ソフトウェア工学 100－9
（1994. 9. 8）

C言語プログラム検査ツールの実装と適用結果

青木 圭子 瀧塚 孝志 橋本 和夫 小花 貞夫

国際電信電話株式会社 研究所

近年，ANSI標準規格準拠のC言語を用いたソフトウェア開発が普及しつつあり，様々な計算機環境でANSI C言語コンパイラが利用可能となっている．筆者らは，ANSI C言語で書かれたプログラムの再利用を促進するため，C言語プログラム検査ツール(ALINT)を実装した．本ツールは，i) コンパイラ毎に異なる検査項目を一つのツールで総合的にかつ効率的に検査する，ii) 保守性や移植性を向上させるために重要な，誤りやすい記述や読みにくい記述等に関する豊富な検査機能を持つ，iii) 警告の選択的出力や不要な警告の抑制方法を提供する，などの特徴を持つ。本稿では，本ツールの設計と実装の概要，および実際の通信処理ソフトウエアへの適用を通じた評価結果について述べる．

Implementation and Evaluation of C Language Program Check Tool: *ALINT*

Keiko AOKI Takashi TAKIZUKA Kazuo HASHIMOTO
Sadao OBANA

KDD R & D Laboratories
2-1-15, Ohara, Kamifukuoka-shi, Saitama 356, JAPAN

As the benefit of ANSI C in software system development is recognized, compilers for ANSI C are becoming popular, and commercially available on various environments. Authors implemented a C language program check tool (ALINT) in the hope of promoting the reuse of ANSI C programs. ALINT is characterized by the following features; i) able to conduct a comprehensive checking efficiently, ii) has a variety of checking functions to detect misleading or hard-to-read descriptions, iii) has a variety of check options to produce warnings selectively, and able to suppress warnings selectively. This paper describes the design and implementation of ALINT, and reports the result when an existing practical telecommunication software is checked by ALINT.

1. はじめに

プログラムの保守性や移植性は，プログラムの記述方法に大きく依存する．例えば，プログラム内で同じ名前の変数を2回以上宣言することは，誤りの原因になる．また，優先順位の異なる演算子が複数用いられている場合には，読みにくくなる．さらに，同じプログラムであっても，計算機やコンパイラに依存してエラーが出たり出なかったりする記述があり，コンパイラ間の移植性に問題がある．

この様な誤りやすい記述，読みにくい記述，計算機環境に依存する記述を検査し排除することにより，バグの数を減らしたり，プログラムを読みやすくして保守の効率を向上させたり，移植性を向上させることができる．

現在，通信処理ソフトウェアの多くは，C言語を用いて開発されている．C言語の中でも移植性を向上させる観点から，GCC[1]やMS-C[2]，Turbo C[3]，Lattice C[4]等のANSI標準規格[5][6]準拠のC言語コンパイラが多くの計算機で利用可能となりつつある．これらのANSI C言語コンパイラでは，厳密な型の定義や関数のプロトタイプ宣言等により，従来のK&RのC[7]よりも厳密な型検査を行うことができるものの，上記で述べた誤りやすい記述，読みにくい記述，計算機環境に依存する記述等に関する検査能力は十分ではない．さらに現状では，各種ANSI Cコンパイラ毎に検査内容が異なるため，プログラムの検査も計算機環境に依存して行なわなければならない．

また，これらを補うための検査ツールとして，UNIX上のlint [8]やPSS(C言語プログラムの移植支援システム) [9][10][11]等の検査ツールがあるが，ANSI C言語を対象としたプログラム検査ツールについては報告されていない．

このようなプログラム検査における問題を解決するため，筆者らは，様々なANSI C言語コンパイラを用いて開発されたプログラムを，総合的に検査するためのC言語プログラム検査ツール[12][13][14] (ALINT)を開発した．本稿では，本ツールの設計と実装の概要，および実際の通信処理ソフトウエアを用いた評価結果について述べる．

2. 既存プログラム検査ツールの問題点

2.1 プログラム検査ツールの位置付け

コンパイラは，それぞれの計算機環境での実行に問題のある記述に対する警告を行なうが，1行の文字数が異なるなどの計算機環境間の移植に関して生じる問題や，保守性を妨げる誤りやすい記述や読みにくい記述を検出する問題に対しては，十分な検査を行なうものは少ない．プログラム検査ツールは，このようなコンパイラの検査機能を補うためのツールであり，通常，コンパイラの検査を終えたソースプログラムを対象に検査を行なう．

2.2 既存ツールの問題点

C言語プログラムの検査ツールとしては，UNIX上のlint[8]やPSS(C言語プログラムの移植支援システム)[9][10][11]等がある．

lintは，UNIX上のプログラム検査ツールで，移植性の検査のための様々なヒューリスティックスを取り入れて，関数の引数や戻り値の型，未定義値の使用，未使用変数， 意味のない文等の検査を行う．

PSSは，矛盾指摘チェッカと呼ばれるモジュールが，通常のコンパイラ並みの文法検査やlint相当の検査の他，移植元準拠文法と移植先準拠文法の相違による問題，バイトオーダ依存部の指摘，アクセス境界制約違反等の検査を行う．

しかしながら，これらの既存の検査ツールには以下のような問題点がある．

問題1 ANSI Cを対象とした検査ツールではない．

問題2 コンパイラ毎に検査項目が異り，しかもこれらをすべて包含する検査ツールがないため，総合的なプログラム検査を効率的に行うことができない．また，コンパイラ間の移植性の問題ともなる．

問題3 ソフトウエアの保守性に重大な影響を与える，読みにくい記述や誤りやすい記述等について，検査機能が不足している．

問題4-1 警告を選択的に出力する方法が提供されていないため，多量の警告が検出されるプログラムでは，検査目的毎の警告の分析が難しい．

問題4-2 通信処理ソフトウェアの作成においては，既存のプログラムを改修することが多いため，改修したプログラムの検査の際には，既に書かれている部分に対する警告数を抑える機能がない．

3. ALINTの設計と実装

3.1 設計方針

C言語検査ツール(ALINT)では，**2.** で述べた問題を解決するため，以下の設計方針を採用することとした．

方針1(問題1に対応) ANSI Cのプログラムを前提とするものとし，ANSI Cに違反する記述に対して警告を行う． あわせて，古い形式のCとANSI Cで扱いの異なる記述に対しても警告を行う．また，ANSI Cの方言の一つであるVAX/Cのプログラムについても検査できるようにする．

方針 2(問題 2 に対応) 関数の引数にポインタ型でない構造体がきたときの警告や，関数スコープで最終的に定義の無い struct 宣言等，各種コンパイラの持つ検査機能をすべて包含させることにより，コンパイラによって警告が出たり出なかったりする問題を解決する．

方針 3(問題 3 に対応) 無限ループの検出などの読みにくい記述や誤りやすい記述の検査に加えて，1 行の文字数や文字列定数の長さなどのコンパイラによって制限のある値の検査を行うため，従来のコンパイラやツールに実装されていなかった検査項目を新たに拡張する．

方針 4(問題 4-1 に対応) 検査目的に合わせて警告の出力を制御可能とするため，各種検査項目をオプションによって個別に指定できるようにする．また，検査オプションの標準的な組み合わせをいくつかのレベルのプロファイルとして提供する．

方針 5(問題 4-2 に対応) 必要に応じてヒューリスティックに警告数を抑制する機能を提供する．

3.2 ソフトウェア構成

本ツールは，VAXStation 3100 (VAX/VMS Version 5.5-2) 上で動作する．ソフトウェア構成は，図 1に示すように，プログラム検査を行うためのプリプロセッサ，字句解析，構文解析，意味解析の各モジュールと，その検査結果を検査目的に応じて文書化するためのドキュメント生成，集計の各モジュールから構成した．各モジュールの処理概要を以下に示す．

(1) プログラム検査用モジュール

(i) プリプロセッサ
入力ファイルからデータを取り込み、処理のできる形式に変換し、#pragma や#include 等の処理系に応じた処理を行う．

(ii) 字句解析モジュール
文字定数等の字句的な検査を行う．

(iii) 構文解析モジュール
型検査や括弧の有無等の構文的な検査を行う．

(iv) 意味解析モジュール
未初期化変数の使用やローカル変数の 2 重定義等意味的な検査を行う．

(2) 検査結果文書化用モジュール

(i) ドキュメント生成モジュール
上記検査モジュールから生成される検査結果を用いて，プログラム中で宣言されている全ての関数や変数の使用状況 (変数や関数の名前や型の情報，定義，参照，代入の区別) をファイル中の行番号とともに相互参照情報として出力し，さらにその情報を用いて，変数や関数の相互参照表を作成する．

(ii) 集計モジュール
検査モジュールの出力結果から、指定した特定の警告メッセージを消すフィルター機能や，メッセージ毎の個数をカウントする集計機能を持つ．

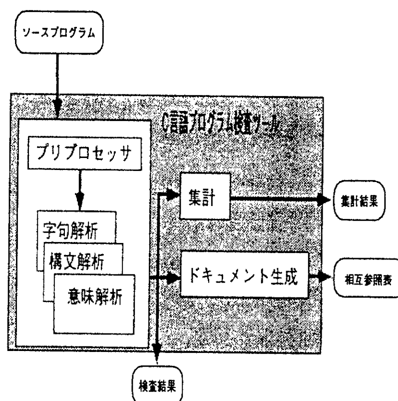

図 1: ALINT のソフトウェア構成

3.3 検査項目とその指定

3.1 で述べた設計方針 1，2，3 に従って，ANSI C で記述されたプログラムを対象とし，各種コンパイラや検査ツールの検査機能を包含し，無限ループの検出等の独自の検査機能も実現するための検査項目を設定した．

表 1に主な検査項目を示す．これらの検査項目は，検査目的別に以下の 5 つに分類される．表中で下線で示した検査項目は，本ツールで新たに追加した項目である．また，関連する検査項目を検査オプションとしてグループ化し，検査項目の指定単位とした．

これらは，各種コンパイラの検査項目を含んでいる．例えば，オプション 27 の#if または#elif の式中の未定義のマクロ名に関する検査は，VAX/C では行うが，GCC では行わない．逆に，オプション 5 の演算子の優先順位が複雑なときの警告は GCC では行うが，VAX/C では行わない．

(1) 必須 (デフォルト) の検査
検査オプションの指定にかかわらず，unsigned int

表 1: 主な検査項目

検査目的	検査内容	option	level
必須 (デフォルト) の検査	unsigned int 型でないビットフィールド		
	static でない関数の static 宣言		
	異なるポインタ間の代入		
	ポインタ型と int 型の比較		
	左辺値のキャスト		
読みにくい記述, 誤りやすい記述	コメント中の '/*'	1	1
	インライン関数がインラインにならなかった場合	2	1
	矛盾した宣言	3	1
	型検査	3	1
	条件式の主たる演算子が代入演算子であった場合	3	1
	enum 型の要素がカンマで終わっていた場合	3	1
	実行文を含む構造体変数の初期化	3	1
	キーワードの後のコメント	3	1
	always true/false になるような場合	4	2
	演算子の優先順位が複雑なとき	5	2
	暗黙の引数の型宣言	6	2
	定数のシフト値が型のサイズより大きい場合	6	2
	< *stdio.h* >で規定される~print~, ~scan~の書式	7	2
	括弧が足りない場合	8	2
	無限ループ	9	2
	到達しない文	9	2
	暗黙の戻り値の宣言	10	2
	関数スコープで未初期化変数を用いた場合	11	2
	未初期化変数の警告	11	2
	意味の無い文	12	2
	異なる enum に対して演算を行った場合	13	3
	char 型の配列インデックス	14	3
	暗黙の int 関数	15	3
	プロトタイプ宣言のない関数	16	3
	case ラベルに使用されていない enum の各値	17	3
	関数スコープで初期化のない struct 宣言	18	3
	使われてない変数	18	3
	キャストによって意味が変わる場合	20	4
	関数呼び出しでプロトタイプ宣言と異なる引数を渡した場合	21	4
	関数スコープで最終的に定義の無い struct 宣言	22	4
	void 型のサイズ演算	23	4
	冗長な宣言	24	4
	シャドーイング	25	4
	引数の型を省略したプロトタイプ宣言	26	4
	#if または#elif の式中の未定義のマクロ名	27	4
	extern で宣言された変数が extern なしで宣言された場合	28	4
	ポインタがアライメントが増える方向にキャストされた場合	30	5
	auto 構造体変数に参照または更新がない場合	34	5
ANSI 以前の C と ANSI C で 扱いの異なるもの	'\n' を用いた実引数の連結	19	3
	define 文中の"/* */"	27	4
	整定数の符号	27	4
	'\x' の扱い	27	4
	構造体や共用体を返す関数の定義や呼び出し	29	5
	関数の内部で extern 宣言された変数	31	5
	トライグラフ	35	5
	'\' を用いた継続行	36	5
	#や##	36	5
	明示的なサイズを持つ文字列配列の初期化	36	5
	union の初期化	36	5
	#error	36	5
移植性	レジスタ変数や定数のアドレス演算	3	1
	配列のインデックスが 0 以下の定数の場合	32	5
	定数のシフト値が負の場合	32	5
	外部シンボル名の最大長の検査	36	5
	VAX/C 特有の記述	36	5
	1 行の文字数	36	5
	文字列定数の最大長	36	5
	識別子の最大長	36	5
	マクロの引数の最大の個数	36	5
	マクロ呼び出しの最大の深さ	36	5
	インクルードファイルの最大の呼び出しの深さ	36	5
	関数呼び出しの引数の最大の個数	36	5
	初期化リストの {} の最大の深さ	36	5
その他	enum の識別子が同じ値を持っているとき	32	5
	auto の構造体変数の初期化	33	5
	未定義のマクロの#undef	33	5
	引数が参照されていない#define 文	33	5

型でないビットフィールドの検査やstaticでない関数のstatic宣言等のC言語プログラムとしての基本的な検査を必ず行なう.

(2) 読みにくい記述, 誤りやすい記述
複雑な演算子の優先順位や条件式の中の代入文, 異なるポインタ型同士の代入や未初期化変数の使用等は, 読みにくい記述, 誤りやすい記述として警告する.

(3) ANSI以前のCとANSI Cで扱いの異なる記述
識別子の通用範囲等, ANSI以前のCとANSI Cで扱いの異なる記述について警告を行う.

(4) 移植性
ポインタのアライメントのように, ハードウェアによって扱いの異なる記述や, マクロ展開の深さ, 初期化リストの大きさのような, コンパイラによって制限のある値を, 移植性に問題のある記述として警告を行なう.

(5) その他
上記の分類に入らない項目で, enumの識別子が同じ値を持っているときやauto構造体変数の初期化等の詳細な検査を行なう. また, 処理系に依存した処理をpragmaで指定するという, ANSI Cの規約に基づき, #pragma standardと#pragma nostandardによって, 処理系に依存した前処理の領域指定を行うこととした. #pragma standardと#pragma nostandardの間で指定されている領域では, ANSI C以外の記述に対して警告を行う.

3.4 入出力例

本ツールの入出力の例として, インデックスが0以下の配列と識別子が同じ値を持っているenum型の変数が宣言されているプログラムを本ツールにかけたときのプログラムと入出力例を図3.4, 3.4, 3.4に示す.

```
1 void func(void){
2   char str[-1];
3   enum e { A = 1, B = 1};
4 }
```

図2: プログラム例

```
$ alint/warning=portability file_name.c
```

図3: 入力コマンド例

図3.4のプログラムの左の番号はプログラムの行番号を示す. 図3.4の/warning=portabilityは表2の検査オプション32を指定している. 図3.4が出力結果の例で

```
file_name.c: In function 'func':
file_name.c:2: size of array 'str' is negative
file_name.c:3: warning: 'B' has a duplicate enum value with 'A'
```

図4: 出力例

あり, ファイル名(file_name.c)の次の数字がプログラム中の行番号を示す. ここでは2行目でインデックスが0以下の配列に関する警告, 3行目で識別子が同じ値を持つenumに関する警告を示している.

3.5 警告出力の制御

3.1で述べた方針4, 5に従って, 本ツールでは警告出力の制御機能として, 警告の選択出力機能と警告の抑制機能を提供する. 以下にその詳細を示す.

3.5.1 警告の選択出力機能と警告のレベル分け

表1に示した検査オプションは個別に設定, 解除ができる. また, ユーザの用途や目的に合わせて検査オプションの組合わせを選択できるように, 5つのレベルの標準プロファイルを設けた.

表1のレベル欄は, 検査オプションが含まれるプロファイルを示す. 高いレベルのプロファイルは, 低いレベルのプロファイルに含まれる検査オプションを全て含み, また低いレベルのプロファイルを指定した場合でも, 高いレベルのプロファイルに含まれる検査オプションを個別に指定できるようにした.

プロファイルと検査オプションは, ツール起動時に指定する. 以下に各レベルのプロファイルで行う主な検査内容を示す.

レベル1
矛盾した宣言がないかの検査や型検査等のプログラム記述上の基本的な検査.

レベル2
条件式が常に真または偽にならないか, 定数のシフト値が型のサイズより大きいくならないか等, 意味的に誤りと思われる記述の検査.

レベル3
プロトタイプ宣言の有無やenum型が正しく用いられているか等の検査.

レベル4
使われてない変数や同じ名前の変数を2回以上定義した場合等, ANSI Cの規約を厳密に遵守した検査.

レベル5
識別子の最大長等の移植性に関する検査や配列のインデックスが0の場合等の検査.

3.5.2 警告の抑制機能

既存のプログラムに対する警告数を抑えるため，以下により警告の抑制を行った．このうち (1)～(5) の警告の抑制は，起動時のオプション指定により行い，(6) は常に適用することとした．

(1) 型検査の緩和

(i) 図5のような構造体のポインタの代入の際，構造体の先頭の型同士が等しければ警告を出さない．

```
struct HEADER { int class_def;} *universal;
struct { struct { struct {
    struct HEADER h; int i;} s_1; } s_2;
} *instance;
universal = instance;
```

図 5: 構造体のポインタ代入の例

(ii) ANSI以前のCでは，void*型がなかったため，例えば従来のK&RのC[7]で書かれたソースではchar*がvoid*の代りに多く使用されている．そこで，char*型をvoid*型と見なす．

(iii) long型とポインタ型間の代入がよく行われているため，long型とポインタ型の間の代入を許す．

(2) signedと，unsignedの違いを無視する．

(3) auto構造体の実行文による初期化を無視する．

(4) いくつかのコンパイラでは，enum型のビットフィールドが使用できるため，enum型のビットフィールドに対して警告を出さない．

(5) システムパス名の一覧を指定できるようにし，指定されたパス中のインクルードファイルについては警告を出さない．

(6) #pragma standardと#pragma nostandardによって領域指定された部分において，テキストライブラリや#dictionary等のVAX/C特有の記述に対する警告は各々最大5回までとする．

4. 評価と考察

既に稼働している通信処理ソフトウェアPを対象として，本ツールの検査オプション毎，プロファイルのレベル毎の警告の数等を調べた．ここで，Pは25個のファイルから成る約20K行のプログラムである．このプログラムは，本ツールを用いて検査することを考慮して作成されたものではない．

4.1 検査オプション別の検査結果

各検査オプションを個別に指定して検査を行なった時の，検査オプション別の警告数を表2に示す．ここでは3.5.2の警告抑制のためのオプション指定を行う前と，すべて指定した場合の各々について示している．

その結果，Pについては引数の型を省略したプロトタイプ宣言 (検査オプション26) やポインタがアライメントの増える方向にキャストされた場合 (検査オプション30) や暗黙のint関数に関する警告 (検査オプション15) が多く出ている．

表 2: 検査オプション別の警告数

オプション	抑制前	抑制後	レベル
1	23	23	1
2	0	0	1
3	0	0	1
4	0	0	2
5	0	0	2
6	0	0	2
7	20	20	2
8	0	0	2
9	0	0	2
10	36	36	2
11	12	0	2
12	0	0	2
13	0	0	3
14	0	0	3
15	209	209	3
16	173	173	3
17	0	0	3
18	35	35	3
19	3	3	3
20	0	0	4
21	570	24	4
22	0	0	4
23	0	0	4
24	175	175	4
25	10	10	4
26	4539	4539	4
27	206	206	4
28	22	22	4
29	152	152	5
30	316	0	5
31	0	0	5
32	19	19	5
33	23	23	5
34	0	0	5
35	0	0	5
36	0	0	5

4.2 レベル分けの評価

Pについて，3.5.2の警告抑制をすべて指定し，各レベルのプロファイルを指定した場合のレベル毎の警告数

を表3に示す．

警告のプロファイルによってレベル分けしたことによって，既存のプログラム等で警告を出したくない場合には低いレベル，開発中のプログラムで細かい警告が欲しい場合には高いレベルを用いることによって使い分けることができるようになった．このことから，今回設定した各プロファイルは以下に示す目的で使用する場合に有効と考えられる．

レベル1
既存のプログラムを改修する等，警告をあまり出したくない場合．

レベル2
プログラム開発の初期の段階で，暗黙の引数の型宣言等の誤りである可能性の高い記述を検査する場合．

レベル3
関数スコープで最終的に定義のないstruct宣言のような，ANSI Cのプログラムを検査する場合．

レベル4
キャストによって意味の変わる場合や関数呼び出しでプロトタイプ宣言と異なる引数を渡した場合等，ANSI Cの規約を厳密に遵守する場合．

レベル5
文字列定数の最大長や識別子の最大長等，あまり頻繁には行われない16ビットパソコンへの移植を考えた検査を行いたい場合や，enumの識別子が同じ値を持つ場合等，特に厳密な検査を行いたい場合．

表3: レベル別の警告数

レベル	1	2	3	4	5
P	23	79	499	5465	5659

また，表2で示される検査目的の分類と，警告のレベルとは異なる基準で分類されるため，同じ検査目的に分類される検査内容でも，いろいろなレベルがある．例えば「読みにくい記述，誤りやすい記述」の検査目的には，1～5のレベルが全て含まれている．

4.3 警告抑制の効果

Pのプログラムを対象に，表2の結果から抑制の効果の高かった検査オプション30を指定し，警告抑制のための各オプションを個別に指定した場合と全て指定した場合の警告の抑制の効果を表4に示す．表中の抑制オプションの番号は，3.5.2の(1)～(5)に該当する．

検査オプション30に関しては(1-ii)の型検査の緩和(char*型をvoid*型と見なす)に効果があった．

表4: 警告抑制の効果

抑制オプション	警告数
(1-i)	0
(1-ii)	316
(1-iii)	0
(2)	0
(3)	0
(4)	0
(5)	0
全て指定	316

4.4 OS，コンパイラの違い等に対する対処

(1) コンパイラによって扱いの異なる記述
同じ記述でも，コンパイラによって警告が出たり出なかったりする場合，ALINTでは警告を出すこととした．

(2) 配列サイズの記述
コンパイラによってサイズが0の配列の記述方法が異なる．例えばVAX/Cでは，図6の1行目のようにサイズを指定しない配列は，サイズが0として扱われるが，2行目のようにサイズを0として指定するとエラーになる．一方，GCCでは，structの中でサイズを指定しない配列はエラー，サイズを0として指定すると警告となっている．これらの

```
char    buffer[]; /* サイズ指定なし */
char    buffer[0]; /* サイズ指定0 */
```

図6: サイズを指定しない配列の例

問題点を解決する方法として，structの中でサイズが0の配列を宣言するときには，サイズを指定しないようにし，警告を出さないようにすることが必要である．

(3) 行末処理
エディタの1行の表し方は，OSによって異なる．例えばUNIX上では行末を表すNLは，VMS上では行末を表す文字ではなく，エディタでファイルを作成する際に，文字(列)定数中等に直接書ける．本ツールでは，プリプロセッサがファイルを読むとき，行の終りにNLを付け，フラットな構造にしているため，行中のNLか，行末のNLか区別が付かず，警告メッセージや行番号が誤って出力されることがある．そこで，可変長レコードファイルに対しては，プリプロセッサで，行中のNLを警告するとともに，文字定数や文字列定数内のNLを'\n'に変換し，その他のNLを'\t'に変換することで対処した．

(4) 警告を避けられない記述
通常では警告が出る記述のうち，利用者が意図的に記述する場合の多い記述に関しては，警告を回避するための記述を提供する必要があった．例えば，レベル１の条件式の主たる演算子が代入演算子であった場合の検査においては，$if(i=1)$ と書いた場合には，通常 $if(i==1)$ の誤りである可能性があるため警告を行うが，意図的に代入文を書きたい場合には，さらに () を追加して $if((i=1))$ と書くと警告を行わないようにした．

5. おわりに

本稿では，C 言語プログラムの再利用を促進するための，C 言語プログラム検査ツールの設計と実装について述べた．本ツールは，ANSI C 言語を対象とするプログラム検査ツールであり，以下の特徴を持つ．

(1) コンパイラ毎に異る検査内容を１つのツールで効率的に検査する．
(2) 保守性や移植性を向上させるために重要な，誤りやすい記述や読みにくい記述等に関する検査能力を持つ．
(3) 警告の選択的出力や，不要な警告の抑制方法を提供する．

さらに，実際の通信プログラムへの適用を通じて，本ツールの実用性を確認した．また，本ツールは，様々な検査項目に関して選択的にプログラムの検査を行うことができるため，プログラムの品質評価ツールとしても有効に活用できると考えられる．

最後に，日頃御指導頂く KDD 研究所浦野所長，眞家次長に感謝します．

参考文献

[1] GCC version2.5.2, Free Software Foundation, 1993
[2] Microsoft C Optimizing Compiler ユーザーズガイド, マイクロソフト株式会社, 1989
[3] Borland International: Borland C++ ユーティリティガイド, ボーランド株式会社, 1992
[4] Lattice C Compiler リファレンスマニュアル１, LIFEBOAT
[5] American National Standards Institute, American National Standard for Information Systems, Programming Language C, X3.159-1989
[6] B.W. カーニハン, D.M. リッチー: プログラミング言語 C ANSI C 規格準拠, 共立出版 (1989)
[7] B.W. カーニハン, D.M. リッチー: プログラミング言語 C UNIX 流プログラム書法と作法, 共立出版 (1981)
[8] S.C.Johnson: "Lint, a C Program Checker", Unix Programmer's Supplementary Documents, Vol1.1, 9, Univ. of California, April.1986.
[9] 西風 一, 原田 稔, 前田 忠彦, 大浜 美雪: PSS:C 言語プログラムの移植支援システムー概要, 情報処理学会第４４回全国大会 4F-07, 5-pp.149-150(1992)
[10] 前田 忠彦, 大浜 美雪, 西風 一: PSS:C 言語プログラムの移植支援システムー矛盾指摘チェッカ, 情報処理学会第４４回全国大会 4F-05, 5-pp.145-146(1992)
[11] 西川 一紀, 川村 耕基, 西風 一, 中野 一俊: PSS:C 言語プログラムの移植支援システムー継承支援 C コンパイラ, 情報処理学会第４４回全国大会 4F-06, 5-pp.147-148(1992)
[12] 宮武圭子, 瀧塚孝志: C 言語プログラム検査システムの設計, 情報処理学会第 45 回全国大会 5T-06, 5-pp.257-258 (1992)
[13] 宮武圭子, 瀧塚孝志, 浅見徹: C 言語プログラム表記検査ツールの実装, 電子情報通信学会秋季大会 D-48, pp.50 (1993)
[14] 宮武圭子, 瀧塚孝志, 小花貞夫: C 言語プログラム表記検査ツールの評価と機能拡張, 電子情報通信学会春季大会 D-03, pp.99 (1994)