Titre: Title:	C/C++ preprocessing directive analysis
Auteur: Author:	Ying Hu
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Hu, Y. (2001). C/C++ preprocessing directive analysis [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6954/

UNIVERSITÉ DE MONTRÉAL

C/C++ PREPROCESSING DIRECTIVE ANALYSIS

YING HU
GÉNIE ÉLECTRIQUE ET INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRE ÈS SCIENCES APPLIQUÉES (M.Sc.A.)
(GÉNIE INFORMATIQUE)
AVRIL 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

C/C++ PREPROCESSING DIRECTIVE ANALYSIS

présenté par: HU Ying

en vue de l'obtention du diplôme de: Maître ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. MERLO Ettore, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. GRANGER Louis, M.Sc.A, membre

DEDICATE

To my family

ACKNOWLEDGEMENTS

I would express my special gratitude to Professor Michel Dagenais, my supervisor, for his encouragement, patient guidance and valuable advice. Without his knowledge and experience, this thesis would not have been possible.

I would like to extend my sincere thanks to Professor Ettore Merlo, for his advice and technical guidance.

I would like to thank all of the enthusiastic and helpful friends in CASI laboratory, for making the past two years cherishable and memorable.

Finally, I would like to thank my family for their love and support.

RÉSUMÉ

La phase d'analyse de logiciels est une activité d'extrême importance lors d'un effort de développement ou de maintenance d'un programme. La prétraitement est la première phase de traduction du procédé de compilation qui en contient cinq: définition des constantes symboliques, inclusion des fichiers d'en-tête, expansion des macros, compilation conditionelle, et contrôle de ligne.

Dans les gros systèmes logiciels, l'inclusion des fichiers, la compilation conditionelle, et la substitution des macros sont directement reliées, et sont souvent largement interchangées. Plusieurs problèmes sont associés à l'utilisation du préprocesseur C et différents auteurs en ont décrit quelques-uns, habituellement d'un point de vue pragmatique. Plusieurs des problèmes discutés peuvent être attribués au manque d'outils de programmation associés à la manipulation et la maintenance de la logique de compilation conditionelle.

Ce mémoire a fait une revue des techniques relatives à la compréhension de programme et des efforts relatifs à la maintenance de logiciels en présence de plusieurs directives du préprocesseur. Dans cette formulation, nous avons appliqué les techniques de compilation, les techniques d'analyse du flux de contrôle et du flux de données, ainsi que la technique d'exécution symbolique pour identifier les parties du code qui peuvent être importantes pour une tâche particulière de maintenance.

Sous divers aspects, les résultats obtenus sont très encourageants. La majorité

de nos expérimentations ont illustré l'applicabilité et l'efficacité des techniques proposées en analyse de programmes des directives du préprocesseur.

Notre expérience avec le jeu d'outils sur les systèmes logiciels de grande taille (tel Linux) est qu'il est facile de fournir de l'information sur trois aspects du prétraitement.

ABSTRACT

Software analysis plays an important role in every software development or maintenance project. Preprocessing is the first phase of translation in compiling a program. The C programming language is intimately connected to its macro preprocessor, CPP. Programs that take advantage of preprocessing facilities incur an additional maintenance burden.

In large software systems, file inclusion, conditional compilation and macro substitution are closely related and are often largely interleaved. There are many problems associated with the use of C preprocessor. Many of the problems can be attributed to the lack of programming tools associated with manipulating and managing conditional compilation logic.

This thesis has reviewed program understanding techniques and efforts related to software maintenance in presence of numerous preprocessor directives. Within this framework, we have applied compiler techniques, control flow, data flow analysis, and symbolic execution techniques to identify parts of code that may be relevant to a particular maintenance task. We presented an approach and developed a tool to analyze the various usage of preprocessor directives.

Most of our experiments have illustrated the applicability and effectiveness of the proposed techniques in program analysis of preprocessor directives.

Our experience with the proposed tool-set on large software system (such as

Linux) is that it can easily provide information on three major aspects of preprocessing.

CONDENSÉ EN FRANÇAIS

I. INTRODUCTION

Comprendre, corriger et maintenir un logiciel est une tâche coûteuse et difficile. Les difficultés sont encore plus présentes dans un programme écrit pour prendre avantage des fonctions du préprocesseur.

La prétraitement, la première phase de traduction du procédé de compilation, en contient cinq: définition des constantes symboliques, inclusion des fichiers d'en-tête, expansion des macros, compilation conditionelle, et contrôle de ligne.

Plusieurs problèmes sont associés à l'utilisation du préprocesseur C et différents auteurs en ont décrit quelques-uns, habituellement d'un point de vue pragmatique.

Une macro-instruction ressemble à un appel de fonction mais n'agit pas toujours comme tel. Cela peut camoufler des défauts difficiles à trouver. Plusieurs erreurs ne sont pas détectées jusqu'à ce que plusieurs développeurs essaient d'intégrer leur code et se rendent compte qu'il ne compile pas, ne se lie pas, ou ne s'exécute pas correctement. Les erreurs de compilation conditionelle peuvent souvent être difficiles à tracer parce qu'elles sont cachées et requièrent une analyse de la sortie du préprocesseur pour trouver des indices afin de trouver ce qui ne va pas. Le préprocesseur (CPP) se permet aussi des manipulations arbitraires du code qui compliquent la compréhension du programme tant par les ingénieurs du logiciel que par les outils eux-mêmes. Les fichiers d'en-tête et leur relations forment

les hiérarchies. Comme toutes les autres parties d'un système logiciel, un fichier d'inclusion grossit avec le projet par l'ajout de fonctions, ou leur raffinement, et devient éventuellement gros et complexe. Dans les gros systèmes logiciels, l'inclusion des fichiers, la compilation conditionelle, et la substitution des macros sont directement reliés, et sont souvent interchangeables.

Plusieurs des problèmes discutés ci-dessus peuvent être attribués au manque d'outils de programmation associés à la manipulation et la maintenance de la logique de compilation conditionelle.

L'objectif de ce projet est de développer des outils qui permettront l'exploration et la compréhension du code en présence de nombreuses directives du préprocesseur.

Les outils de compréhension et d'exploration du programme obtiennent les données en analysant le code source. Nous devons donc nous assurer qu'une telle analyse peut extraire tous les détails concernant les directives reliées au préprocesseur.

Le premier but du projet est de créer un outil d'analyse CPP qui prend en considération toute la complexité de la grammaire du préprocesseur.

Le second but de ce projet est de créer un outil d'analyse des dépendances d'inclusion.

Le troisième but de ce projet est de créer un outil d'analyse des directives conditionelles du préprocesseur. L'utilité est ici de trouver toutes les conditions pour chaque version possible.

Le quatrième but de ce projet est de collecter et d'analyser les larges sys-

tèmes logiciels et d'améliorer les outils d'analyse en se basant sur les données expérimentales.

II. DESCRIPTION DE NOTRE APPROCHE

Une vue logicielle est une représentation d'un logiciel, ou un rapport à propos d'un logiciel. Une vue logicielle peut être humainement compréhensible ou non, mais typiquement c'est une représentation intermédiaire d'un logiciel que des humains pourraient vouloir consulter.

Dans ce projet, nous définissons quatre classes de vues de l'information des directives préprocesseur du logiciel.

Les directives de compilation conditionelle impliquent une expression booléenne suivie par du code source. Le code source est compilé seulement si l'expression booléenne qui lui est associé est évaluée à vrai durant le procédé de prétraitement. L'évaluation d'une condition dépend des valeurs assignées aux variables de compilation conditionelle.

Dans ce projet, l'analyse de la compilation conditionelle porte surtout sur l'extraction des conditions de compilation.

Concernant les conditions de compilation, il y a deux façons d'exprimer des conditions. La première est d'exprimer les conditions en tant que contraintes sur les valeurs initiales des variables du préprocesseur. La seconde est de les exprimer

en tant qu'expression associée aux valeurs finales des variables du préprocesseur juste avant l'évaluation des expressions conditionelles. Les deux techniques sont très utiles, la première particulièrement pour le programmeur qui réutilise le code, la seconde pour le programmeur qui le comprend.

Les expressions conditionelles sont utiles dans la connection avec les macros, parce que celles-ci sont le seul moyen où la valeur d'une expression varie d'une compilation à l'autre.

Les macros du préprocesseur C peuvent aussi bien être paramétrisées ou ne pas l'être. Les macros paramétrisées requièrent une liste d'arguments séparés par des virgules à l'intérieur d'une paire d'accolades, et sont plus souvent utilisées pour donner la notation concise d'un appel de fonction tout en retenant l'efficacité du code procédural. Les macros non-paramétrisées sont des remplacements textuels pour un identifiant unique et sont surtout utilisées pour les constantes symboliques. En se basant sur leurs différents usages, les identifiants macro peuvent être classés en trois catégories:

- définition de constante
- définition de fonction
- compilation conditionelle

Dans l'étude de la compilation conditionelle, les macros qui servent seulement en tant que définition de constante ou définition de fonction ne sont pas tellement

importantes puisqu'elles n'affecteront pas le contrôle de la compilation conditionnelle. Les plus importantes sont les macros qui apparaissent dans les directives de prétraitement conditionelles. Ces identifiants de macros sont appelés variables du préprocesseur. Un autre type de macro important est ce qui est indirectement utilisé par la compilation conditionelle. Les variables du préprocesseur ont plusieurs utilisations, l'une d'entre elles est un garde d'inclusion de fichier. Une autre utilisation des variables du préprocesseur est un selecteur de plate-forme. Ces identifiants de macros ne sont jamais définis dans les codes source.

Toutes les autres variables du préprocesseur sont définies en tant qu'autre catégorie de variables du préprocesseur.

Pour mieux comprendre comment les variables du préprocesseur sont utilisées, nous avons écrit des outils pour catégoriser l'usage des identifiants de macros et collecté des statistiques selon les différents usages.

Un fichier source peut inclure un certain nombre de fichiers d'en-tête (*.h*), qui eux-mêmes peuvent aussi contenir des fichiers d'en-tête. La séquence possible d'inclusions entre ces fichiers est exprimée dans un graphe de dépendances d'inclusion. Pour chaque fichier source, un graphe de dépendances d'inclusion peut donc être construit. Certains fichiers peuvent être inclus plus d'une fois, à travers des relations transitives.

Un graphe de dépendances d'inclusion (IDG) est un modèle de graphe abstrait des fichiers d'en-tête d'un code source système et de leur relations d'interdépendance.

Chaque noeud d'un IDG correspond à un fichier d'en-tête distinct, et une ligne dirigée entre deux noeuds, disons x et y, représente le fait que le fichier x dépend du fichier y.

Dans ce projet, nous ne générons pas seulement le graphe de dépendances d'inclusion mais nous donnons aussi la condition de chaque relation de dépendance. Les statistiques sommaires sont une bonne base objective pour la comparaison de différents programmes.

Afin de quantifier les attributs internes du logiciel, des mesures ont été définies sur le code, le graphe de flux de contrôle ainsi que sur le graphe de dépendances d'inclusion.

L'ensemble des mesures choisies inclut: les mesures de code, les mesures sur le graphe, les mesures de réutilisation, et les mesures de complexité.

III. TECHNIQUES ET IMPLANTATIONS

Le système d'outils consiste en quatre phases qui concernent les techniques d'analyse, les techniques de traverse d'arbres abstraits de syntaxe, les techniques d'exécution symbolique, et les techniques d'analyse des graphes du flux de contrôle. L'information sur les multiples vues logicielles est extraite et mémorisée dans la base d'information pour des analyses externes futures.

Afin d'extraire l'information, nous devons premièrement représenter le code à

un niveau d'abstraction plus élevé. L'analyse commence par l'abstraction du code source dans un arbre de syntaxe abstraite (Abstract Syntax Tree, AST). Le AST est une représentation sous forme d'arbre des jetons contenus dans le code source. Il offre une représentation exacte du code source.

Un arbre de syntaxe démontre la structure hiérarchique naturelle d'un programme source. La prochaine étape s'associe avec l'interprétation sémantique et l'analyse des dépendances. À ce point, certains modèles et formats intermédiaires sont définis, ceux-ci sont spécifiés de la même façon pour tous les languages source pour permettre aux outils d'analyse de venir à ne pas se préoccuper des variations concernant le language.

Le procédé d'extraction de l'information intermédiaire correspond à une traversée de l'arbre de syntaxe abstraite qui commence à la racine, visite un noeud avant son enfant, et visite récursivement les enfants à chaque noeud de gauche à droite. La méthodologie utilisée ici sert à développer les visiteurs correspondants.

Dans ce projet nous allons utiliser l'arbre dominant pour permettre d'identifier certaines conditions nécessaires pour compiler n'importe quelle ligne de code. Le flux de contrôle dans le préprocesseur est très simple, il ne contient en fait qu'une seule branche. Le noeud conditionnel est le point de décision. Seul un noeud conditionel peut avoir plus d'un enfant dans l'arbre dominant, les autres noeuds n'ont qu'un seul enfant direct.

L'exécution symbolique, aussi appelée évaluation symbolique, diffère de l'exécution

des programmes au sens traditionnel. La notion traditionnelle d'exécution requiert qu'une sélection de chemins à travers le programme soit effectuée par un ensemble de cas de test. En exécution symbolique, les valeurs prisées par les données sont remplacées par des valeurs symboliques. L'exécution traditionnelle de programmes utilise des entrées constituées de valeurs. L'exécution symbolique, quant à elle, produit un ensemble d'expressions à concurrence d'une expression par variable de sortie.

L'utilisation la plus courante de l'exécution symbolique est d'effectuer une analyse du programme, conduisant à créer un graphe de flux, qui est un graphe dirigé contenant des points de décision et les affectations associées à chaque branche. En traversant le graphe de flux le long d'un chemin particulier, à partir d'un point d'entrée, une liste d'instructions d'affectation et de prédicats est produite.

A la fin de l'exécution symbolique d'un chemin, la variable de sortie sera représentée par des expressions en terme de valeurs symboliques des variables d'entrée, et des constantes. Les expressions de sorties sont sujettes à des contraintes.

L'exécution symbolique peut être appliquée aux directives du preprocesseur en vue d'analyser la compilation conditionnelle.

L'utilisation effective des valeurs symboliques requiert la capacité de simplification des expressions symboliques et le support pour l'éclatement d'un chemin d'exécution en plusieurs chemins quand des valeurs symboliques booléennes sont

rencontrées dans des instructions symboliques.

L'exécution symbolique est basée sur le graphe de flux de contrôle (CFG) des directives du préprocesseur et du graphe d'instance de fichier. La procédure est gérée par une pile de noeuds actifs qui est une queue de priorité de noeuds de type CFG.

Pour trouver une condition suffisante pour compiler une ligne de code donnée, nous utilisons un algorithme de recherche en largeur d'abord (BFS). Pour trouver toutes les conditons du chemin pour compiler une ligne de code donnée, nous utilisons un algorithme de recherche en profondeur d'abord (DFS).

IV. RÉSULTATS EXPÉRIMENTAUX

Nous avons appliqué notre outil à certains systèmes logiciels réels de grande taille. Les expérimentations se sont déroulées sur un Pentium III 500 MHz avec 512MB RAM tournant le système d'exploitation Linux.

IV.1 Les métriques du code

Nous collectons d'abord des métriques pour chaque sous-système du noyau et chaque sous-système du répertoire d'inclusion, et nous utilisons ensuite les statistiques de chaque sous-système pour avoir une bonne compréhension de tout le système.

A partir de la comparaison, on note que la version du noyau Linux-2.2.12 a plus de fichiers d'interface que celle du noyau Linux-2.2.3, ce qui peut être dû à l'augmentation de la capacité du système.

IV.2 Analyse de l'identificateur du préprocesseur

Pour permettre une bonne compréhension de comment le préprocesseur est utilisé, nous lançons nos outils pour analyser l'utilisation de la variable du préprocesseur.

En étudiant les directives du préprocesseur, nous sommes plus intéressés par celles qui affectent le flux de contrôle de la compilation. Les plus importantes sont les macros qui apparaissent dans les directives de prétraitement conditionnel. Ces identificateurs de macros sont appelés variables du préprocesseur. Un autre type important de macros sont celles indirectement utilisées par la compilation conditionnelle.

La classification des identificateurs de macros est très importante, elle peut être utilisée pour réduire l'espace des variables de la phase d'exécution symbolique.

Il est intéressant de savoir quelles et combien de variables du préprocesseur ont été utilisées comme des gardes de l'inclusion de fichiers. Cette information peut devenir le point de départ pour reconstruire le système d'inclusion de fichiers en une plus simple organisation.

IV.3 Graphe de la hiérarchie d'inclusion

Le graphe de la hiérarchie d'inclusion a été généré pour chaque fichier du noyau de Linux. Les vues graphiques fournissent aux usagers une information structurale importante qui est généralement cachée dans le code du programme. La représentation graphique des relations des répertoires peut être construite sur la base des statistiques des directives d'inclusion.

Les graphes de dépendance d'inclusion peuvent contenir des cycles. Dans le contexte d'inclusion de fichiers, la détection de tels cycles est importante. Ils peuvent signifier une forte relation de couplage entre ces fichiers. Les algorithmes de la théorie des graphes ont été utilisés pour détecter les cycles dans le graphe.

IV.4 Le graphe de flux de contrôle

Un graphe de flux de contrôle (CFG) est constitué de noeuds représentant une entrée simple, une sortie simple, des régions du code exécutable et des arcs qui représentent des branches d'exécution possible entre les régions du code. Un CFG pour les directives de prétraitement fournit une illustration graphique de l'utilisation et de la complexité de la compilation conditionnelle dans un programme.

Premièrement, un graphe de flux de contrôle complet a été généré pour le code source.

Le graphe de flux de contrôle complet des directives du préprocesseur garde toutes les informations du code, ce qui est de loin plus que ce dont nous avons

besoin pour faire l'analyse de la compilation conditionnelle. Ceci réduit de façon substancielle la performance de la procédure d'analyse. Dans certaines situations, par exemple dans l'analyse du code en présence des inclusions de fichiers, un graphe de flux de contrôle réduit peut être suffisant pour le fichier inclus.

Pour chaque graphe de flux de contrôle, un graphe de flux de contrôle réduit correspondant a été généré.

IV.5 Exécution symbolique des directives de prétraitement

L'exécuteur symbolique traverse les noeuds du CFG stockés dans la pile de noeuds. Chaque noeud a une table de constantes associées à une macro et une contrainte de la condition courante attachée. La pile de noeuds est initialisée avec le noeud de départ du CFG en cours de test. L'exécution symbolique procède pour chaque noeud retiré de la pile de noeuds, met à jour constamment les modifications apportées à l'unité testée de la table de la macro, annote le chemin correspondant avec les contraintes appropriées rencontrées au points de branchement le long des chemins, et étend la pile de noeuds avec le prochain noeud éventuel dans le CFG selon le sélecteur de chemins.

Plus court chemin à un noeud donné

Pour trouver le plus court chemin à un noeud donné, la sortie dépend de chaque étape d'exécution, l'information complète du flux de contrôle est alors requise. Avec cette stratégie, les graphes de flux de contrôle complets sont utilisés pour tous les

fichiers dans la hiérarchie d'inclusion. Deux exemples sont donnés au chapitre 5.

La condition résultante de la stratégie du plus court chemin est une expression booléenne du genre "et-logique" avec des valeurs symboliques, chaque sous-expression étant une constante.

Condition suffisante à un noeud donné

Pour trouver la condition suffisante à un noeud donné, nous utilisons la stratégie de recherche en profondeur. Chaque étape en exécution n'est pas importante en soi, mais la condition finale est celle qui est souhaitée. Pour le fichier contenant le noeud cible, le graphe de flux de contrôle complet est utilisé. Pour tous les autres fichiers dans le graphe de la hiérarchie d'inclusion, le graphe de flux de contrôle complet n'est pas nécessaire, celui réduit suffit. Deux exemples sont donnés au chapitre 5. La condition complète était alors soumise à un simplificateur de condition de chemin. Après simplification, la contrainte finale était fournie. Le but de trouver la condition suffisante pour rejoindre une ligne de code testée est de couvrir toutes les combinaisons de conditions, ce qui prend bien sûr trop de temps. La performance peut être largement améliorée en utilisant le CFG réduit. Un autre moyen d'adapter la performance serait l'initialisation de quelques variables du préprocesseur au début de l'exécution symbolique, en se basant sur la connaissance que le programmeur a des programmes.

IV.6 Pré-condition

La condition de chemin collectée à partir de l'exécuteur symbolique donne les contraintes sur les valeurs initiales. Ici nous utilisons l'arbre de dominateur de la technique d'analyse du flux de données et construisons un outil qui peut facilement collecter les requis de la valeur finale des variables. Le noeud au sommet est le noeud de départ d'un fichier donné. Chaque noeud du CFG correspond à une ligne de code. Tous les noeuds qui ont une même condition de compilation sont groupés ensemble et mis dans une même boite. La condition est décorée sur l'arc dirigé à partir du noeud racine vers chaque boîte étiquetée C_n (figure 5.16). Une condition vide (l'arc le plus à gauche) renvoie toujours à une condition vraie.

Comparant ce résultat pour kernel.h avec celui obtenu de l'exécuteur symbolique, nous pouvons trouver que ces résultats sont similaires, puisque dans cet exemple concret, il n'y a pas de redéfinition de variables le long du chemin en exécution.

Dans ce chapitre, nous choisissons le noyau de linux comme cas d'étude. Nous avons lancé notre outil et avons collecté beaucoup d'informations intéressantes sur les directives du préprocesseur. Comme nous le savons, dans un grand système, l'inclusion de fichiers, la compilation conditionnelle, et la substitution de macros sont intimément liées et sont souvent largement imbriquées. Dans les exemples que nous avons donnés dans ce chapitre, nous démontrons comment extraitre et contruire les représentations textuelle et graphique de ces trois principaux composants des directives du préprocesseur.

V. CONCLUSION

Ce mémoire a fait une revue des techniques relatives à la compréhension de programme et des efforts relatifs à la maintenance de logiciels en présence de plusieurs directives du préprocesseur. Dans cette formulation, nous avons appliqué les techniques de compilation, les techniques d'analyse du flux de contrôle et du flux de données, ainsi que la technique d'exécution symbolique pour identifier les parties du code qui peuvent être importantes pour une tâche particulière de maintenance.

V.1 Contributions

Cette section résume la contribution majeure de cette thèse.

Premièrement, nous utilisons JavaCC et jjtree pour contruire un analyseur syntaxique de CPP qui prend en compte la complexité totale de la grammaire du préprocesseur.

Deuxièmement, nous avons présenté une approche et avons développé un outil pour analyser l'usage varié des variables du préprocesseur.

Troisièment, nous avons concu un outil d'analyse des dépendances d'inclusion.

Quatrièmement, nous avons présenté une approche et conçu un outil pour les directives conditionnelles du préprocesseur.

Nous avons aussi défini des séries de métriques sur le code, les variables, les composantes de fichiers et les répertoires. La collection et l'analyse de telles métriques

fournit une approximation rapide de l'analyse des dépendances.

V.2 Discussion et travaux futurs

Sous divers aspects, les résultats reportés sont très encourageants. La majorité de nos expérimentations ont illustré l'applicabilité et l'efficacité des techniques proposées en analyse de programmes des directives du préprocesseur.

Dans ce mémoire , nous avons essayé d'intégrer quelques techniques pour fournir des méchanismes d'analyse des directives du préprocesseur.

L'approche d'appariement des patrons est utilisée pour classifier l'utilisation variée des variables de macros.

Les techniques de graphe de flux de contrôle et de graphe de flux de données sont efficaces pour obtenir la condition de compilation représentée en termes de valeurs initiales des variables de macros.

Avec l'approche basée sur les métriques, nous avons essayé de représenter la structure et les dépendances entre les composants logiciels de haut niveau.

Notre expérience avec le jeu d'outils sur les systèmes logiciels de grande taille (tel Linux) est qu'il est facile de fournir de l'information sur trois aspects du prétraitement.

Deux principaux sujets de recherche futures sont:

- accroître l'efficacité de l'exécuteur symbolique pour grands systèmes.
- ajouter quelque options de visualisation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Understanding, debugging, and maintaining software is a costly and difficult task. The difficulties are exacerbated in programs written to take advantage of source code preprocessing facilities [31].

Preprocessing is the first phase of translation in compiling a program. The C programming language [23] is intimately connected to its macro preprocessor, CPP [18]. CPP supplies five separate facilities: defining symbolic constants, inclusion of header files, macro expansion, conditional compilation, and line control.

C and C++ software systems typically share global type definitions, function declarations, inline function definitions, data declarations, constant definitions, enumerations, and macros by including common header files.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the need for finding and changing

all the copies as well as the risk that a failure to update one copy will result in inconsistencies within a program.

Conditional compilation is a preprocessor feature that provides text substitution, macro expansion and file inclusion. Jaeschke, a member of the ANSI C and C++ standards committees, states that conditional compilation is "one of the most powerful parts of a C environment available for writing code that is to run on different target systems" [20]. Conditional compilation permits a single source file to generate different programs. This is useful when a program is targeted to multiple platforms.

CPP is a useful and often necessary adjunct to C, for it provides capabilities unavailable in the language or its implementations. CPP permits definition of portable language extensions that can define new syntax, abbreviate repetitive or complicated constructs, eliminate reliance on a compiler implementation to inline functions, propagate symbolic constants, or eliminate dead code and short-circuit constants tests. The latter guarantees are especially valuable for compilers that do a poor job optimizing or when the programmer wishes to override the compiler's heuristics. CPP also permits system dependencies to be made explicit and tested, resulting in a clearer separation of concerns. Finally, CPP permits a single source to contain multiple different dialects of C; a frequent use is to support both K&R-style and ANSI-style declarations.

Programs that take advantage of preprocessing facilities incur an additional

maintenance burden. Typically, the maintainer interacts with source code; however, the compiled program is based on the preprocessed code. Thus preprocessor commands can be a two-edged sword: they can increase code readability and programmer productivity while simultaneously obfuscating the program's mechanics and, consequently, maintainability.

There are many problems associated with the use of C preprocessor and different authors have described some of them, usually from a specific or pragmatic point of view [8,11,12,31,35,41,42].

A macro looks like a function call, but doesn't always act like one. This can bury difficult-to-find bugs [8]. Conditional compilation programming errors are easy to make and difficult to detect. Many errors go undetected until multiple developers attempt to integrate their code, and find that it does not compile, link or execute correctly on some target platforms. Conditional compilation errors can often be difficult to track down because they are hidden, and require analyzing the preprocessor output for clues as to what has gone wrong. CPP also lends itself to arbitrary source code manipulations that complicate understanding of the program by both software engineers and tools. The header files and their interdependencies form include hierarchies. As with any other parts of a software system, an include file hierarchy grows with a project as features are added, or refined, and eventually becomes large and complex. When an include file hierarchy is sufficiently complex, it is hard for programmers to find out exactly why a file must be included. Since

including a file that does not contain useful information is usually harmless, the tendency is to include enough files so that the code will compile. For large projects, this practice may even be institutionalized by providing global header files that simply include the world. This practice simplifies programming at the extra cost of compilation overhead due to the processing of unneeded include files.

In large software systems, file inclusion, conditional compilation and macro substitution are closely related and are often largely interleaved.

The designer of C++, which shares C's preprocessor, also noted these problems: "Occasionally, even the most extreme uses of CPP are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and tool builders." [44]

Many of the problems discussed above can be attributed to the lack of programming tools associated with manipulating and managing conditional compilation logic. Other than the actual preprocessor itself, many programming tools, such as debuggers and programming editors, are not able to process the conditional compilation logic and keep track of the individual threads of compilation and screen out sections of code that aren't being compiled.

Tools and, to a lesser degree, software engineers have three options for coping with CPP. They may ignore preprocessor directives (including macro definitions) altogether, accept only post-processed code (usually by running CPP on their input), or attempt to emulate the preprocessor.

Ignoring preprocessor directives is an option for approximate tools (such as those based on lexical or approximate parsing techniques), but accurate information about function extents, scope nesting, declared variables and functions, and other aspects of a program requires addressing the preprocessor.

Operating on post-processed code, the most common strategy, is simple to implement, but then the tool's input differs from what the programmer sees, information is lost in the mapping back to the original source code [33,37,49], which is an undesirable and error-prone situation. A tool supplied with only one post-processed instantiation of the source code cannot reason about the program as a whole, only about that version that results from one particular set of preprocessor variables.

The third option, emulating the preprocessor, is fraught with difficulty. Macro definitions consist of complete tokens but need not be complete expressions or statements. Conditional compilation and alternative macro definitions lead to very different results from a single original program text. In many situations, some sort of preprocessing or CPP analysis can produce useful answers. This is the focal point of this project, to which tools were developed to analyze preprocessor directives.

1.2 Research Goals

The objective of this project is to develop facilities that will allow code exploration and understanding, in the presence of numerous preprocessing directives.

Program understanding and exploration tools obtain the data by parsing source code. We must therefore ensure that such parsing can extract every detail of preprocessor directives. The first step of the project is to design a CPP parser, which takes into account the full complexity of the preprocessor grammar.

The second step of this project is to design an analysis tool for include dependencies. This includes: (1)displaying the dependencies among include files in graphical forms, (2)inferring the conditions of such dependence relationships, (3)providing ways to group files and refine the structure of hierarchies.

The third step of this project is to design an analysis tool for conditional preprocessor directives. When conditional compilation is used in a program, the actual file compiled is only one of many possible versions of the source code, each determined by a particular setting of conditional compilation variables. The compiler does not care about the parts of the code that are not included in the particular compilation. However, program comprehension and reverse engineering are concerned with the understanding of all the information in a source code. Effective tools should therefore be able to show all entities in a system along with information about which states of conditional compilation variables permit that entity to be considered by the compiler. The purpose here is to find all the conditions for each possible version.

The fourth step of this project is to collect and analyze large software systems and improve the analysis tools based on the experimental data.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 surveys related works.

Chapter 3 defines the information to be collected and analyzed.

Chapter 4 describes the techniques used in the project and some details of implementation.

Chapter 5 presents experiment results of a case study on Linux kernel.

Our conclusions and future work are discussed in Chapter 6.

CHAPTER 2

LITERATURE REVIEW

C/C++ preprocessor usually features mechanisms to include files, to conditionally include a block of code, and to perform macro substitutions. We could not find other study dedicated to C/C++ preprocessor and covering all the above three aspects. However, we did find some research work on other preprocessors or concentrated on one individual feature of C preprocessor.

2.1 Conditional Compilation Related Work

Conditional compilation is not restricted to C and C++, its use in other languages such as FORTRAN and COBOL has increased to the point where new standards have been proposed which include conditional compilation [2,19]. Furthermore, the power of this technique has resulted in published examples of using conditional selection in Ada and Java, and has prompted some Pascal and BASIC compiler vendors to retrofit conditional compilation capabilities into their products [14,40].

Much related work has been performed over the past ten years in the area of conditional compilation [10,12,13,35,38,39].

Some et al [39] discussed difficulties when parsing code with conditional compila-

tion. They analyzed software systems written in a proprietary language called Mitel Pascal with extensive use of conditional compilation. Their research objective is to parse source files to extract all the information. Their basic assumption is that an invalid program cannot be completely parsed, one only wants to consider variants of the source code that are syntactically correct. The straightforward way is 1) to find all the combinations of conditional compilation variables and their possible values, and 2) to parse the source code repeatedly with each of these combinations. This will guarantee that each section of the source code be parsed; although many sections will clearly be parsed many times. The problem with this exhaustive parsing is that the time and resources needed to parse large software systems may be quite significant. They proposed an optimized multi-parsing algorithm based on the following fact: 1) The objective is to extract all the information from code files. Therefore situations where conditional compilation directives are skipped are not pertinent unless not doing a such skipping induces errors. 2) Some directives may be independent enough to be parsed together even if their conditions cannot be logically true at the same moment. 3) Erroneous situations can be skipped once detected. The principle of this optimization is to analyze as many conditional compilation directives as possible during a single parse. Starting with the set of all the conditions in a source file, the idea is to build subsets of syntactically compatible conditions using the dependency and the inconsistency relationships. The approach is based on heuristics. Tests show that there is a gain with the optimized

approach. However the optimized approach may fail for certain source files with a particular use of conditional compilation directives.

Pearse et al [35] discussed their experiences using conditional compilation, within the multi platform parallel development model, to create portable, scalable software systems and described a tool that was developed to help understand code containing conditional compilation. They showed the conditional compilation structure of a source file by generating a graphical representation. They also computed some metrics (lines of input source code, lines of conditional compilation logic, number of conditional logic branches and number of preprocessor variables) in order to evaluate conditional compilation complexity. They suggest that the problems of measuring and managing conditional compilation complexity are parallel to those of measuring and managing traditional source code complexity. That is, the complexity of the code is influenced by such characteristics as the number and nesting of conditional compilation statements, the number of conditional compilation branches, the complexity of the logical expressions used in those branches, and the number and span of preprocessor variables used in the logical expressions. However, as mentioned in this article, measuring the complexity of conditional compilation is not well defined yet, more research and better measures are needed to quantify conditional compilation. To help manage conditional compilation, they compiled a list of coding standards and guidelines from existing literature which describes the problem and recommended guideline.

J.M. Favre [12,13] presented an approach to the problem with conditional compilation by introducing APP, an abstract language semantically equivalent to CPP but based on traditional programming-in-the-small concepts. The use of abstractions considerably increases problem comprehension and solutions can be derived directly from existing techniques like slicing, program specialization or interprocedural data flow analysis. He argued that rigorous description of the semantics of this language makes it possible to develop reliable reverse engineering tools

D. Epstein [10] discussed benefits of Fortran 90 Conditional Compilation Facility (CCF) which avoids problems with CPP conditional compilation. CCF is a line-based language that can easily be adapted to other programming languages.

Snelting's approach [38] involved rediscovering code configuration structure by computing a concept lattice based on conditional compilation directives. The structure found may then be used to assess the code structure and help reorganizing it.

2.2 Include Dependency Related Work

The exploration of relationships among include files is an example of software reverse engineering. Such kind of research is of particular interest to big telecommunication companies.

Vo et al [48] from AT&T Bell Laboratories presented a tool, Incl, which could be used to generate textual or graphical representations of relationships among include

files. One basic assumption is that the generated code will remain the same with or without unused declarations which cannot influence program behavior. The objecti ve is to detect and ignore include files that consist only of unused declarations. Determining when files are needed for compilation requires knowledge of reference relationships among symbols. Their C information abstractor creates a C program database from C source files that stores the reference relationships between all global program objects (types, macros, functions, variables, and files). Based on the extracted reference relationships, they described a set of conditions under which include files could be safely ignored during compilation and implemented a linear time algorithm to compute such files. Eliminating unused include files can save compile time. Deleting include files is inherently dangerous because header files are usually shared. Incl is supposed to be used in conjunction with a smart C preprocessor to ignore unused include files during compilation. This approach is different from the standard practice of using #ifndef _FOO to avoid multiply included files.

2.3 Macro Related Work

Macro related work is concentrated on analysis of macro expansion and substitution. Such work is based on token analysis of macro definition and/or of user code that invokes those macros.

In [31], Livadas et al examined problems associated with source code containing

preprocessor constructs. They defined some useful mappings from tokens in the preprocessor's output to the source file(s), and proposed that by capturing these correspondences, an internal program representation can be built which allows for the use of maintenance techniques including program slicing, ripple analysis and dicing. They discussed ANSI C preprocessor macro substitution and explained the modus developed to handle them in GHINSU, an integrated maintenance environment for ANSI C programs.

Another research effort on macro analysis is the categorization of macro definition bodies [9,35].

In [9], Ernst et al developed a framework for analyzing the purpose of macros. Their focus is on macros as definition and their uses. They gave a report on their analysis of 27 packages comprising 1.2 million lines of publicly available C code, determining how the preprocessor is used in practice. Extracted information includes the percentage of original C source code lines that are preprocessor directives, and how often each macro is defined and expanded. In general identifiers are defined relatively few times. They were particularly interested in determining the frequency of use of macros that are difficult to convert to other language features, such as those that string together characters as opposed to manipulating lexeme or syntactic units (less than one third of one percent of all macro definitions), those that expand to partial syntactic units such as unbalanced braces or partial declarations (half of one percent), and others not directly expressible in the programming lan-

guage (about four percent). While in [35], Pearse et al outlined patterns of macros as preprocessor variable used in conditional compilation directives.

2.4 Summary

In this chapter, a number of interesting articles on conditional compilation, include dependency and macro analysis are presented. From the survey, we found that: 1) there is a strong need for better programming tools to help understand preprocessor logic. 2) the research has not been accomplished yet, more research are needed. 3) the problems involved in preprocessor directives are relevant to software development and maintenance in all industrial programming languages. We can take advantage of existing techniques. Some techniques such as parsing techniques, static analysis techniques presented in these literatures are found to be very useful for the analysis of C/C++ preprocessor directives. We extend the research by taking advantage of other techniques such as symbolic execution and control flow analysis. This is reflected on the extraction of include dependencies with the condition of compilation expressed by the preprocessor variables.

CHAPTER 3

DESCRIPTION OF OUR APPROACH

A software view is a representation of software or a report about software. A software view may be for human viewing or not, but it typically is a significant interim representation of software that humans may want to see. View information means the specific information in a view, or the information base of knowledge decomposed from information in the view [3].

Examples of software views are specifications, source code, measurements, reports derived from static source code analysis, and test data used to characterize software behavior.

The information base is the repository of information about the software. In this chapter, we define software view information about preprocessing directives.

Preprocessing directives are lines in programs that start with '#'. The '#' is followed by the identifier for the directive name. Table 3.1 lists preprocessing directives.

The preprocessor directives can be categorized into four groups: conditional compilation directives, file inclusion, macro definition and "other" directives for the rest. In general, the "other" directives are rarely used. In this project, the analysis is focused on the first three groups. In the following we outline all the

Table 3.1: Preprocessor directives

#assert	#cpu	#define	#elif
#else	#error	#ident	#if
#ifdef	#ifndef	#import	#include
#include_next	#line	#machine	#pragma
#pragma once	#system	#unassert	#undef
#warning			

Table 3.2: ANSI C conditional compilation directives

directive	code is selected under condition
#ifdef VARIABLE	VARIABLE is defined
#ifndef VARIABLE	VARIABLE is not defined
#if EXPRESSION	EXPRESSION non-zero
#elif EXPRESSION	EXPRESSION non-zero
#else	matching #ifdef, #ifndef, #if statement is not selected
#endif	end of conditional compilation

information and data to be extracted from the code.

3.1 Compilation Condition

ANSI C supports six different conditional compilation directives, which are described in Table 3.2. Conditional compilation directives test for the existence or values of preprocessor symbolic constants. By changing the definition of the preprocessor variables in conditional directives, one can change the way the code is preprocessed.

A conditional compilation directive involves a boolean expression followed by some source code. The source code is compiled only if its associated boolean

expression evaluates to true during a pre-processing process. The evaluation of a condition depends on values assigned to conditional compilation variables.

In this project, the analysis of conditional compilation is emphasized on extraction of compilation conditions. There are two ways to express conditions. One is to express them as the constraints on the initial values of preprocessor variables. The other is to express them as the expression associated with the final values of preprocessor variables right before the evaluation of the condition expression. Both kinds of conditions are very useful, the first one is helpful for programmers to reuse the code, the second one is helpful for programmers to understand the code.

```
#ifdef __FIRST__
#define __SECOND__ 1
#else
#define __SECOND__ 2
#endif
#if __SECOND__ == 2
  code segment 1
#else
  code segment 2
#endif
```

For example, in the above code, the condition under which code segment 1 is compiled, expressed as the constraints on initial value of preprocessor variable,

should be:

```
! defined __FIRST__
```

Expressed as the constraints on the final value of preprocessor variable right before the evaluation, it should be:

```
__SECOND__ == 2
```

These are two completely different ways to think about the condition. The first one gives the requirement of initial values, what the programmer should put into the makefile to compile that code. The second one gives the final constraints on variables, which is the intention of the original programmer to ask such code to be compiled. One is the real condition, another is the intended condition.

Based on the above abstraction, we define three concrete goals to achieve, which are: for any given preprocessor directive or C/C++ source code line,

- quickly finding one sufficient condition to reach/compile it
- finding the full condition to reach/compile that code line
- finding the complete intended condition

3.2 Macro Identifier Classification

Conditional directives are useful in connection with macros, because those are the only ways that an expression's value can vary from one compilation to another.

Macros are created using the *#define* preprocessor directive in C source files or in the header files, or as an option on the compiler command line. When a macro identifier is used in a program, it is replaced with the macro body during preprocessing. Macros may also be undefined, using preprocessor directive *#undef*.

C's preprocessor macros can be either parameterized or non-parameterized. Parameterized macros require a list of comma-separated arguments inside a pair of brackets, and are most commonly used to give the concise notation of a function call, whilst retaining the efficiency of in-line code. Non-parameterized macros are textual replacements for a single identifier, and are most commonly used for symbolic constants. According to their different usage, macros identifiers can be put into three categories:

- constant definition
- function definition
- conditional compilation

In the study of conditional compilation, macros which only serve as constant definition or function definition are not very important, since they will not effect the control flow of compilation. Most important are those macros which appear in the conditional preprocessing directives. These macro identifiers are called preprocessor variables. Another important type of macros are those indirectly used by conditional compilation, an example is shown in figure 3.1.

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#if TABLESIZE > 1024
     cpp context
#endif
```

Figure 3.1: Macro identifier indirectly used by conditional directive

Although *BUFSIZE* does not appear in a conditional directive, it appears in the body of a macro which is used by conditional directives. *BUFSIZE* is called an indirect preprocessor variable. Also, if a macro identifier occurs in the body of another indirect preprocessor variable (macro), it is an indirect preprocessor variable.

Preprocessor variables have many uses, one of which is as file inclusion guarder. Very often, one header file includes another, it can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, programmers often wish to prevent multiple inclusion of a header file. To prevent the content of an include file from being included twice, the header file is wrapped with conditional logic to test for a previous inclusion.

The standard way to do this is to enclose the entire content of the file in a conditional, like in figure 3.2.

The macro *_LINUX_CONFIG_H* indicates that the file has been included once already. When this file is scanned for the first time by the preprocessor, the

```
#ifndef _LINUX_CONFIG_H
#define _LINUX_CONFIG_H
#include <linux/autoconf.h>
#endif
```

Figure 3.2: Use pattern of preprocessor variable as safe guarder

symbol *_LINUX_CONFIG_H* is not yet defined. The *#ifndef* condition succeeds and *#include* are scanned. In addition, the symbol *_LINUX_CONFIG_H* is defined.

When this file is scanned for a second time during the same compilation, the symbol *_LINUX_CONFIG_H* is defined. All information between the *#ifndef* and *#endif* directives is skipped.

The symbol name *_LINUX_CONFIG_H* serves in this context only for recognition purposes. Preprocessor variable *_LINUX_CONFIG_H* is called a safe guarder. To extract such usage of preprocessor variables, a pattern extractor was implemented. By comparing the terminal and nonterminal, one can identify the use of safe guarder pattern. Usually, the preprocessor variables used as safe guarder are not defined at the beginning of the compilation.

Another usage of preprocessor variables is as platform selector. Some macros are predefined on each kind of machine. For example, on a Vax, the name 'vax' is a predefined macro. On other machines, it would not be defined. Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros

with conditionals to avoid using a system feature on a machine where it is not implemented. Macros are a common way of allowing users to customize a program for different machines or applications. Macros can be defined or undefined with '-D' and '-U' command options when one compiles the program. All those predefined macros and those variables need to be defined in a configuration file or a makefile are categorized as platform selector. These macro identifiers are never defined in the source code.

All the other preprocessor variables are classified in the "other preprocessor variable" category.

To gain a better understanding of how preprocessor variables are used, we wrote tools to categorize the usage of macro identifiers and collect statistics of different usages.

3.3 Include Dependency

A source file may include a number of header files (.h), which themselves also include other header files. The possible sequence of includes between these files is depicted in an include dependency graph. For each source file, a file dependency graph can thus be built. Some files can be included more than once through transitive relationships.

Definition : A include graph is a triple $G = (V, E, r)$, where V is the set of files including an initial file r, each node of an include graph corresponds to a distinct

header file. Let c be a vertex of the control flow graph of u that contains an include directive to v. Then E is the set of all directed edges between two nodes $(u \rightarrow v, c)$.

Files can be further distinguished by file instances. A instance depends on the include sequence. The file instances of an include graph are defined below:

Definition : Let $G(V, E)$ be a include graph where V is the set of files including an initial file and E is a set of pairs (u, v). The edge $e = u \rightarrow v$ denotes include on v within file u. The vertex c is a vertex of the control flow graph of u that contains an include directive to v. Then, the set of file instance is defined recursively:

- the initial file has a single instance
- let $(u \rightarrow v, c) \in E$ and u_i be an instance of u, then, v_{c,u_i} is an instance of v

An include dependency graph(IDG) is an abstract graph model of the header files of a system source code and their depends-on relationships. Each node of an IDG corresponds to a distinct header file, and a directed edge between two nodes, say x and y, represents the fact that file x depends on (includes) file y.

Definition : A include dependency graph is a triple $G = (F, I, r)$, where

- F is a finite set of files
- $I \subset (F \times F)$ is the include-dependency relation, i.e., a pair $< u, v > \in F$ exists iff file u includes file v; and
- there is a path in F connecting the root file, $r \in F$, to every file

The above definition captures two typical properties of flow graphs: there is a specific node at which to begin, and every node is accessible from this initial node.

At the bottom of the graph, one finds low level interface files that do not include any other file. At the top of the graph, one finds files that are never included by any other file.

The file include graph without recursion (i.e., a directed acyclic graph) can be transformed into a tree of file instances by a depth-first search traversal of the include graph. Back-edges in the include graph corresponding to recursive includes can be detected by marking vertices as visited during the depth-first traversal.

Note that, since this is only a static graph, the potential impact of conditional compilation statements are not taken into account. In reality, this is not complete. Each oriented arc is associated with a condition. If the condition is always true, it implies that one file unconditionally (always) includes another file, which is a strong dependency. If the condition is always false, it implies that this connection should be removed. In other situations, one gets the constraints of preprocessor variables to include the file. The IDG with each oriented arc decorated with compilation condition can become a starting point for refining the include hierarchy.

In this project, we not only generate the include dependency graph, but also provide the condition of each dependency relationship.

3.4 Metrics

The summary statistics provide a good objective basis for the comparison of different programs.

In order to quantify internal attributes of software, metrics have been defined on code, control flow graph and include dependency graph.

The metrics suite selected contains: code metrics, graph metrics, reuse metrics, and complexity metrics. The definitions of those metrics are given below.

Code metrics. The metrics defined on source code includes the number of each different type of directives:

- number of #include directives
- number of #defines
- number of #conditional directives

Graph metrics. These metrics defined on include dependency graphs can be divided into two main classes: size metrics e.g.

- number of nodes
- number of edges

and structure metrics e.g.

- nesting depth and width

These metric values help understanding the static dependencies.

Reuse metrics. A lot of metrics fall into this category:

Metrics defined on macros such as:

- number of macro identifiers
- number of preprocessor variables(macro symbol used in conditional directives)
- number of safeguards
- number of platform selectors

Metrics defined on conditions such as:

- number of different preprocessor variables involved in the condition
- number of nested conditions

Metrics defined on included files such as:

- fan-in of header files
- fan-out of header files

The fan-in gives the number of files that include a particular file, and the fan-out is the number of files included by a file. These metrics show file level coupling characteristics. Metrics defined on system include:

- total number of include files

- number of distinct include files
- number of high level include files (the root file of IDG)
- number of low level include files (the leaf file of IDG)

Complexity metrics are metrics that measure size or logical structure of the software. One of the most commonly used complexity metric is

- lines of code(LOC)

other examples are:

- number of preprocessor directives
- lines of ordinary C/C++ code

3.5 Summary

In this chapter, we have defined different analysis view information. Views can be grouped into four classes [3]:

- Class 1: nonprocedural, and/or meta-oriented views
- Class 2: pseudo-procedural, and/or architectural-oriented views
- Class 3: highly procedural views, or close derivatives
- Class A: analysis views that may accompany any other view

Compilation condition and macro classification is Class 3 view; include dependency falls into Class 2; metrics are derived by analyzing software, which fall into Class A.

CHAPTER 4

TECHNIQUES AND IMPLEMENTATION

4.1 Overview

In figure 4.1, the system architecture of the tool built can be seen.

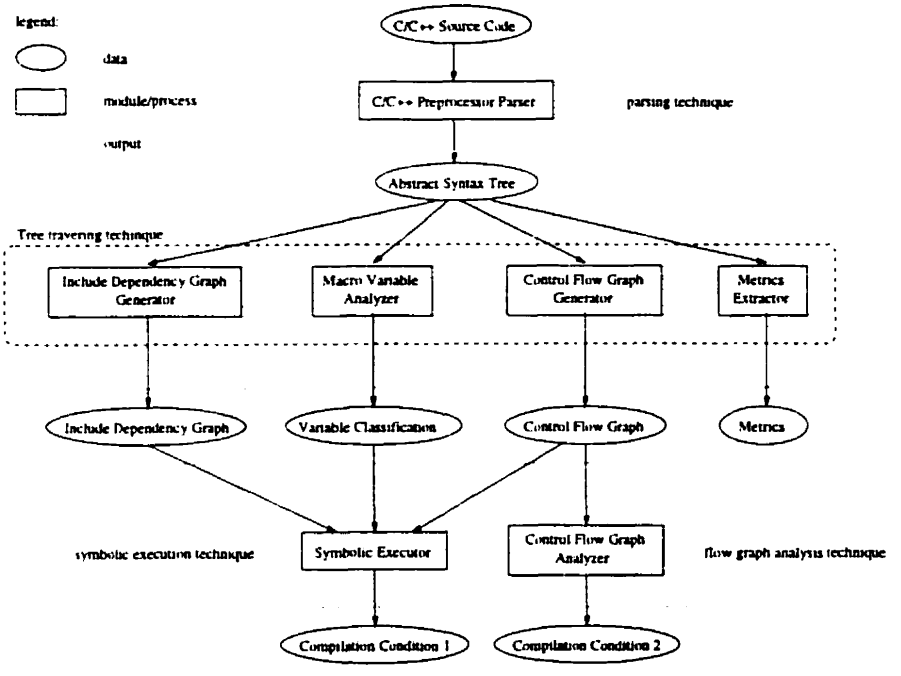


Figure 4.1: System architecture of the tool

The system consists of four phases: parsing, abstract syntax tree traversal, symbolic execution, and flow graph analysis. Multiple view information is extracted from source code and stored in the information base for further external analysis.

4.2 C/C++ Preprocessor Parser

In order to extract information we have first to represent code in a data structure. The analysis starts by abstracting the source code into an Abstract Syntax Tree (AST). The AST is a tree-based representation of the tokens contained in the source code. It provides an exact representation of the source code. The advantages of this representation are:

- it does not require any overhead to be computed as it is a direct product of the parsing process, and
- it can be easily analyzed to compute several data and control flow program properties

Creating the AST is a three-step process. First, a grammar and an object (domain) model must be designed for the programming language of the subject system. The second step is to use the parser on the subject system to construct the AST representation of the source code. Once the AST is created, further steps operate in an essentially language-independent fashion.

In this project we choose JavaCC(Java Compiler Compiler) [32] to generate the C/C++ preprocessor parser. JavaCC is a compiler generator that accepts language specifications in BNF-like format as input. The generated parser contains a lexical analyzer and a syntax analyzer. The lexical analyzer "*TokenManager*" is used to group characters from an input stream into tokens according to specific rules.

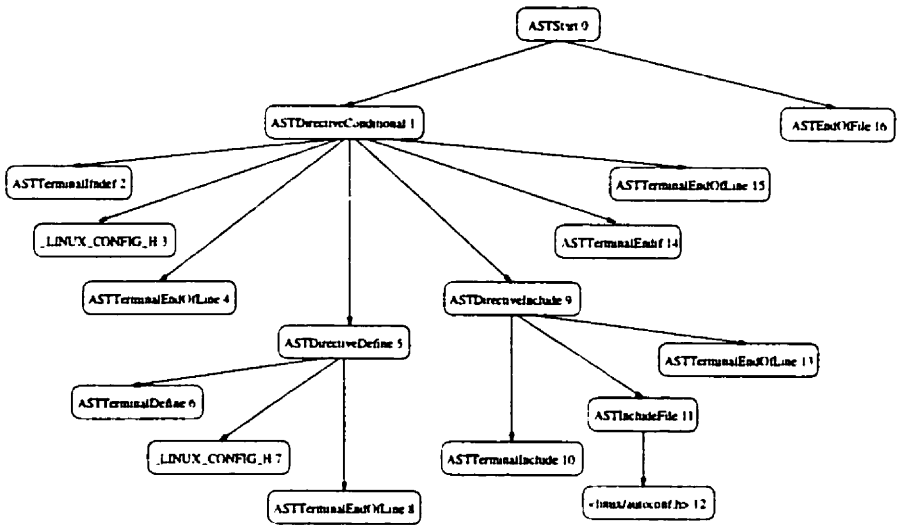


Figure 4.2: A sample AST

The syntax analyzer in JavaCC is a recursive-decent LL(k) parser. This type of parser uses k number of lookahead tokens to generate a set of mutually exclusive productions, which recognize the language.

Generating C/C++ preprocessor parser has three steps:

- a BNF-like grammar for the preprocessor was written
- then JJtree was used to build the Abstract Syntax Tree
- finally, JavaCC (Java Compiler Compiler) was used to automatically generate the C/C++ preprocessor parser

A sample AST for a C/C++ code is given in figure 4.2. Nodes of the AST are represented as objects. The AST nodes fall into two categories *ASTNonTerminal* nodes and *ASTTerminal* nodes. All the leaves in AST are *ASTTerminal* nodes.

4.3 Information Extraction from Abstract Syntax Tree

A syntax tree depicts the natural hierarchical structure of a source program. The next step associates with semantic interpretation and dependence analysis. At this step, some intermediate models and formats are defined, which are specified in the same way for all source languages, so that the subsequent analysis tools need not know about the language idiosyncrasies. The intermediate information extraction process corresponds to a depth-first traversal of the abstract syntax tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left-to-right order.

There are four core components related to directly extracting information from AST. Table 4.1 lists modules' names and corresponding functionality. Each module defines numerous operations on the AST nodes. Most of these operations need to treat different nodes in different ways. Distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It is better if each new operation can be added separately, and the node classes are independent of the operations that apply to them. The methodology used here is to develop visitors for each operation. E. Gamma et al [16] have given a detailed introduction to the *visitor* design pattern.

With the *visitor* pattern, one defines two class hierarchies: one for the elements being operated on and one for the visitors that define operations on the elements. Figure 4.3 gives the node hierarchy, figure 4.4 shows the visitor hierarchy.

Table 4.1: Core components of source code analysis process

Module	Functionality
Include dependency Graph Generator	Generate static include hierarchy graph annotate edge by compilation condition
Macro Variable Analyzer	Classify macro variables
Control Flow Graph Generator	Derive control flow from AST
Metrics Extractor	Count corresponding variables

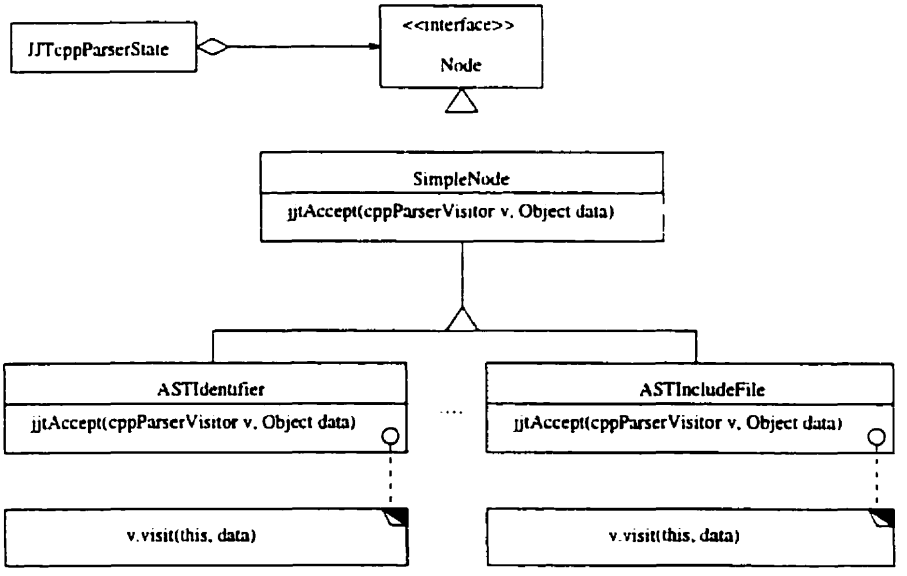


Figure 4.3: AST node hierarchy

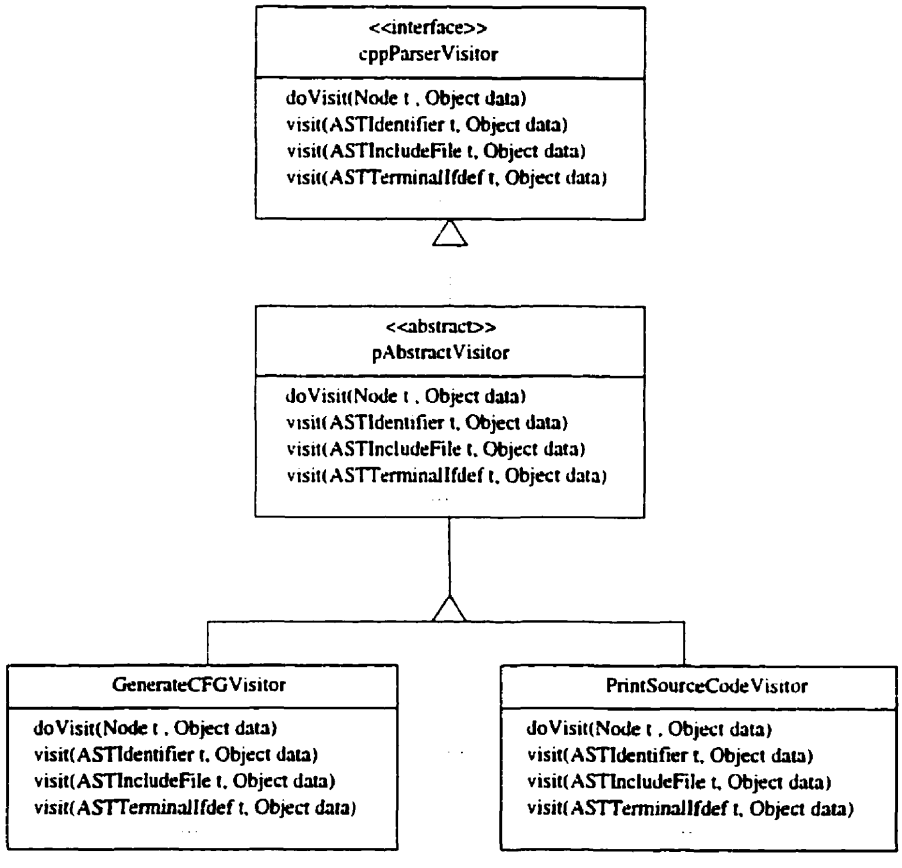


Figure 4.4: Visitor hierarchy

```
public Object doVisit(Node t, Object data) {
    initVisit();
    Object o = t.jjtAccept(this,data);
    return finalizeVisit(o);
}
```

Figure 4.5: Method *doVisit*

```
protected Object visitNonTerminal(Node t, Object data){
    if (t != null) {
        for (int i = 0; i < t.jjtGetNumChildren(); i++ ) {
            Node n = t.jjtGetChild(i);
            n.jjtAccept(this,null);
        }
    }
    return null;
}
```

Figure 4.6: Method *visitNonTerminal*

First an abstract parent class *pAbstractVisitor* for all visitors of an abstract syntax tree is needed. Abstract visitor defines *doVisit* method (figure 4.5) which is the only method invoked externally. Class *pAbstractVisitor* must also declare an operation for each node class. It uses a *visitNonTerminal* method (figure 4.6) to visit *NonTerminals* and a *visitTerminal* method to visit terminal nodes. Method *visitNonTerminal* implements a recursive left-to-right depth-first traversal.

The *visitor* pattern relies on a technique called double dispatching [6, 46]. Using this technique, one can ensure that the method executed by a message sent, depends

```
Generate_Include_Graph(AST_of_file_i)
1 construct node i in IDG for file i
2 if file i includes file j an oriented arc(f_i, f_j)is created
```

Figure 4.7: Algorithm for generating static include dependency graph

```
public Object visit(ASTStart t, Object data){
      construct a node in IDG for this file
}
public Object visit(ASTDirectiveInclude t, Object data){
      create an oriented arc from the node of the
      current file to the node of included file
}
```

Figure 4.8: Visit methods defined in *IDGConstructor* visitor

not only on the runtime value of the receiver, but also on the value of the message's argument.

Our four core components described in Table 4.1 are four concrete applications. Each defines its operation by adding a new subclass to the *visitor* class hierarchy.

Include dependency graph is built by traversing the AST and capturing each *#include* statements. For each *#include* statement, one directed edge from the analyzed file to the included file is generated. The algorithm to generate the static include dependency graph is given in figure 4.7.

The above algorithm is easily implemented via the *visitor* model. Figure 4.8 displays visit methods defined in *IDGConstructor* visitor.

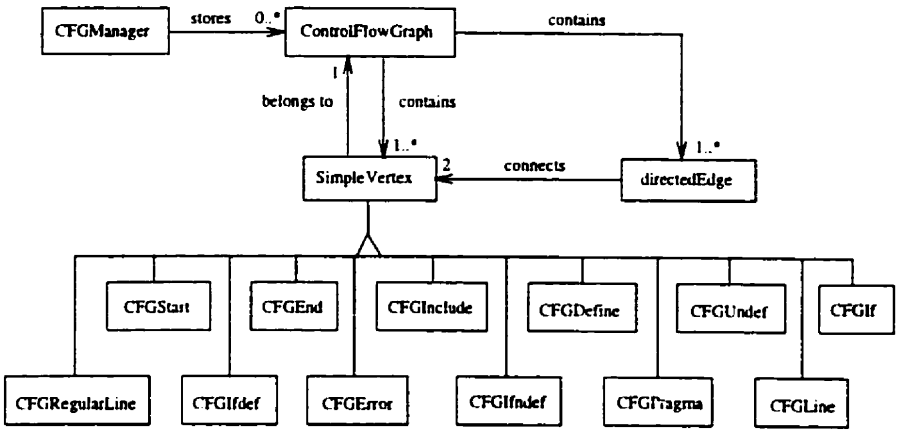


Figure 4.9: Class relationships in control flow graph design

A control flow graph (CFG) for a header file consists of nodes which represent single-entry, single-exit regions of executable code and edges which represent possible execution branches between code regions. Formally CFG is a quadruple (N, ε, v_s, v_e) where N is the set of nodes, ε is the set of edges, v_s is the entry node, and v_e is the exit node. The control flow of the preprocessor directives is similar to, but simpler than the one of ordinary programming language. There are macro definitions similar to assignment statements, file inclusions similar to function call statements, conditional directives similar to the branch statements, but there is no analogue to loop statements. Furthermore, variables in preprocessor are all global. The nodes in CFG are typed.

$$CFG\_Node\_Types = Preprocessor\ \ Directives \cup Regular Line$$

Figure 4.9 illustrates class design of control flow graph. The construction of

control flow graph follows its definition. For each *ASTDirective* node, a corresponding CFG node is generated, while traversing the AST, *directedEdges* are also added to the *ControlFlowGraph*. The algorithm is easily implemented as *visit* methods (see figure 4.10).

Macro variable analysis is based on pattern matching. For example, figure 4.11 gives the implementation of the algorithm to find all the direct preprocessor variables.

Code metrics computation is based on counting different AST nodes.

Generally speaking, a client that uses the *visitor* pattern must create a *ConcreteVisitor* object and then traverse the object structure, visiting each element with the *visitor*. When an element is visited, it calls the *visit* operation that corresponds to its class. The element supplies itself as an argument to this operation to let the *visitor* access its state, if necessary. *Visitor* declares a *visit* operation for each class of *ConcreteElement* in the object structure. The operation's name and signature identifies the class that sends the *visit* request to the *visitor*. That lets the *visitor* determine the concrete class of the element being visited. Then the *visitor* can access the element directly through its particular interface. *ConcreteVisitor* implements each operation declared by *visitor*. Each operation implements a fragment of the algorithm defined for the corresponding class of objects in the structure. *ConcreteVisitor* provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of

```
public Object visit(ASTStart t, Object data){
   generate a new ControlFlowGraph g;
   generate a CFGStart node v_s,
   add v_s to g ;
   set it to be initial node of g;
   visit each child of t;
   generate a CFGEnd node v_e after all the children have
   been visited;
   generate directedEdge from previous nodes to v_e
   and them to g
}

public Object visit(ASTDirectiveDefine t, Object data){
   generate a CFGDefine node,
   visit t.child to set attributes of CFGDefine node
   add the CFGDefine node to the ControlFlowGraph
   construct a directedEdge from each previous node to this node,
   add the generated directedEdge(s) to the ControlFlowGraph
}

public Object visit(ASTDirectiveUndef t, Object data)
                ...
public Object visit(ASTDirectiveInclude t, Object data)
                ...
public Object visit(ASTDirectiveError node, Object data)
                ...
public Object visit(ASTDirectivePragma node, Object data)
                ...
public Object visit(ASTDirectiveLine node, Object data)
public Object visit(ASTRegularLine node, Object data)
public Object visit(ASTDirectiveConditional t, Object data)
public Object visit(ASTTerminalIf node, Object data)
public Object visit(ASTTerminalIfdef node, Object data)
public Object visit(ASTTerminalIfndef node, Object data)
public Object visit(ASTTerminalElif node, Object data)
public Object visit(ASTTerminalElse node, Object data)
public Object visit(ASTTerminalEndif node, Object data)
```

Figure 4.10: Methods defined in *GenerateCFGVisitor*

```
public Object visit(ASTIdentifier t, Object data){
      if t.ancestor is an conditional expression
 Identier t is direct preprocessor variable;
}
```

Figure 4.11: Algorithm used in finding direct preprocessor variables

the structure. *Element* (e.g. *SimpleNode*) defines an *accept* operation that takes a *visitor* as an argument. *ConcreteElement* (e.g. *ASTIncludeDirective*) implements an *accept* operation that takes a visitor as an argument. *Object* structure can enumerate its elements, may provide a high-level interface to allow the *visitor* to visit its elements, and may either be a composite or a collection such as a list or a set. The *visitor* pattern encapsulates the operations for each compilation phase in a *visitor* associated with that phase.

In this project, we implemented 23 concrete AST visitors. Most of them work for four core components described in Table 4.1.

4.4 Control Flow Analysis

Definition : A flow graph is a triple $G = (N, A, s)$, where (N,A) is a (finite) directed graph, and there is a path from the initial node, $s \in N$, to every node.

A flow graph is an abstract graph model of a control flow graph with statements, and instructions inside each basic block.

The definition of flow graph captures two typical and reasonable properties of

control flow graphs: there is a specific node at which to begin, and every node is accessible from this initial node. An important restriction on flow graphs follows from the nature of branches in programs.

Definition : A flow graph $G = (N, A, s)$ in which $r = O(n)$, where $n = \sharp N$ $r = \sharp A$, is called a sparse flow graph.

In reality, virtually all flow graphs resulting from programs are sparse because (a) binary branching is generally used for control flow, (b) programmers use disciplined and sparse control flow structures for conceptual simplicity. When no branching more complex than binary is used, $r \leq 2n$. The significance of sparseness is reflected by algorithmic complexity. If sparseness is assumed, $O(r) = O(n)$.

Definition : If x and y are two (not necessarily distinct) nodes in a flow graph G, then x dominates y iff every path in G from its initial node to y contains x. for convenience, $DOM(y) = \{x | x \ dominates \ y\}$, for each node y. Node x properly dominates y iff $x \neq y$ and x dominates y.

We say that x directly dominates (or immediately dominates) y iff (a) x properly dominates y, and (b) if z properly dominates y, and $z \neq x$, then z (properly) dominates x.

The dominance relation of a flow graph G is a partial ordering. The initial node s of a flow graph G dominates all nodes of G. The dominators of a node form a chain (i.e., a linear ordering). Every node except s has a unique direct dominator. The graph of the reflexive and transitive reduction of the dominance relation of

```
1   D(n0) := {n0};
2   forall n ∈ V − {n0} do D(n) := V
3   while some D(n) are changed do
4         forall n ∈ V − {n0} do
5            D(n) := {n} ∪ {∩p/<p,n>∈E D(p)};
```

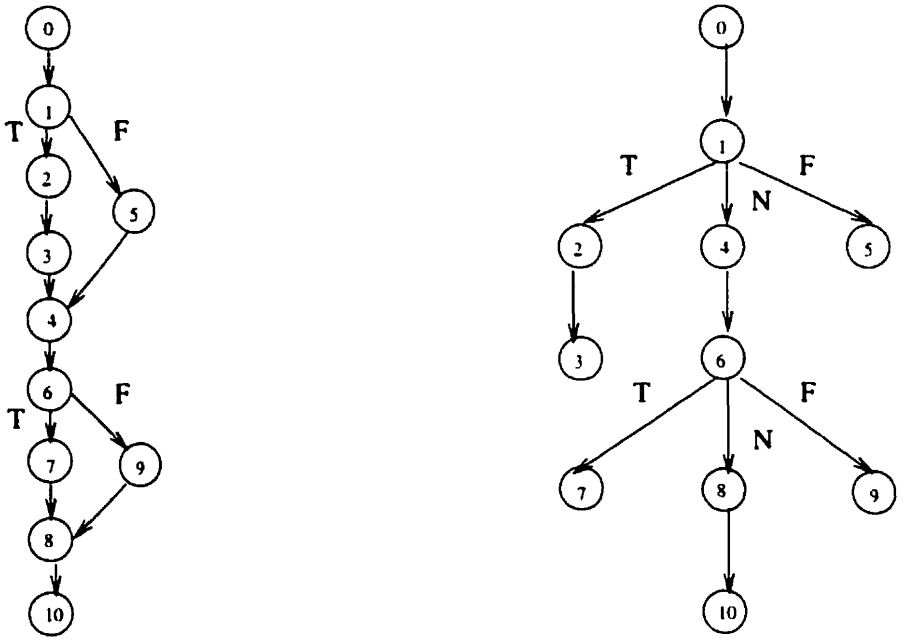
Figure 4.12: Algorithm used in finding dominators

a flow graph is a tree. There is an arc (x, y) in the tree if and only if x directly dominates y, and there is a path from u to v in the tree if and only if u dominates v. We say node d of a flow graph dominates node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d. Under this definition, every node dominates itself, and the entry of a branch dominates all nodes in the branch.

A useful way of presenting dominators' information is in a tree, called the dominator tree, in which the initial node is the root, and each node d dominates only its descendants in the tree.

The existence of dominator trees follows from a property of dominators; each node n has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has that property that if $d \neq n$ and d *dom* n, then d *dom* m.

In figure 4.13 we show a flow graph and its dominance tree. In this project, we are going to use dominator tree to get intended necessary condition to compile any line of code. The control flow in preprocessor is very simple, only branches exist.



(a) (b)

Figure 4.13: CFG example and corresponding dominator tree

The conditional node is a decision point. Only conditional nodes have more than one child in the dominator tree, other nodes have only one direct child. For each conditional node, there are at most three outgoing edges, which have the following possible path conditions:

- the condition is true, or
- the condition is false, or
- no matter the condition is true or false

In figure 4.13 (b), we annotate the condition value on each edge, T for "$condition == true$", F for "$condition == false$", and N for null condition. The necessary con-

dition to one node is the accumulation of the conditions along the path of the dominator tree from root to the node.

4.5 Symbolic Execution

Symbolic execution, sometimes referred to as symbolic evaluation, differs from the traditional sense of executing a program. The traditional notion of execution requires that a selection of paths through the program is exercised by a set of test cases. In symbolic execution, actual data values are replaced by symbolic values. Traditional programs execute using inputs consisting of actual values. Symbolic execution, on the other hand, produces a set of expressions, one expression per output variable [7].

The most common approach to symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph, which is a directed graph containing decision points and the assignments associated with each branch. By traversing the flow graph from an entry point along a particular path, a list of assignment statements and branch predicates is produced.

The resulting path is represented by a series of input variables, condition predicates and assignment statements. The execution part of the approach takes place by following the path from top to bottom. During this path traversal, each output variable is given a symbol in place of an actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input

variables and constants.

At the end of the symbolic execution of a path, the outputs will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions will be subject to constraints. A list of these constraints is provided by the set of symbolic representations of each condition predicate along the path. Analysis of these constraints may indicate that the path is not executable due to a contradiction. This infeasibility problem is encountered in all forms of path testing.

Partition analysis uses symbolic execution to identify subdomains of the input data domain. Symbolic execution is performed on both the software and the specification. The path conditions are used to produce the subdomains, such that each subdomain is treated identically by both the program and the specification. Where a part of the input domain cannot be allocated to such a subdomain then either a structural or functional (program or specification) fault has been discovered. Symbolic evaluation occupies a middle ground between testing data and program proving.

Symbolic execution can be applied to preprocessing directives to achieve the analysis of conditional compilation. Most preprocessor features are active when enabled explicitly using preprocessing directives.

The basic idea of symbolic execution on preprocessor directives is described in the following example. In executing the code

```
#define __KERNEL__
  code segment 1
#else
  code segment 2
#endif
```

when $\_\_KERNEL\_\_$ is either defined or undefined in the execution path, the program splits into two paths, one in which "$defined\ \_\_KERNEL\_\_$" is asserted and code segment 1 is executed, and the other in which "! $defined\ \_\_KERNEL\_\_$" is asserted, and code segment 2 is executed. If $\_\_KERNEL\_\_$ is defined, only code segment 1 is chosen to be executed; if $\_\_KERNEL\_\_$ is undefined, only code segment 2 is chosen to be executed. Symbolic execution represents many different executions.

Effective use of symbolic values requires a capability for the simplification of symbolic expressions. and support for splitting a single execution path into multiple paths when symbolic boolean values are encountered in the conditional statements.

The symbolic execution is conducted on the control flow graph (CFG) of preprocessor directives. The procedure is managed by an active node stack, which is a priority queue of $Stateinfos$. $Stateinfo$ is a data structure containing a reference to CFG node, an attached symbol table, a history list of include directives and path condition. For different type of CFG nodes, the action is different.

Figure 4.14 shows class design of active node stack and attached state informa-

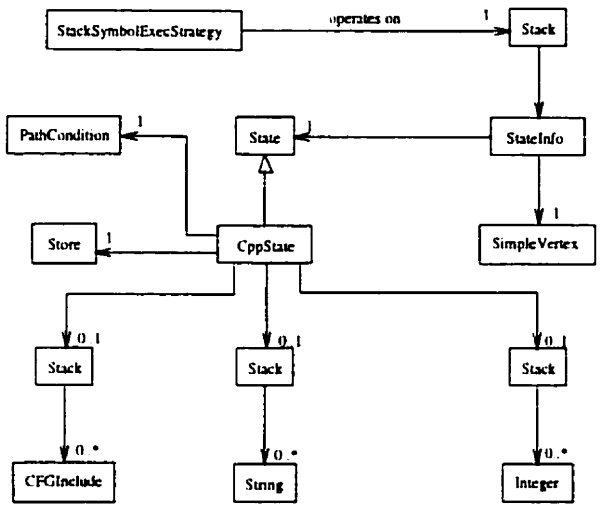


Figure 4.14: Class design of state information

tion (*Stateinfo*).

Description of information in *CppState*:

- *PathCondition* keeps the current condition
- *Store* is a symbol table to keep track of macro definition
- a *CFGInclude* node stack is used to keep track of the return point of each file inclusion
- a *String* stack is used to memorize file names.
- an *Integer* stack, where a different number is assigned to each file instance

To achieve different functionalities, we need to choose different algorithms. The *strategy* design pattern is used to encapsulate different algorithms and make them interchangeable.

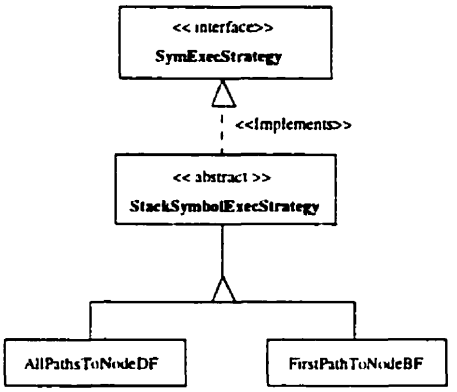


Figure 4.15: Class design of symbolic execution strategy

```
public void execute(ControlFlowGraph,State)
1  getFirst()
2  while(currentIsValid())
3     processCurrent()
4     getNext()
5  output()
```

Figure 4.16: Symbolic execution strategy

Figure 4.15 shows class design of symbolic execution strategies.

Figure 4.16 to figure 4.23 list all the algorithms used in the symbolic execution.

The symbolic execution is based on the control flow graph (CFG) of preprocessor directive and file instance graph. The procedure is managed by an active node stack, which is a priority queue of CFG nodes.

To find one sufficient condition to compile any given code line, we use breadth-first-search algorithm. Breadth-first-search is one of the simplest algorithms for searching a graph and the archetype of many important graph algorithms. It

```
protected void getFirst()
1  construct an active node stack
2  push the first node of CFG and attached state information into
   the stack
```

Figure 4.17: Get the start point of the execution

```
protected void processCurrent()
1  update the attached store of the top node in the active node
   stack
```

Figure 4.18: Update corresponding store

```
protected boolean currentIsValid()
   // seek for all the paths to a given node
1  if(state_infos.isEmpty())// no node to process
2      return false
3  else if (the top node of state_infos is target node)
       //given node found, make a union of path conditions
4      path_condition.dump()
5      return true
6  else // continue to process
7      return true
```

Figure 4.19: Validate current node for finding all paths

```
protected boolean currentIsValid()
   // seek for the shortest path to a given node
1  if(state_infos.isEmpty())// no node to process
2      return false;
3  else if (the top node of state_infos is target node)
       //given node found
4      record path_condition
5      return false
6  else // continue to process
7      return true
```

Figure 4.20: Validate current node for shortest path

```
protected void output()
1  path_condition.dump();
```

Figure 4.21: Dump path condition

```
protected void getNext() //DFS
1  pop the stateinfo from the stack state_infos
2  get the successors of the current node
3  put the successors into the head of the stack
```

Figure 4.22: Find next executing element using DFS

```
protected void getNext() //BFS
1  pop the stateinfo from the stack state_infos
2  get the successors of the current node
3  put the successors into the tail of the stack
```

Figure 4.23: Find next executing element using BFS

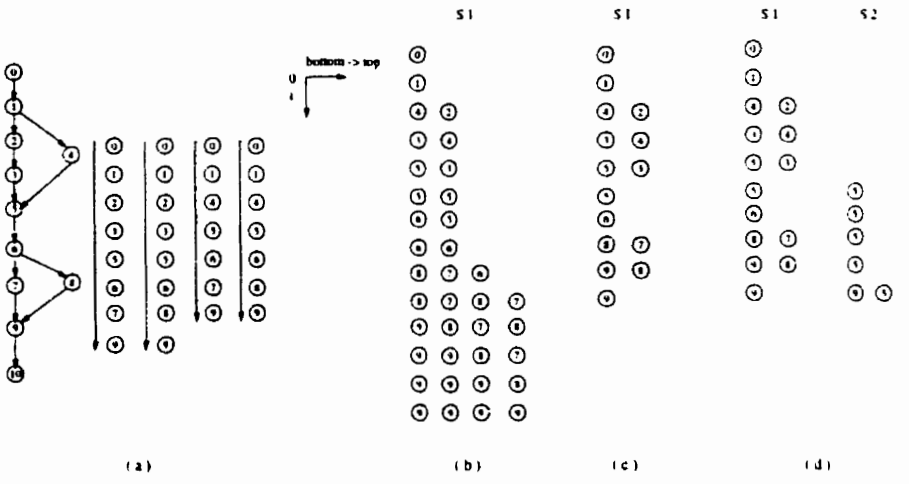


Figure 4.24: Examples of BFS

expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k+1$. The algorithm uses a queue to manage the vertices. There are three approaches in using BFS algorithm in our projects. Figure 4.24 and figure 4.25 illustrate how the three approaches work. In figure 4.24, we assume all four paths are plausible (figure 4.24(a)), while in figure 4.25 we assume that only two paths are plausible (figure 4.25 (a)).

The first approach is that the vertex is put into the queue each time it is encountered during the search. The status of active node stack $S1$ is depicted in figure 4.24 (b) and figure 4.25 (b). The problem with the first approach is that the size of the stack monotonously increases due to the branching of CFG conditional node before the target node is reached.

The second one is that the vertex is put into the queue only at the first time it is

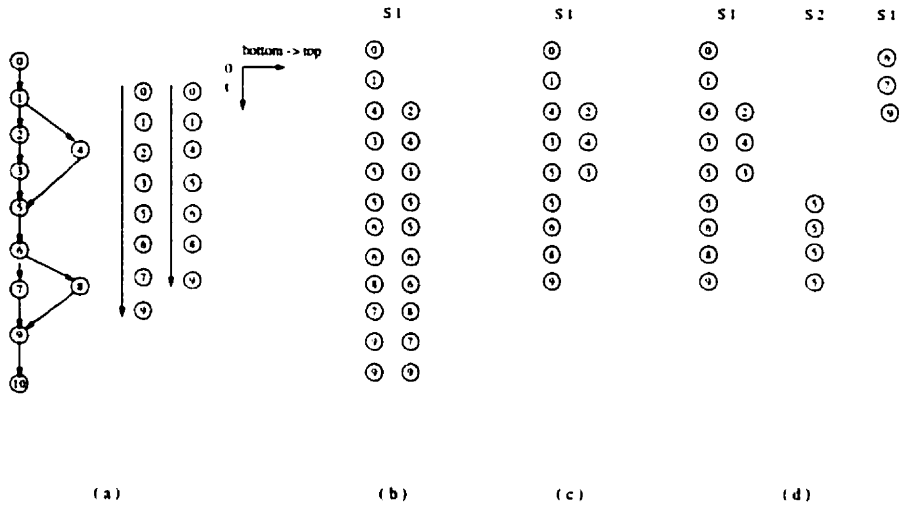


Figure 4.25: Examples of BFS

encountered during the search. The content of active node stack $S1$ is depicted in figure 4.24 (c) and figure 4.25 (c). In fact, the longer path is automatically thrown away by the algorithm. There is a problem with the second approach: if later the path is found to have an impossible constraint, the assumption under which the longer path is thrown away become invalid. This path, longer than an infeasible path, may still be the shortest path.

The third approach is to use two stacks to store nodes and attached information. The first stack is an active node stack, same as the one used in the first two approaches. Now instead of simply ignoring the node visited already, we push the visited node into the second stack. The nodes in the second stack will not be further expanded, until the search in the first stack is exhausted.

To find full path conditions to compile any given code line, we use depth-first-search (DFS) algorithm. For DFS, the strategy is to search deeper in the graph

whenever possible. In DFS, edges are explored out of the most recently discovered vertex v that has unexplored edges leaving it. When all of these edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as the new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered. The algorithm uses a queue (stack) to manage the vertices. If the action of one node being put into stack is counted as one operation, the include graph of file without recursion (i.e., a directed acyclic graph) can be transformed into a tree of file instances by a depth-first search traversal of the include graph. The total number of such operations is proportional to the number of nodes in its fully expanded executing tree. The big concern with DFS strategy is performance. We propose two solutions to address this problem:

- initializing some preprocessor variables in the beginning of the symbolic execution. based on the knowledge of programs to reduce the branching
- simplifying the control flow graph

The complete control flow graph of preprocessor directives keeps all the information of the code, which is far more than what we need to do conditional compilation analysis. This dramatically reduces the performance of the analysis procedure. In

some situations, for example in analyzing code with presence of file inclusion, a reduced control flow graph may be sufficient for the included file. However, the complete control flow graph need to be reduced, sufficiently and safely. Here are the rules:

- all the control flow nodes, except macro definitions (#define, #undef), file inclusions (#include) and conditional directives (#if, #ifdef, #ifndef, #elif), can be safely removed, since these directives will not change control flow
- Macro definition node (#define, #undef), if the macro identifier is neither a preprocessor variable nor an indirect preprocessor variable, it can be safely removed
- conditional node (#if, #ifdef, #ifndef, #elif), if neither branch contains other nodes, it can be safely removed
- include nodes should never be removed

4.6 Summary

In this chapter we have briefly illustrated the techniques and approaches used in the project. Some implementation details were also presented. For each approach, its advantages and disadvantages were discussed.

CHAPTER 5

EXPERIMENTS

We have applied our tool to some large real software systems. Experiments were conducted on a Pentium III 500MHz with 512MB RAM running Linux.

Here the Linux kernel was chosen as a case study, since it is a large well constructed and widely used system.

Within the kernel layer, Linux is composed of five major subsystems: the kernel(*kernel*), the memory manager(*mm*), the virtual file system(fs), the network interface (*net*), and the interprocess communication (*ipc*). This decomposition is quite close to the directory structure of the source code, as shown in figure 5.1. In linux-2.2.12 kernel there are 1630 C source files (.c files), which are distributed in 127 directories; and there are 1273 header files distributed in 78 directories. System header files are in directory linux-2.2.12/include. System header files declare the interfaces to parts of the operating system. Programmers includes them to supply the definitions and declarations to invoke system calls and other library functions.

5.1 Code Metrics

Metrics can be used to extract dependencies between resources, where a resource can be a subsystem, file, module, procedure, or variable. Several metrics on

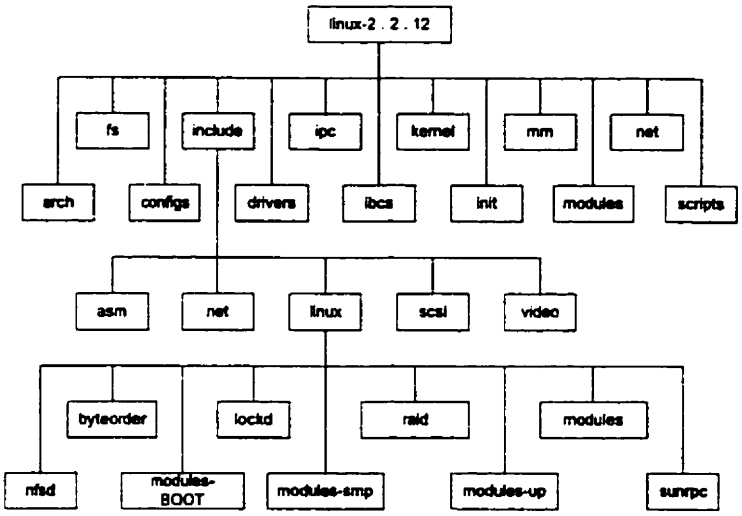


Figure 5.1: Directory structure of linux-2.2.12

the preprocessing directives were calculated, and a graphical representation of the include file hierarchy for a given C source file was built. Such metrics can provide a starting point for understanding a system architecture.

We first collect metrics for each subsystem of the kernel and each subsystem of the include directory, and then use the statistics of each subsystem to get a better understanding of the whole system.

The first experiment is to extract several simple metrics for each system header file. Metrics are defined as:

- $Incl\_d_i$: number of include directives in file i ($M1$ in Table 5.1)
- $Incl\_f_i$: size of the set of include files directly included by file i ($M2$ in Table 5.1)
- $Cond\_d_i$: number of Conditional directives in file i ($M3$ in Table 5.1)

- $Total\_d_i$: total number of directives in file i ($M4$ in Table 5.1)

- $Missing\_f_i$: size of the set of include files, which are outside directory linux-2.2.12/include, included by file i ($M5$ in Table 5.1)

In linux-2.2.12, system header files are in directory linux-2.2.12/include (see figure 5.1). Within the include directory there are five subdirectories, of which linux is the core. Within include/linux there are nine subdirectories, while most header files are in the include/linux directory itself. Table 5.1 lists the result for a few header files in directory linux-2.2.12/include/linux. From Table 5.1, we can see that some files don't include other files, and only serve as macro definitions, with $Incl\_d_i = 0$. We checked all 373 files in this directory, we found that an ordinary header file includes less than five other header files. File modversions-up.h includes 93 files in directory include/linux/modules-up, which store version signature for each module used for checking if such module matches the current kernel. $Missing\_f_i$ is the number of include files not found within directory linux-2.2.12. After checking the conditions for including these files, we found that either they are used for other platforms, thus not in the directory set for our platform, or they are not used for kernel, or they are supposed to be created during the build process.

The second experiment is to extract several composite metrics for source files (.c file) in each subsystem. These metrics are defined as:

Table 5.1: Extracted metrics for header files under linux-2.2.12/include/linux

file name	M1	M2	M3	M4	M5
a.out.h	2	2	36	87	0
acct.h	2	2	3	15	0
adfs_fs.h	1	1	2	23	0
adfs_fs_i.h	0	0	1	2	0
adfs_fs_sb.h	1	1	1	3	0
affs_fs.h	1	1	1	8	0
affs_fs_i.h	2	2	1	7	0
affs_fs_sb.h	0	0	1	18	0
affs_hardblocks.h	0	0	1	5	0
amifd.h	1	1	3	7	0
amifdreg.h	0	0	2	40	0
amigaffs.h	2	2	2	67	0
apm_bios.h	1	1	2	51	0
arcdevice.h	2	2	9	71	0
atalk.h	1	1	6	31	0
atari_rootsec.h	0	0	1	2	0
auto_fs.h	5	5	3	16	0
autoconf-BOOT.h	0	0	0	574	0
autoconf-smp.h	0	0	0	1028	0
autoconf-up.h	0	0	0	1031	0
autoconf.h	6	5	7	20	2
awe_voice.h	0	0	6	133	0
ax25.h	0	0	1	29	0

- $INCL\_F_i$: size of Closure(F_i), the set of include files directly or indirectly included by file i ($M1$ in Table 5.2)

- $INCL\_D_i$: number of include directives in file i, which is calculated based on the Closure(F_i)

  $INCL\_D_i = \sum_{f_j \in Closure(F_i)} Incl\_d_i$ ($M2$ in Table 5.2)

- $COND\_D_i$: number of conditional directives which may be used in file i

  $COND\_D_i = \sum_{f_j \in Closure(F_i)} Cond\_d_i$ ($M3$ in Table 5.2)

- $TOTA\_D_i$: number of all directives which may be used in file i $TOTAL\_D_i = \sum_{f_j \in Closure(F_i)} Total\_d_i$ ($M4$ in Table 5.2)

- $MISSING\_F_i$: size of Closure(MF_i), the set of include files, which are outside directory linux-2.2.12/include, directly or indirectly included by file i ($M5$ in Table 5.2)

- $Depth\_I_i$: include hierarchy depth of file i. ($M6$ in Table 5.2)

Table 5.2 lists the extracted metrics for the kernel subsystem of linux-2.2.12

Table 5.2: Extracted metrics for source files in linux-2.2.12/kernel

file name	M1	M2	M3	M4	M5	M6
acct.c	259	157	383	6553	6	12
capability.c	239	150	371	6480	6	12
dma.c	254	155	389	6680	6	15
exec_domain.c	507	405	405	13908	8	12
exit.c	530	412	431	14148	8	14
fork.c	529	415	433	14360	8	14
info.c	257	158	384	6749	6	12
itimer.c	253	155	378	6523	6	12
kmod.c	249	156	379	6652	6	12
ksyms.c	585	438	496	14771	8	15
module.c	524	410	423	14122	8	13
panic.c	251	159	389	6471	7	11
printk.c	256	158	384	6570	6	12
resource.c	234	151	374	6388	6	11
sched.c	277	167	418	6869	6	14
signal.c	518	411	427	14222	8	13
softirq.c	264	159	391	6657	6	14
sys.c	271	164	399	6768	6	14
sysctl.c	286	171	441	6838	6	15
time.c	253	155	380	6525	6	12

kernel.

From Table 5.2 one can find that an ordinary .c file may use about 150 header files (Metric $M2$). The depth of include hierarchy is about 12. Without adequate tools, it is impossible for programmers to figure out which part of the 150 include files is most relevant.

The above composite metrics can also be used to do version comparison. We performed the same analysis for the linux-2.2.3 kernel and give the results in Table 5.3.

Table 5.3: Extracted metrics for source files in linux-2.2.3/kernel

file name	M1	M2	M3	M4	M5	M6
acct.c	175	111	228	2733	3	10
capability.c	162	106	215	2578	2	10
dma.c	177	111	231	2773	2	13
exec_domain.c	169	111	230	2696	3	10
exit.c	181	116	243	2777	3	11
fork.c	184	119	243	2992	3	11
info.c	174	112	235	2935	3	10
itimer.c	173	110	226	2714	3	10
kmod.c	163	107	219	2794	3	10
ksyms.c	238	142	304	3374	2	12
module.c	181	115	240	2783	3	11
panic.c	162	110	220	2483	3	9
printk.c	174	112	231	2752	3	10
resource.c	153	104	210	2441	2	9
sched.c	190	122	260	2992	3	12
signal.c	179	118	252	3012	3	11
softirq.c	177	113	231	2721	3	12
sys.c	184	117	238	2818	3	12
sysctl.c	196	125	292	3132	3	12
time.c	171	110	231	2778	3	10

Table 5.4: Statistics of macro definition

measure	linux-2.2.12	linux-2.2.3
$NumDefineVar$	17288	28288
$NumUndefVar$	546	114
$NumDefineOnlyVar$	17222	28193
$NumUndefOnlyVar$	480	19
$NumDefAndUndefVar$	66	95
$NumDefOrUndefVar$	17768	28307

Comparing Table 5.2 and Table 5.3, one can notice that the Linux-2.2.12 kernel has more system header files, which may be due to the increased capability of the system.

5.2 Preprocessor Identifier Analysis

A macro is a sort of abbreviation which one can define once and then use later. Each macro has an identifier as name and an associated text string as body.

Macros are created using the *#define* preprocessor directive and may also be undefined using preprocessor directive *#undef*. First we count the number of variables being defined or undefined in all the C source files and header files (see Table 5.4).

The metrics displayed in Table 5.4 are:

- $NumDefineVar$: number of variables which have been defined in the files
- $NumUndefVar$: number of variables which have been undefined in the files

- *NumDefineOnlyVar*: number of variables which have been defined but never undefined in the files
- *NumUndefOnlyVar*: number of variables which have been undefined but never defined in the files
- *NumDefAndUndefVar*: number of variables which have ever been defined and also undefined in the files
- *NumDefOrUndefVar*: number of variables which have been either defined or undefined in the files

According to macro definition syntax format, macro can be classified as parameterized or non-parameterized. Non-parameterized macro definition can be further classified in the following specific categories: 1) null define, the *#define* gives only an identifier name but no macro body; 2) constant define, the macro is defined to be either a literal or an operator applied to constant values. Table 5.5 lists the statistics of macros that fit into these categories for linux kernel version 2.2.12 and version 2.2.3.

The metrics displayed in Table 5.5 are:

- *NumSimpleMacro*: number of identifiers which have been defined as constant
- *NumArgumentMacro*: number of identifiers which have been defined as parameterized macro

Table 5.5: Classification of macro definition

measure	linux-2.2.12	linux-2.2.3
NumSimpleMacro	15183	24801
NumArgumentMacro	1465	2265
NumNoValueMacro	699	1382
NumNovalAndSimpleMacro	19	33
NumNovalAndArgumentMacro	0	2
NumSimpleAndArgumentMacro	40	125

- *NumNoValue*: number of identifiers which have been defined but without macro body

- *NumNovalAndSimpleMacro*: number of identifiers which appear in both null define macro definition and constant definition

- *NumNovalAndArgumentMacro*: number of identifiers which appear in both null define macro definition and parameterized macro definition

- *NumSimpleAndArgumentMacro*: number of identifiers which appear in both constant macro definition and parameterized macro definition

To build a better understanding of how the preprocessor is used, we ran our tools to analyze preprocessor variable usage. Briefly, macro usage falls into the following three categories:

- constant definition (simple macros)

- function definition (argument macros)

Table 5.6: Statistics of the preprocessor identifier

measure	linux-2.2.12	linux-2.2.3
NumIdentifier	20611	29044
NumPPVariable	1095	1909
NumIPPVariable	123	448
NumGuard	602	1221
NumPlatformSelector	93	186

- conditional compilation

Macros which serve as constant definition or function definition are mainly extended in C code segment. In study of preprocessor directives, we are more interested in those which effect the control flow of compilation. The most important are those macros which appear in the conditional preprocessing directives. These macro identifiers are called preprocessor variables. Another important type of macros are those indirectly used by conditional compilation.

Table 5.6 presents statistics about preprocessor identifiers in linux-2.2.12 kernel and linux-2.2.3 kernel.

The metrics displayed are:

- *NumIdentifier*: number of macro identifiers
- *NumPPVariable*: number of macro identifiers used in conditional directives
- *NumIPPVariable*: number of macro identifiers which may be indirectly used in conditional directives

- *NumGuard*: number of macro identifiers used as safe guard of multiple inclusion

- *NumPlatformSelector*: number of macro identifiers used as platform selector

From Table 5.6, one can see that: 5% of macro identifiers in linux-2.2.12 are directly or indirectly used in conditional compilation, 50% of which are safe guarders, which are supposed to be not defined at the beginning. Here, platform selectors refer to those predefined variables used in conditional directives, they are never defined in header file or source files. The classification of macro identifiers is very important, it can be used to reduce the variable space of the symbolic execution phase.

It is interesting to know which and how many preprocessor variables have been used as file inclusion guarder. This information may become the starting point of reconstructing the include file system into a better organization.

Table 5.7 gives the statistics of preprocessor variables used as safe guarder for each subdirectory of linux-2.2.12/include.

For all 81 header files in directory linux-2.2.12/include/asm/ there are 72 files which have preprocessor variables used to prevent multiple inclusions of the same file. For all files in directory linux-2.2.12/include/linux/ there are 337 preprocessor variables for such usage.

In Table 5.8, we list some safe guarder preprocessor variables extracted from all

Table 5.7: Statistics of preprocessor variables used as safe guarder

directory	NumSafeGuard	NumNoSafeGuard	Total
asm-generic	2	1	3
asm-i386	72	9	81
linux	337	36	373
linux/byteorder	6	0	6
linux/lockd	8	0	8
linux/modules	0	0	0
linux/modules-BOOT	0	0	0
linux/modules-up	0	0	0
linux/modules-smp	0	0	0
linux/nfsd	11	0	11
linux/raid	12	1	13
linux/sunrpc	12	0	12
net	43	22	65
net/irda	41	2	43
scsi	4	0	4
video	19	1	20

the header files under directory linux-2.2.12/include/. From this table, we can find some clue of safe guarder name convention used in linux kernel. File safe guarder should be unique, usually it is transfered from file name, and directory name.

5.3 Include Hierarchy Graph

The exploration of relationships among include files is an example of software reverse engineering. Files refer to each other by means of file inclusions. Their relative order is relevant for the reader. The presence of numerous conditional directives makes the structure almost impossible to follow. The graphical views provide users with helpful structural information that is usually hidden in the pro-

Table 5.8: Examples of file name and corresponding safe guarder variable name

linux/adfs_fs_sb.h	_ADFS_FS_SB
linux/coda.h	_CODA_HEADER_
linux/efs_fs_i.h	__EFS_FS_I_H__
linux/byteorder/generic.h	_LINUX_BYTEORDER_GENERIC_H
linux/lockd/lockd.h	LINUX_LOCKD_LOCKD_H
linux/sunrpc/xdr.h	_SUNRPC_XDR_H_
asm-i386/checksum.h	_I386_CHECKSUM_H
asm-i386/irq.h	_ASM_IRQ_H
net/ipip.h	__NET_IPIP_H
scsi/scsicam.h	SCSICAM_H

gram text. File inclusion information can be used to generate make dependencies automatically. Include directives are analogous to function calls in an ordinary program, and an include hierarchy graph is analogous to a function call graph. The graph simply contains one node for each file that composes the system, and an oriented arc(f_i,f_j) is created when file f_i includes file f_j. At the bottom of the graph , one finds low level files that do not include any other file. At the top of the graph, one finds the file that need to be analyzed.

Figure 5.2 shows the include hierarchy of file acct.c. There are altogether 158 nodes in this graph. The top node is file acct.c itself which is labeled 0. In the graph, we give all the corresponding file names. For example, we can see that <linux/autoconf.h> (node 2) includes <linux/autoconf-BOOTsmp.h> (node 7), "/boot/kernel.h" (node 3), $< linux/autoconf - up.h >$ (node 4), $< linux/autoconf - smp.h >$ (node 5) and $< linux/autoconf - BOOT.h >$ (node 6)..

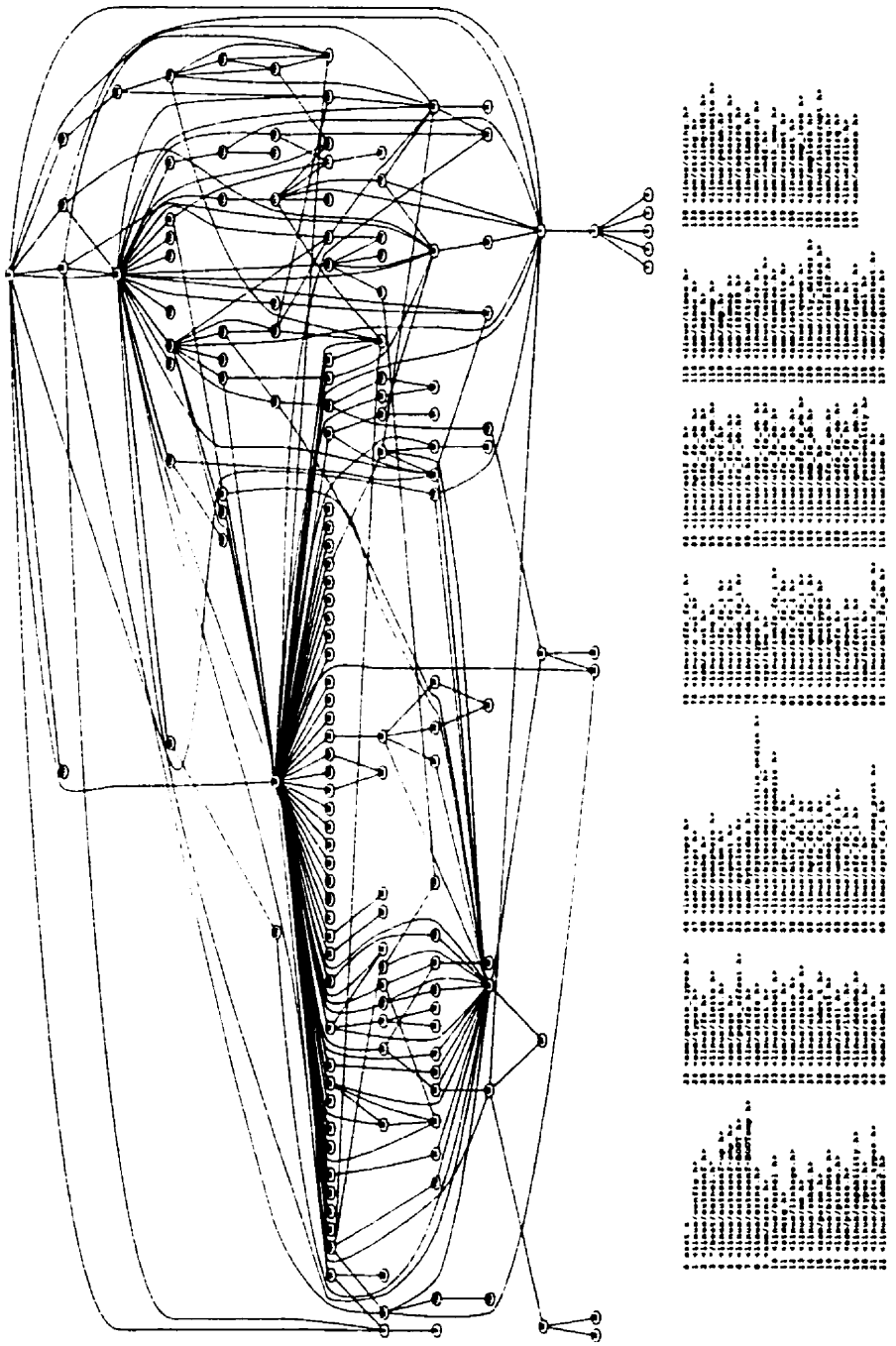


Figure 5.2: Include hierarchy of file acct.c

Table 5.9: Statistics of the header files

measure	linux-2.2.12	linux-2.2.3
$NumAllFile$	1043	1660
$NumRootFile$	182	563
$NumLeafFile$	396	268
$NumNodeFile$	208	186
$NumIndivFile$	257	643

The following information can be provided by such include hierarchy graph: how large the analyzed system is, what header files are involved. In Table 5.9 we collected some statistics on all header files in include directory of linux kernel

The metrics displayed are:

- $NumAllFile$: total number of header files
- $NumRootFile$: number of files which are only at the top of hierarchy graph
- $NumLeafFile$: number of files which are only at the bottom of hierarchy graph
- $NumNodeFile$: number of files which are internal nodes of hierarchy graph
- $NumIndivFile$: number of files which nether are included by other header files, nor includes other header files

Table 5.10 shows the measures for header files in directory linux-2.2.12/include/asm/, linux-2.2.12/include/linux/ and linux-2.2.12/include/net/.

Here are the metrics displayed in the table:

Table 5.10: Statistics for three subdirectories of linux-2.2.12/include (part 1)

measure	asm	linux	net
NumSrcFiles	81	754	108
NumIncAsm	25	47	10
NumIncLinux	17	460	48
NumIncNet	0	4	48
NumIncTotal	42	517	106
NumIncOut	0	9	2
NumNoIncFiles	44	517	40

- *NumSrcFiles*: number of files in the measured directory
- *NumIncAsm*: number of files in directory linux-2.2.12/include/asm/ which have been directly included by files in the measured directory
- *NumIncLinux*: number of files in directory linux-2.2.12/include/linux/ which have been directly included by files in the measured directory
- *NumIncNet*: number of files in directory linux-2.2.12/include/net/ which have been directly included by files in the measured directory
- *NumIncTotal*: number of files which have been directly included by files in the measured directory
- *NumIncOut*: number of files outside directory linux-2.2.12/include which have been directly included by files in the measured directory
- *NumNoIncFiles*: number of files in the measured directory, which don't include any other files

Table 5.11: Statistics for three subdirectories of linux-2.2.12/include (part 2)

measure	asm	linux	net
$NumIncByteorder$	1	3	0
$NumIncLockd$	0	4	0
$NumIncModules - smp$	0	56	0
$NumIncModules - up$	0	91	0
$NumIncModules$	0	92	0
$NumIncNfsd$	0	8	0
$NumIncRaid$	0	7	0
$NumIncSunrpc$	0	10	0
$NumInc/.$	16	189	48

For example, in directory linux-2.2.12/include/linux and all its subdirectories, there are 754 header files altogether, of which 373 files are in the current directory. They may directly reference (include) 460 file under directory linux-2.2.12/include/linux and its subdirectories, directly include 47 files in directory linux-2.2.12/include/asm, and 4 files in directory linux-2.2.12/include/net and all its sub-directories. They may include 9 files which are not under directory linux-2.2.12/include/. Furthermore, 517 files in this directory only contain macro definitions but do not include other files.

In Table 5.11 we give the statistics on each subdirectory of linux-2.2.12/include/linux/ for three subdirectories of linux-2.2.12/ include/. As we know, under linux-2.2.12/include/linux/, there are nine directories. Here are the statistics for these subdirectories.

- *NumIncByteorder*: number of files in directory linux-2.2.12/include/linux/byteorder which have been directly included by files in the measured directory

- *NumIncLockd*: number of files in directory linux-2.2.12/include/linux/ lockd which have been directly included by files in the measured directory

- *NumIncModules – smp*: number of files in directory linux-2.2.12/include/ linux/modules-smp which have been directly included by files in the measured directory

- *NumIncModules – up*: number of files in directory linux-2.2.12/include/ linux/modules-up which have been directly included by files in the measured directory

- *NumIncModules*: number of files in directory linux-2.2.12/include/linux/ modules which have been directly included by files in the measured directory

- *NumIncNfsd*: number of files in directory linux-2.2.12/include/linux/nfsd which have been directly included by files in the measured directory

- *NumIncRaid*: number of files in directory linux-2.2.12/include/linux/raid which have been directly included by files in the measured directory

- *NumIncSunrpc*: number of files in directory linux-2.2.12/include/linux/ sunrpc which have been directly included by files in the measured directory

- *NumInc/.*: number of files in directory linux-2.2.12/include/linux/. which have been directly included by files in the measured directory

Table 5.12: Statistics for subsystems of linux-2.2.12 kernel (part 1)

measure	kernel	mm	fs	net	ipc
NumSrcFiles	20	15	313	256	4
NumIncLinux	49	18	146	103	10
NumIncAsm	9	3	23	16	2
NumIncNet	0	0	4	96	0
NumIncTotal	58	22	201	223	12
NumIncOut	1	0	8	3	0
NumNoIncFiles	0	1	13	5	0

Table 5.13: Statistics for subsystems of linux-2.2.12 kernel (part 2)

measure	kernel	mm	fs	net	ipc
NumIncLockd	0	0	7	0	0
NumIncNfsd	0	0	8	0	0
NumIncSunrpc	0	0	9	11	0
NumInc/.	49	18	122	92	10
NumIncTotal	49	18	146	103	10

For example only 1 file in linux-2.2.12/include/linux/byteorder has been used by files in linux-2.2.12/include/asm.

Next we try to figure out how many files in each of three directories under include have been directly used by the five subsystems of the kernel layer.

The measures displayed in this table are the same as those in table 5.10.

- *NumIncLockd*: number of files in directory linux-2.2.12/include/linux/lockd which have been directly included by files in the measured directory

- *NumIncNfsd*: number of files in directory linux-2.2.12/include/linux/nfsd which have been directly included by files in the measured directory

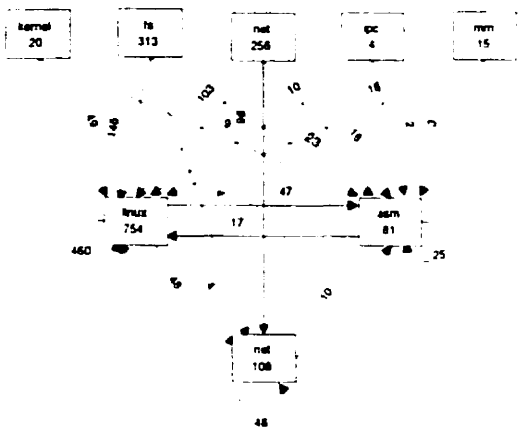


Figure 5.3: Dependence relationship of linux-2.2.12 subdirectories

- *NumIncSunrpc*: number of files in directory linux-2.2.12/include/linux/sunrpc which have been directly included by files in the measured directory
- *NumInc/.*: number of files in directory linux-2.2.12/include/linux/. which have been directly included by files in the measured directory
- *NumIncTotal*: number of files in directory linux-2.2.12/include/linux/ and all the other subdirectories of linux-2.2.12/include/linux/ which have been directly included by files in the measured directory

The graphical representation of relationships of directories is based on the statistics, shown in figure 5.3. For each directory D_i there is a number N_i associated, which gives the number of C source files or header files in directory D_i and its subdirectories. A weighted arc $D_i \rightarrow D_j$ represents the number of files in directory D_j which have been included by files in directory D_i.

Directories scsi and video have not been presented in this graph, since they are

```
#ifndef _FEATURES_H
#define _FEATURES_H 1
  CPP Context1
#ifndef __ASSEMBLER__
# include <sys/cdefs.h>
  CPP Context2
#endif /* !ASSEMBLER */
  CPP Context3
#endif /* features.h */
```

Figure 5.4: File features.h

```
#ifndef _SYS_CDEFS_H
#define _SYS_CDEFS_H 1
#ifndef _FEATURES_H
#include <features.h>
#endif
  CPP Context
#endif /* sys/cdefs.h */
```

Figure 5.5: File sys/cdefs.h

independent and have not been used by the kernel.

5.4 Detection of Cycle in Dependence Graph of Include Hierarchy

Dependency graphs of inclusion may consist of loops. While in the context of inclusion of files, detection of such loops is important. Loops may mean that there is a strong coupling relationship among these files. Graph algorithms were used to detect the cycles in the graph. Here is an example of files referring to each other which we detected during the analysis. The files are feature.h and <sys/cdefs.h>. Figure 5.4 gives code fragments of file features.h. Figure 5.5 gives code fragments of file sys/cdefs.h.

For file features.h. when preprocessor variable $\_\_ASSEMBLER\_\_$ is defined it will include sys/cdef.h. For file sys/cdefs.h, when $\_FEATURES\_H$ is not defined it will include features.h. When first including features.h, $\_FEATURES\_H$ will be defined. Then in sys/cdefs.h, since $\_FEATURES\_H$ has been defined, features.h will not be included again. But when first reaching sys/cdef.h, if $\_FEATURES\_H$ is not defined, features.h will be included. The presence of numerous conditional directives and include directives makes the structure almost impossible to follow. This calls for the further analysis of conditional directives.

5.5 Control Flow Graph

A control flow graph (CFG) consists of nodes, which represent single entry, single exit, regions of executable code and edges which represent possible execution branches between code regions. In this project, we focus on the preprocessor directives. CFG for preprocessing directives provides a graphic illustration of the use and complexity of conditional compilation in a program.

First, a complete control flow graph was generated for the source code. For each preprocessor directive there is one corresponding CFG node constructed. Also, each non-directive code line is kept as a CFG node for further analysis. Figure 5.6 lists the the code of <linux/config.h> in $linux - 2.2.3$ with node numbers printed. Figure 5.7 (a) is the corresponding graphical representation.

As we know, to be portable to different compilers and programming tools, the

```
 1 #ifndef _LINUX_CONFIG_H
 2 #define _LINUX_CONFIG_H
 3 #include <linux/autoconf.h>
 4 #ifndef UTS_SYSNAME
 5 #define UTS_SYSNAME "Linux"
   #endif
 6 #ifndef UTS_MACHINE
 7 #define UTS_MACHINE "unknown"
   #endif
 8 #ifndef UTS_NODENAME
 9 #define UTS_NODENAME "(none)"
   #endif
10 #ifndef UTS_DOMAINNAME
11 #define UTS_DOMAINNAME "(none)"
   #endif
12 #define DEF_INITSEG 0x9000
13 #define DEF_SYSSEG 0x1000
14 #define DEF_SETUPSEG 0x9020
15 #define DEF_SYSSIZE 0x7F00
16 #define NORMAL_VGA 0xffff
17 #define EXTENDED_VGA 0xfffe
18 #define ASK_VGA 0xfffd
   #endif
```

Figure 5.6: Source code of <linux/config.h>

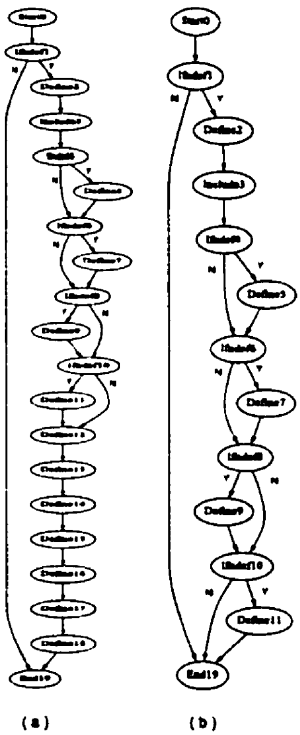


Figure 5.7: (a) Complete control flow graph of file <linux/config.h> in linux-2.2.3 (b)Reduced control flow graph of file <linux/config.h> in linux-2.2.3

conditional directives should be left- justified, that is the "#" character must be in the first column followed immediately by the conditional directive, as shown in figure5.6. This makes it to be difficult to read and understand. Control flow graphs help programmers see nested structures and relate blocks of code to the associated control statements.

The complete control flow graph of preprocessor directives keeps all the information of the code, which is far more than what we need to do conditional compilation analysis. This dramatically reduces the performance of the analysis procedure. In some situations, for example in analyzing code with presence of file inclusion, a reduced control flow graph may be sufficient for the included file. However, the complete control flow graph need to be reduced, sufficiently and safely. The rules are given in chapter 4.

Figure 5.7 (b) is the reduced control flow graph for file <linux/config.h> in linux-2.2.3.

It is worth noting that, in some cases, the reduced one is exactly the same as the complete control flow graph, e.g. header file <linux/config.h> in linux-2.2.12, with control flow graph shown in figure5.8, corresponding code listed in figure 3.2. Usually, these files are not complicated, with fewer nodes and fewer branch points.

In some cases, the benefits are significant. The reduced graph is much smaller compared to the complete one, as in header file <linux/linkage.h>. The complete control flow graph is shown in figure 5.9. There are 30 nodes, and 7 branch points,

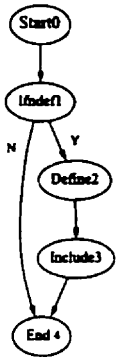


Figure 5.8: Control flow graph of file <linux/config.h> in linux-2.2.10, the complete one and reduced one are the same

deeply nested. In the reduced control flow graph (figure 5.10), four nodes and only one branch point are left.

5.6 Symbolic Execution of Preprocessing Directives

The symbolic executor traverses the CFG nodes stored in the node stack. Every node has a related macro constant table and a current condition constraint attached. The node stack is initialized with the start node of the tested CFG. Symbolic execution proceeds on each node popped from the node stack, constantly updating the modifications made to the tested unit's macro table, and decorating the corresponding path with the appropriate constraints encountered at branching points along the paths, expanding the node stack with the next possible node in the CFG according to the path selector.

Symbolic execution is time consuming. To improve the performance of the symbolic execution, the following procedures are performed before the symbolic

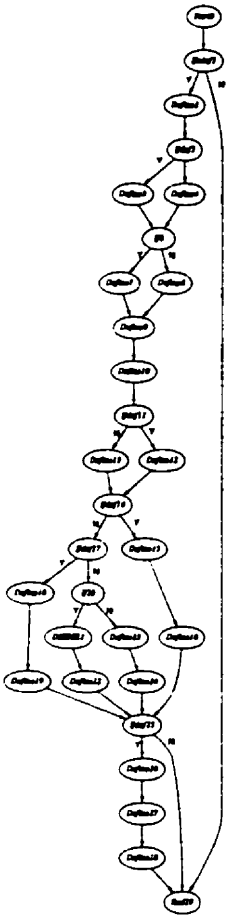


Figure 5.9: Complete control flow graph of file <linux/linkage.h> in linux-2.2.10

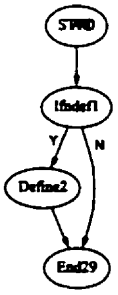


Figure 5.10: Reduced control flow graph of file <linux/linkage.h> in linux-2.2.10

execution phase.

- finding include hierarchy for the tested file
- identifying the preprocessor identifiers directly or indirectly used in the conditional directives for each file in the include hierarchy
- extracting the preprocessor variables served as safe guarders for each file in the include hierarchy
- undefining safe guarders at the beginning of the symbolic execution
- constructing the complete control flow graph for each file in the include hierarchy
- constructing the reduced control flow graph for each file in the include hierarchy, according to the classification of the preprocessor identifiers

Our two goals are: 1) for any given preprocessor directive line or C/C++ source code line, quickly finding one sufficient condition to reach/compile it; 2) finding the full condition to reach/compile that code line. To achieve these two goals, two strategies have been chosen, one is to find the shortest path to any given node by breadth-first expansion, the other is to find all possible conditions to a given node by depth-first expansion. Here <linux/kernel.h> of linux-2.2.12 was chosen as an example. The include hierarchy of <linux/kernel.h> was shown in figure 5.11.

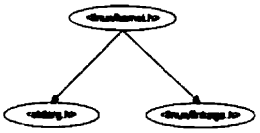


Figure 5.11: Include hierarchy of file <linux/kernel.h>

The include hierarchy graph stores the index of the files needed to be analyzed further. There are only three nodes in this graph. The top node is file <linux/kernel.h> itself. From this include hierarchy graph, one knows that <linux/kernel.h> may include <linux/linkage.h> and/or <stdarg.h>, but <linux/linkage.h> and <stdarg.h> do not include any other file. When one analyzes <linux/kernel.h>, only these three files need to be considered.

Shortest path to a given node

To find the shortest path to a given node, the output is sensitive to each execution step, thus the complete control flow information is required. With this strategy, complete control flow graphs are used for all the files in the include hierarchy. Two examples are given below.

Experiment 1: finding the condition for the first path from the starting node of <linux/kernel.h> to the 5th node of <linux/kernel.h>.

code: #include <linux/linkage.h>

path condition obtained is:

defined α_1

Symbol values and corresponding preprocessor variables are:

$\alpha_1 \equiv$ __KERNEL__

Experiment 2: finding the shortest path to the 18th node of $< linux/linkage.h >$.

code: $\#define \ \_\_ALIGN \ .align \ 4$

path condition obtained is:

$defined\ \alpha_8$ && ! $defined\ \alpha_7$ && ! $defined\ \alpha_6$&& ! ($defined\ \alpha_3$ && ($\alpha_4 > 2 || \alpha_5 > 7$))&& ! $defined\ \alpha_2$ && $defined\ \alpha_1$

symbol value table:

$\alpha_1 \equiv \_\_KERNEL\_\_$

$\alpha_2 \equiv \_\_cplusplus$

$\alpha_3 \equiv \_\_i386\_\_$

$\alpha_4 \equiv \_\_GNUC\_\_$

$\alpha_5 \equiv \_\_GNUC\_MINOR\_\_$

$\alpha_6 \equiv \_\_STDC\_\_$

$\alpha_7 \equiv \_\_arm\_\_$

$\alpha_8 \equiv \_\_mc68000\_\_$

The resulting condition of shortest path strategy is a boolean and-expression with symbolic values, each sub-expression is a constraint. The following metrics are very important to the analysis of constraints.

- *NumConstraint*: number of constraints (sub-expressions) in one path condition

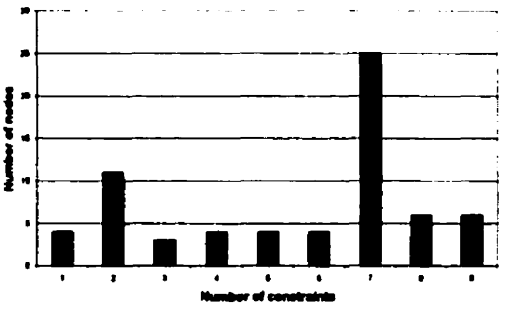


Figure 5.12: Statistic of constraints in shortest path condition for kernel.h

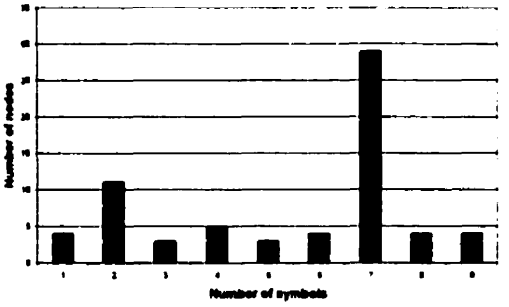


Figure 5.13: Statistic of symbols in shortest path condition for kernel.h

- *NumSymbol*: number of different symbols in one path condition

Figure 5.12 gives the statistics of number of constraints in shortest path condition for all 90 nodes in file <linux/kernel.h> and its two possibly included files. The average number of constraints per path condition is 4.44. Figure 5.12 gives the statistics of symbols in shortest path condition. The total CPU time for finding the shortest path to each of these 90 nodes is 3.46s.

The two other metrics that are also very interesting to programmers are:

- *NumIfBranch*: number of if-branches (#if,#ifdef,#ifndef) encountered a-

long the shortest path

- *NumIncludeDirective*: number of include directives (#include) encountered along the shortest path

Metric *NumIfBranch* differs from metric *NumConstraint*. In symbolic execution, once one encountering a conditional directive, if the predicate is true after evaluation, it will not contribute to the path condition expression; only if the predicate is undecided, it will become a sub-expression of the current and-expression. Thus metric *NumIfBranch* is always greater or equal to metric *NumConstraint*. The counting of *NumIncludeDirective* does not exclude multiple inclusion of the same file, the prevention of multiple inclusion is implemented by using safe guarders.

Next, experiments were conducted on all the header files under directory linux-2.2.12/include/linux. For each file f_i, one can get the set of all the files it may directly or indirectly include: $S_i = \{f_j\}$. For each file $f_j \in S_i$, there is a node set (preprocessor directive line and/or C/C++ ordinary code) associated: $N_j = \{n_k\}$. For each f_i, there are a set of nodes $NC_i = \sum_{f_j} N_j$ associated. The experiments were performed in the following procedures: first, for each file f_i, for every node in its corresponding closure set NC_i, the shortest path strategy was used to find one sufficient condition to this node, which consists in a path condition set PC_i; then, for each path condition in PC_i, the metrics *NumConstraint NumSymbol NumIfBranch* and *NumIncludeDirective* were calculated; finally, for each file,

the maximum value and average value of these four metrics were computed, and the distributions were drawn in figures.

Figure 5.14 and figure 5.15 give distributions of maximum and average value of these four metrics for individual files under directory linux-2.2.12/include/linux. Each square represents a file, with its vertical coordinate giving the value of corresponding metric.

Full condition to a given node

To find full condition to a given node, we use depth-first-search strategy. Each executing step itself is not important, but the final condition is what is needed. For the file the target node is in, the complete control flow graph is used. But for all the other files in the include hierarchy graph, the complete control flow graph is not necessary, a reduced one is sufficient.

Experiment 1: finding the path condition to code:

```
#define  FASTCALL  ( x ) x
```

which is the 25th node of $< linux/kernel.h >$.

path condition :

! defined α_2 && defined α_1

symbol value table:

$\alpha_1 \equiv$ __KERNEL__

$\alpha_2 \equiv$ __i386__

The condition is in its simplest form, so it does not need to be further simplified.

Experiment 2: finding the the path condition to

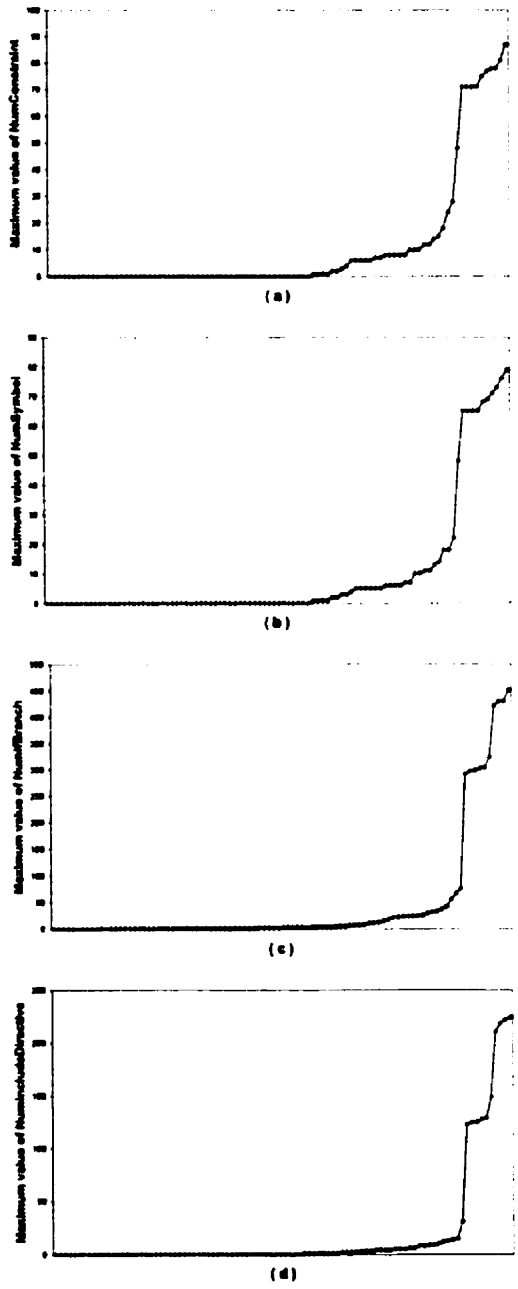


Figure 5.14: Distributions of maximum values of four metrics for individual files, sorted by increasing metrics values

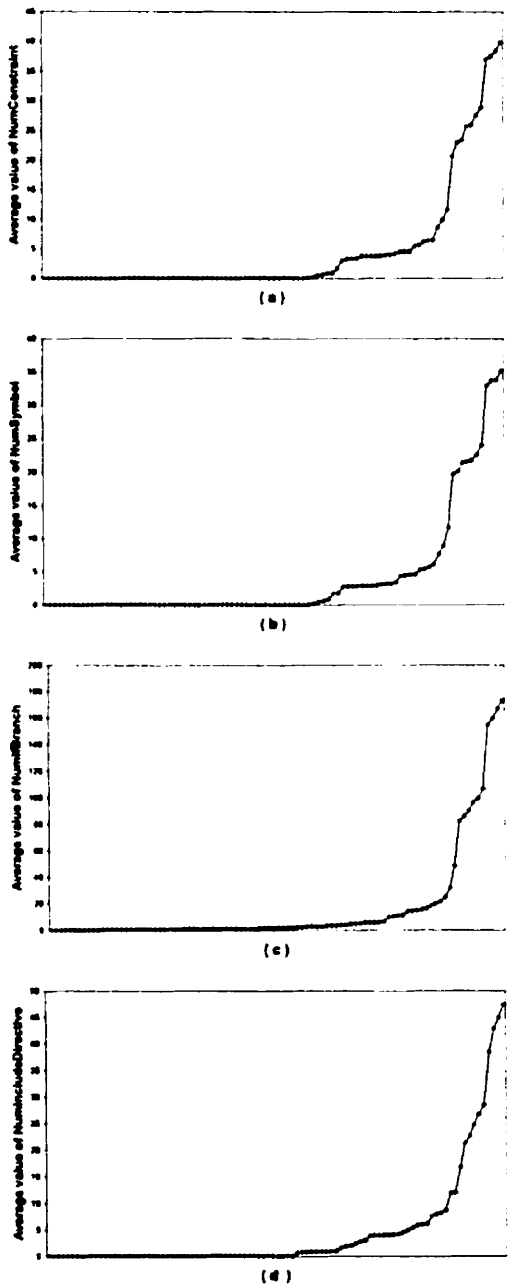


Figure 5.15: Distributions of average values of four metrics for individual files, sorted by increasing metrics values

#define _ALIGN .align 4

which is the 18th node of <linux/linkage.h>.

path condition :

$$
\begin{aligned}
&(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ !\ defined\ \alpha_6 \&\&\ !\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ !\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ defined\ \alpha_6 \&\&\ !\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ !\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ !\ defined\ \alpha_6 \&\&\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ !\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ !\ defined\ \alpha_6 \&\&\ !\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ !\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ defined\ \alpha_6 \&\&\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ !\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ defined\ \alpha_6 \&\&\ !\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ !\ defined\ \alpha_6 \&\&\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ ) \\
&\|(defined\ \alpha_8\ \&\&\ !\ defined\ \alpha_7\ \&\&\ defined\ \alpha_6 \&\&\ (defined\ \alpha_3 \\
&\&\&\ (\alpha_4 > 2 \| \alpha_5 > 7)) \&\&\ defined\ \alpha_2\ \&\&\ defined\ \alpha_1\ )
\end{aligned}
$$

symbol value table:

$\alpha_1 \equiv$ __KERNEL__

$\alpha_2 \equiv$ __cplusplus

$\alpha_3 \equiv$ __i386__

$\alpha_4 \equiv$ __GNUC__

$\alpha_5 \equiv$ __GNUC_MINOR__

$\alpha_6 \equiv$ __STDC__

$\alpha_7 \equiv$ __arm__

$\alpha_8 \equiv$ __mc68000__

The complete condition then was submitted to a path condition simplifier. After simplification, the final constraint was given. For example, the simplified expression for the above condition is:

defined α_8 && ! defined α_7 && defined α_1

The goal of finding the full condition to reach a tested code line seeks to cover all the combination of conditions, thus is of course time consuming. The performance can be largely improved by using the reduced CFG. Another way to tune the performance may be initializing some preprocessor variables in the beginning of the symbolic execution, based on the programmer's knowledge of programs.

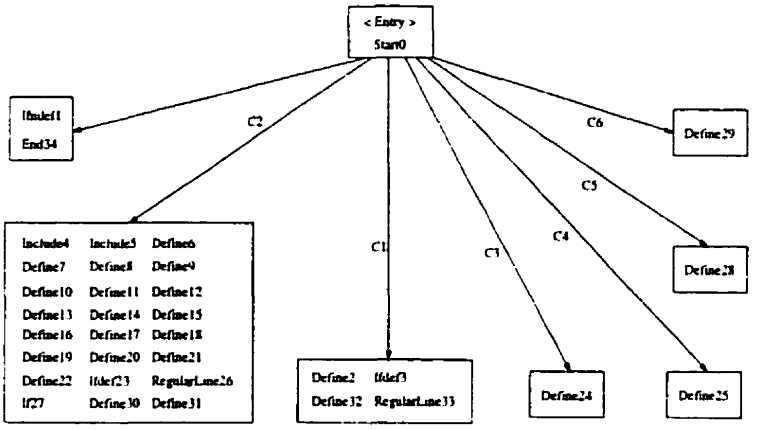


Figure 5.16: Compilation condition graph of kernel.h

5.7 Pre-condition

The path condition collected from symbolic executor gives the constraints of initial values. Here we use data flow analysis technique dominator tree, and build a tool which can easily collect the requirement of the final value of variables.

Figure 5.16 displays the compilation condition (intended condition or pre-condition) graph of header file kernel.h. The top node is the start node of kernel.h. Each CFG node corresponds to one code line. All the nodes which have the same compilation condition are grouped together and put in the same box. The condition is decorated on the directed edge from the root node to each box labeled as Cn. Empty condition (leftmost edge) refers to always true condition.

The value of each condition is listed in figure 5.17. Comparing this result with the one we got from symbolic executor, we can find that results are the same. Since in this concrete example, there is no redefinition of variables along the executing

```
C1: !defined _LINUX_KERNEL_H
C2: !defined _LINUX_KERNEL_H && defined __KERNEL__
C3: !defined _LINUX_KERNEL_H && defined __KERNEL__
    && defined __i386__
C4: !defined _LINUX_KERNEL_H && defined __KERNEL__
    && !defined __i386__
C5: !defined _LINUX_KERNEL_H && defined __KERNEL__ && DEBUG
C6: !defined _LINUX_KERNEL_H && defined __KERNEL__ && !(DEBUG)
```

Figure 5.17: Condition value table of kernel.h

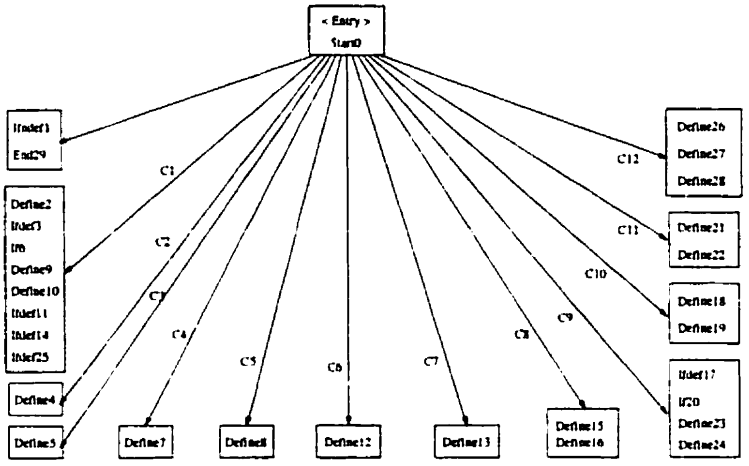


Figure 5.18: Compilation condition graph of linkage.h

path.

Figure 5.18 displays the compilation condition (intended condition or pre-condition) graph of header file linkage.h.

The value of each condition is listed in figure 5.19.

```
C1: !defined _LINUX_LINKAGE_H
C2: !defined _LINUX_LINKAGE_H && defined __cplusplus
C3: !defined _LINUX_LINKAGE_H && !defined __cplusplus
C4: !defined _LINUX_LINKAGE_H && defined __i386__
    &&(__GNUC__>2||__GNUC_MINOR__>7)
C5: !defined _LINUX_LINKAGE_H &&
    !(defined __i386__&&(__GNUC__>2||__GNUC_MINOR__>7))
C6: !defined _LINUX_LINKAGE_H && defined __STDC__
C7: !defined _LINUX_LINKAGE_H && !defined __STDC__
C8: !defined _LINUX_LINKAGE_H && defined __arm__
C9: !defined _LINUX_LINKAGE_H && !defined __arm__
C10: !defined _LINUX_LINKAGE_H && !defined __arm__
    && defined __mc68000__
C11: !defined _LINUX_LINKAGE_H && !defined __arm__
    && !defined __mc68000__ && !defined (__i486__)
    &&!defined (__i586__)
C12: !defined _LINUX_LINKAGE_H && defined __ASSEMBLY__
```

Figure 5.19: Condition value table of linkage.h

5.8 Summary

In this chapter we choose the linux kernel as a case study. We ran our tool and collected numerous useful information of preprocessor directives. As we know, in large system, file inclusion, conditional compilation and macro substitution are closely related and are often largely interleaved. In the examples we gave in this chapter, we demonstrated how to extract information and build text and graphic representation on three major components of preprocessor directives.

CHAPTER 6

CONCLUSION

Many of the problems related to software maintenance originate from the overall poor source code structure of large complex systems. The essence of the software maintenance problem may be ultimately traced back to the lack of sufficient understanding of the structure, functionality, characteristics and component dependencies in large software systems.

This thesis has reviewed program understanding techniques and efforts related to software maintenance in presence of numerous preprocessor directives. Within this framework, we have applied compiler techniques, control flow, data flow analysis, and symbolic execution techniques to identify parts of code that may be relevant to a particular maintenance task.

6.1 Contributions

This section summarizes the major contribution of this thesis.

Firstly, we use JavaCC and jjtree to build a CPP parser, which takes into account the full complexity of the preprocessor grammar. In this project we applied our CPP parser to numerous large C/C++ open source codes and succeeded in building Abstract Syntax Trees.

Secondly, we presented an approach and developed a tool to analyze the usage of macro variables.

Thirdly, we designed an analysis tool for include dependencies. This includes: 1) displaying the dependencies among include files in graphical forms, 2) inferring the conditions of such dependence relationships, 3) providing ways to group files and refine the structure of hierarchies.

Fourthly, we presented an approach and designed an analysis tool for conditional preprocessor directives. In the context of conditional compilation, the actual file compiled is only one of many possible versions of the source code, each determined by a particular setting of conditional compilation variables. The major function of this tool is to extract the compilation condition of arbitrary code line. In this project, we defined two types of conditions, one is defined on constraints of initial value for each macro variable, another is defined on constraints of final value for each macro variable. The first type of condition helps programmers to reuse the existing code, while the second type of condition helps programmers to understand the logic structure of the existing code.

We also defined series of metrics on code, variables, file components, and directories. The collection and analysis of such metrics provide a fast approximation of dependency analysis.

6.2 Discussion and Future Work

In many ways the results reported are very encouraging. Most of our experiments have illustrated the applicability and effectiveness of the proposed techniques in program analysis of preprocessor directives.

In this thesis we have tried to integrate some state-of-art techniques to provide generic mechanisms to analyze preprocessor directives.

The pattern matching approach is used to classify various usage of macro variables.

The control flow and data flow analysis techniques are quite efficient in obtaining compilation conditions represented in terms of final values of macro variables.

The symbolic execution technique is applicable in extracting the compilation condition represented in terms of initial values of macro variables.

With the metrics-based approach we have tried to represent the structure and the dependencies between high level software components.

Our experience with the proposed tool-set on large software system (such as linux) is that it can easily provide information on three major aspects of preprocessing.

Two major future research concerns are:

- increase the effectiveness of the symbolic executor for large systems
- add some visualization options

BIBLIOGRAPHIE

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-wesley, 1988.

[2] ANSI Standards X3-J4 COBOL Technical Committee *X3 Information Processing Systems Minutes.* Post Falls, ID. 1996.

[3] R.S. Arnold. *Software Reengineering.* IEEE Computer Society Press, Los Alamitos, California. 1990.

[4] K.R. Apt. *Analysis of CSP Programs.* In *Logics of Programs: Proceedings,* Springer Verlag, NY, 1983.

[5] D. Atkins, T. Ball, T. Graves, and A. Mockus. *Using version control data to evaluate the impact of software tools.* In *Proceedings of the 1999 IEEE International Conference on Software Engineering,*p.324-33, Los Angeles, California, May, 1999.

[6] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[7] P.D. Coward. *A review of software testing.* In *Software Engineering: A European Perspective,* IEEE Computer Society Press, Los Alamitos, 1993.

[8] B. Eckel. *Understanding inline functions in C++.* In *Embedded Systems Programming,* 7(4), p.64-6,68-74, April, 1994.

[9] M. Ernst. *An Empirical Analysis of C Preprocessor Use.* Technical Report UW-CSE-97-04-06, Dept. of Computer Science and Engineering, Univ. of Washington USA, 1997.

[10] D. Epstein. *Conditional compilation.* In *Dr. Dobbs Journal*, 21(5), p.44, 46, 80-3, May, 1996.

[11] J.M. Favre. *Reengineering-In-The-Large vs Reengineering-In-The-Small.* In *first SEI Workshop on Software Reengineering*, Software Engineering Institute, Carnegie Mellon University, May 1994.

[12] J.M. Favre. *Preprocessors from an Abstract Point of View.* In *Proceedings of the 1996 IEEE International Conference on Software Maintenance*, California, Nov. 1996.

[13] J.M. Favre. *A rigorous approach to support the maintenance of large portable software.* In *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*, Berlin, Germany, March 1997.

[14] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[15] D. Garlan, and D.E. Perry. *Introduction to the Special Issure on Software Artechiture.* In *IEEE Trans. Softw. Eng.*, 21(4), 269-274, Apr. 1995.

[16] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading MA, 1995.

[17] R. Getov. *Not-so-obvious utility macros.* In *C/C++ Users Journal*, 18(3), 27, p. 30-4, 36, 38-40, March, 2000.

[18] S.P. Harbison, and G.L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1995.

[19] ISO/IEC-JTC1/SC22 *FORTRAN 95.* Draft International Standard IS 1539.

[20] R. Jaeschke. *Portability and the 'C' Language.* Hayden Books, Indianapolis, IN, 1989.

[21] K. Jameson. *Multi-Platform Code Management.* O'Reilly & Associates, Sebastopol, CA, 1994.

[22] Sun Microsystems Inc. Jdk 1.2: Java development kit.

[23] B.W. Kernighan, and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[24] M. Krone, and G. Snelting. *On the Inference of Configuration Structures from Source Code.* In *Proc. 16th International Conference on Software Engineering*, Sorrento, Italy, 1994.

[25] B. Lague, C. Leduc, E. Merlo, and M. Dagenais. *A Framework for the Analysis of Layered Software Architectures.* In *WESS*, Bary, Italy, 1997.

[26] B. Lague, and C. Leduc. *Assessment of the Partitioning of Large OO design in Files Using Metrics.* In *OOPSLA Workshop 97*, 1997.

[27] B. Lague, C. Leduc, A. Le Bon, E. Merlo, and M. Dagenais. *An Analysis Framework for Understanding Layered Software Architectures.* In *IEEE Workshop on Program Comprehension*,1998.

[28] M.M. Lehman, and L.A. Belady. *Program Evolution.* Academic Press, London and New York, 1985.

[29] T.C. Lethbridge, and N. Anquetil. *Architecture of a Source Code Exploration Tool: A Software Engineering Case Study.* Technical Report TR-97-07, University of Ottawa, Computer Science, December, 1997.

[30] P. Linos, P. Aubet, and L. Dumas. *CARE: An Environment for Understanding and Re-engineering C Programs.* In *Proceedings of the 1993 IEEE Conference on Software Maintenance*, Montreal, Quebec, September, 1993.

[31] P.E. Livadas,and D.T. Small. *Understanding Code Containing Prepreocessor Construct.* In *IEEE Third Workshop on Program Comprehension*, Washington, November 1994.

[32] Metamata,Inc. *Java Compiler Compiler(JavaCC). URL: http://www.metamata.com/JavaCC*, May,1999.

[33] G.C. Murphy, D. Notkin, and E. Lan. *An empirical study of static call graph extractors.* In *Proceedings of the 18th International Conference on Software Engineering*, March, 1996.

[34] G. Naumovich, G.S. Avrunin, and L.A. Clarke. *Data Flow Analysis for Checking Properties of Concurrent Java Programs.* In *Proceedings of the 1999 IEEE International Conference on Software Engineering*, Los Angeles, California, May, 1999.

[35] T.T. Pearse and P.W. Oman. *Experiences Developing and Maintaining Software in a Multi-Platform Environment.* In *Proceedings of the 1997 IEEE International Conference on Software Maintenance*, Italy, Oct. 1997.

[36] M. Platoff, M. Wagner, and J. Camaratta. *An integrated program representation and toolkit for the maintenance of C programs.* In *Proceedings of the 1997 IEEE International Conference on Software Maintenance*, Sorrento, Italy, October, 1991.

[37] M. Siff, and T. Reps. *From C to C++.* In *Proceedings of SIGSOFT'96 Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October, 1996.

[38] G. Snelting. *Reengineering of configurations based on mathematical concept analysis.* In *ACM Transactions on Software Engineering and Methodology*, 5(2): 146-189, April,1996.

[39] S.S. Somé and T.C. Lethbridge. *Minimizing Parsing when Extracting Information from Code in the Presence of Conditional Compilation.* In *Proceedings 6th International Workshop on Program Comprehension,*Ischia, Italy, June 1998.

[40] Software Productivity Consortium *Ada '83 Guidelines for Professional Programmers.* Software Productivity Consortium, Herndon, VA, 1992.

[41] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX*, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pages 185-197

[42] D. Spuler, A.S.M. Sajeev. *Static Detection of Preprocessor Macro Erros in C.* In *Technical report 92-7* James Crook University, 1992, 18 pages.

[43] D. Spuler. *C++ and C Debugging, Testing and Reliablity.* Prentice Hall, Englewood Cliffs, NJ, 1994

[44] B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, Massachusetts, 1994.

[45] G. Succi, and R.W. Wong. The application of JavaCC to develop a C/C++ preprocessor. In *Applied Computing Review*, vol.7, no.3, p. 11-18, 1999.

[46] T. Tourwé, and W.D. Meuter. Optimizing Object-Oriented Languages Through Architectural Transformations. In *LNCS 1575 Compiler Construction*, pp. 244-258, 1999.

[47] W.M. Turski. *Reference Model for Smooth Growth of Software Systems.* In *IEEE Transactions on Software Engineering*, Vol. 22, No.8,pp. 599-600, Aug. 1996.

[48] K.P. Vo, and Y.F. Chen. *Incl: A Tool to Analyze Include Files.* In *USENIX*, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pages 199-208.

[49] P.T. Zellweger. *An interactive high-level debugger for control-flow optimized programs.* In *Technical Report CSL-83-1*, Xerox Palo Alto Research Center, Palo Alto, Califomia, January 1983.