, les sommets 11 et 16 sont des attracteurs faibles vers $V'_0$, 1 et 10 n'en sont pas.

**Définition 2.3** (Ensemble SDCF, *weakly control-closed set*)**.** *$V'$ est* stable par dépendances de contrôle faibles (SDCF) *dans $G$ si tous les sommets atteignables depuis $V'$ sont des attracteurs faibles vers $V'$ dans $G$.*

Intuitivement, si $V'$ est SDCF, il n'existe aucun point de choix en dehors de $V'$ qui soit atteignable depuis $V'$ et qui puisse mener vers deux images différentes dans $V'$. Ce concept d'ensemble SDCF est effectivement adapté pour le slicing, car de tels points de choix hors de $V'$ pourraient permettre au programme initial de diverger vis-à-vis de sa slice. Dans l'exemple considéré, $\emptyset$ et $\{16\}$ sont SDCF parce qu'aucun sommet de $V'_0$ n'est atteignable en dehors de l'ensemble considéré. $\{0\}$ et $\{0, 16\}$ sont SDCF parce que 0 n'est atteignable que de 0. $V'_0$ n'est pas SDCF, mais si on l'étend en $V'_0 \cup \{1, 10\}$, on obtient un ensemble SDCF.

## 2.2 Reformulation

On désirerait un algorithme pour calculer le plus petit SDCF contenant un ensemble donné (i.e. sa clôture). Les ensembles SDCF possèdent une autre caractérisation qui justifie un tel algorithme.

**Définition 2.4** (Décideur faible, *weakly deciding vertex*)**.** *Un sommet $u$ est un* décideur faible *pour $V'$ dans $G$ s'il existe deux chemins non triviaux terminant dans $V'$, qui partent de $u$ et ne possèdent aucun autre sommet commun.*

Dans notre exemple, 10 est un décideur faible pour $V'_0$, les deux chemins étant $10 \to 11 \to 5$ et $10 \to 14 \to 16$. De même, 1 est un décideur faible pour $V'_0$ ($1 \to 2 \to 8 \to 16$ et $1 \to 10 \to 11 \to 5$). 2 n'est pas décideur faible pour $V'_0$ puisqu'il ne possède que 16 comme image dans $V'_0$.

**Propriété 2.1** (Caractérisation des SDCF)**.** *$V'$ est SDCF dans $G$ si et seulement si tous les décideurs faibles pour $V'$, qui sont atteignables depuis $V'$, se trouvent dans $V'$.*

### 2.3 Algorithme de Danicic pour le calcul de la clôture

Les décideurs faibles pour $V'$ atteignables depuis $V'$ sont les sommets à ajouter à $V'$ pour obtenir un ensemble SDCF dans $G$. Intuitivement, ils représentent en effet les points de choix les plus proches de $V'$ pouvant mener à deux images différentes dans $V'$. L'algorithme pour calculer la clôture proposé par Danicic et al [6] ne calcule pas ces éléments directement, mais propose d'ajouter itérativement des sommets dont la caractérisation est donnée ci-dessous.

**Définition 2.5** (Arête critique)**.** *On dit qu'une arête de $G$ est une* arête critique *pour $V'$ si son début a au moins deux images dans $V'$ et que sa fin a exactement une image dans $V'$.*

On montre que les débuts d'arêtes critiques pour $V'$ sont des décideurs faibles pour $V'$, et réciproquement que tant que $V'$ n'est pas SDCF, il existe une arête critique pour $V'$ dont le début est atteignable depuis $V'$. En revanche, à une itération donnée, tout décideur faible pour $V'$ n'est pas nécessairement le début d'une arête critique pour $V'$. Cela justifie l'algorithme itératif suivant.

**Algorithme 2.1** (Construction du plus petit ensemble SDCF)**.** *On commence par poser $W = V'$. Trouver une arête critique pour $W$ dont le début est atteignable depuis $W$, et ajouter son début à $W$. Répéter cette étape autant que possible. Lorsqu'il n'existe plus de telles arêtes, retourner $W$.*

Appliquons l'algorithme sur $W_0 = V'_0$. Ici tous les sommets sont atteignables depuis 0, qui est dans $W_0$, donc on peut ignorer la condition d'atteignabilité. D'après ci-dessus, (10,11) est une arête critique pour $W_0$, donc on pose $W_1 = W_0 \cup \{10\}$. (1,10) est une arête critique de $W_1$, puisque 10 n'a que 10 comme image, tandis que 1 a 10 et 16. Donc $W_2 = W_1 \cup \{1\}$. Il n'existe pas d'arête critique pour $W_2$, donc $W_2$ est le plus petit ensemble SDCF contenant $V'_0$. On peut noter que les éléments de $W_2 = \{0, 1, 5, 10, 16\}$ (coloriés dans la figure 2(b)) correspondent bien aux instructions préservées dans la slice structurée de la figure 1(b) (les dépendances de données n'ajoutent ici aucun sommet à la slice).

Nous avons fait la preuve de cet algorithme et la formalisation des notions sous-jacentes en Coq[1].

## 3 Algorithme optimisé

### 3.1 Description informelle

Une raison possible du manque d'efficacité de l'algorithme de Danicic est l'absence de partage d'information entre les différentes itérations de l'algorithme. Les images de chaque sommet sont recalculées à chaque itération puisque l'ensemble vis-à-vis duquel elles sont calculées change. Nous proposons une optimisation pour remédier en partie à ce problème : au lieu d'ajouter les nœuds un par un, il est possible à chaque itération de détecter toutes les arêtes critiques et d'ajouter leurs sources, ce qui permet de recalculer moins souvent les images des nœuds du graphe.

Mais nous proposons également d'aller plus loin et de réutiliser l'information obtenue sur les chemins du graphe dans les itérations ultérieures. L'idée principale de notre algorithme optimisé consiste à étiqueter chaque nœud $u$ par une image de $u$ dans l'ensemble résultat en construction $W$. Ces étiquettes survivent aux itérations et peuvent donc être réutilisées.

Contrairement à l'algorithme de Danicic, notre algorithme ne calcule pas directement le plus petit SDCF, en ce qu'il ne vérifie pas tout de suite l'atteignabilité des sommets qu'il accumule. Il est cependant facile d'éliminer les nœuds non-atteignables dans un second temps pour obtenir le SDCF.

Notre approche nécessite de manier les étiquettes avec précaution, de manière à ce qu'elles restent à jour et restent des images des nœuds qui les portent. En pratique, certains nœuds ne sont parfois pas à jour, mais cela n'empêche pas l'algorithme de fonctionner correctement.

Notre algorithme prend en entrée un graphe $G$ et un sous-ensemble de ses sommets $V'$. Il manipule trois objets : un ensemble $W$ égal à $V'$ initialement, qui grossit pendant l'algorithme et qui à la fin contient le résultat ; une table d'association *obs* qui associe à un sommet $u$ au plus un sommet dans $W$ atteignable depuis $u$ et qui est une image de $u$ dans $W$ à la fin ; une file $L$ de nœuds en attente de traitement et qui contient tous les nœuds de $V'$ initialement. Chaque itération se déroule comme suit. Un sommet $u$ est prélevé de $L$ si $L$ est non vide et tous les nœuds qui sont directement ou indirectement des prédécesseurs de $u$ non masqués par $W$ sont étiquetés par $u$. Après cette phase de propagation, de nouveaux décideurs faibles pour $V'$ sont détectés. Chacun de ces nœuds voit son étiquette mise à jour à lui-même et est ajouté dans $W$ et $L$. Si $L$ n'est pas vide, une nouvelle itération commence. Sinon, l'algorithme se termine et, après un filtrage des nœuds non atteignables depuis $V'$, renvoie dans $W$ le plus petit ensemble SDCF contenant $V'$ (i.e. la clôture de $V'$), ainsi qu'un étiquetage de chaque nœud par son image dans $W$, si elle existe.

### 3.2 Preuve formelle et évaluation expérimentale

Nous avons réalisé la preuve de l'algorithme optimisé dans Why3[1], afin de profiter de prouveurs automatiques. Cette preuve a nécessité d'identifier des invariants très subtils afin d'exprimer correctement les propriétés de l'étiquetage. Tous les lemmes nécessaires sauf un ont été formalisés et prouvés en Why3. Ce lemme non prouvé était toutefois déjà prouvé dans la formalisation Coq. Une preuve papier partielle est faite dans [10].

Nous avons également implémenté l'algorithme de Danicic et notre algorithme optimisé en OCaml à l'aide d'OCamlgraph. Pour nous assurer de leur correction, nous les avons testés sur une centaine d'exemples (prenant pour oracle l'implémentation certifiée, extraite grâce à la formalisation Coq et beaucoup plus lente). Ensuite, les deux versions ont été exécutées sur plusieurs milliers de graphes aléatoires générés par OCamlgraph, avec le nombre de noeuds variant entre 10 et 6500. Les résultats (cf. [10]) montrent que l'algorithme de Danicic explose pour les graphes avec quelques centaines de nœuds, alors que notre algorithme reste efficace sur des graphes de plusieurs milliers de nœuds.

## 4 Conclusion

Nous avons réalisé une formalisation en Coq d'une théorie de dépendances de contrôle faibles incluant la définition d'une notion de dépendance, d'une notion de stabilité correspondante et d'un algorithme pour le calcul de la clôture proposé par Danicic et al. La version actuelle de cette formalisation comprend plus de 8000 lignes de code Coq. Elle présente un intérêt à la fois théorique et pratique : une version certifiée peut servir d'oracle de tests. Nous avons également proposé un nouvel algorithme et réalisé sa preuve dans l'outil Why3. Enfin, nous avons réalisé une étude expérimentale qui montre un gain de performance important de notre technique par rapport à l'algorithme initial de Danicic. Dans le présent article, nous avons présenté nos motivations et quelques éléments clefs de ces contributions. Un article plus complet a été soumis [10]. Les formalisations complètes et les versions implémentées des deux algorithmes sont disponibles en ligne[1].

# Références


[1] Why3, a tool for deductive program verification, GNU LGPL 2.1. http://why3.lri.fr.

[2] Formalization of relaxed slicing, 2016. http://perso.ecp.fr/~lechenetjc/slicing/.

[3] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13) :1372–1386, 2010.

[4] José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *SEFM 2010*, 2010.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[6] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theor. Comput. Sci.*, 412(49) :6809–6842, 2011.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[8] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1) :45–64, 2003.

[9] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs : Sound verification on relaxed slices. In *FASE'16 (Part of ETAPS'16)*, pages 179–196, 2016.

[10] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Fast computation of arbitrary control dependencies. 2018. Submitted.

[11] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Coq a dit : fromage tranché ne peut cacher ses trous. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, 2016.

[12] The Coq Development Team. The Coq proof assistant, v8.6, 2017. http://coq.inria.fr/.

[13] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[14] Mark Weiser. Program slicing : Formal, psychological and pracical investigation of an automatic program abatraction method. phd dissertion. university of michigan. *Ann Arbor, Michigan*, 1979.

[15] Mark Weiser. Program slicing. In *ICSE 1981*, 1981.