ソフトウェア工学 97－1
（1994．3．1）

言語処理系CAT／386の開発における ソフトウェア再利用の定量的評価

中村浩之，早川栄一，並木美太郎，高橋延匡

（東京農工大学　工学部　電子情報工学科）

本稿では，言語処理系の開発において行ったソフトウェア再利用と，言語Cの関数インタフェースチェックツールの効果について定量的に評価する．

言語処理系の開発において，早期実現と信頼性確保のために，既存の言語処理系からソースの再利用を行った．M68000用コードジェネレータのプロセッサ独立部の再利用を行い，完成した80x86用コードジェネレータの35.7%を再利用したソースで構成することができた．また，実現において言語Cの関数インタフェースのチェックツールを利用し，29.2%のエラーを実行前に検出することができた．

A Quantitive Evaluation of Software Reuse in the Implementation of CAT／386，a C Compiler

Hiroyuki Nakamura，Eiichi Hayakawa，
Mitarou Namiki and Nobumasa Takahashi

Department of Computer Science，
Faculty of Technology，
Tokyo University of Agriculture and Technology，
2－24－16，Naka－cho，Koganei－shi，Tokyo，184 Japan

In this paper we describe a quantitive evaluation into the effectiveness of software reuse and using interface validation tools on C functions.

We reused the source program from an existing compiler to speed implementation and improve reliability. In this Implementation, we reused the source program of a code generater for M68000 to implement a code generater for 80x86, and we achieved this with 35.7% reuse of the original program's source. We also used tools for validating the interface functions of C, these tools finding 29.2% of errors in the code generater for 80x86 before execution.

1．はじめに

言語処理系はプログラム開発の基礎となる重要なソフトウェアである．近年の RISC プロセッサや並列計算機などの新しいアーキテクチャの出現と共に，その重要性は益々増加している．

言語処理系の開発手法としては，コンパイラ・コンパイラのような言語処理系の生成系の研究がなされている．しかし，コードの実行性能の問題から，実際にプログラム開発で利用する言語処理系が自動生成されることは稀である．

ソフトウェアの早期実現と信頼性確保を行う方法の一つとして，既存のソフトウェアから再利用を行う方法がある．我々は，新しい言語C処理系の実現において，既存の言語C処理系からプロセッサ独立部分の再利用を行った．

本稿では OS/omicron 第４版[1]用の言語C処理系 CAT/x86「知紙（かずし）」の実現において行ったソフトウェアの再利用のについて定量的に評価する．また，実現において利用した関数インタフェースのチェックツールの効果について定量的に評価する．

2．言語C処理系「知紙」の設計

ここでは最初に，被再利用対象となった言語C処理系 CAT の構成について述べ，次に言語C処理系「知紙」の構成と，CAT から再利用を行う部分について述べる．

2．1　言語C処理系 CAT

言語C処理系 CAT は我々の研究室で開発し，利用している言語C処理系である．CAT は OS/omicron 用のシステム記述言語であり，OS/omicron 第１版から第３版までが CAT を用いて開発された．現在，我々は CAT 第３版を用いて，OS/omicron 上でのソフトウェア開発を行っている．

言語C処理系 CAT の構成を図 1 に示す．CAT のフェーズ分割は，各フェーズを他の言語処理系と共有できるように行われている．言語依存部と言語独立部を明確に分離し，言語依存部をパーザに集約することで，コードジェネレータ以降のフェーズを言語独立としている．これは，

（１）他言語の処理系の開発を容易に行える

（２）クロスコンパイラの開発を容易に行える

ことを目的に，再利用を目指した設計思想に基づいている[2]．

本研究では，言語／プロセッサ独立に設計された CAT のソフトウェアアーキテクチャを，コード生成部に着目し，プロセッサ独立／依存性を評価する．

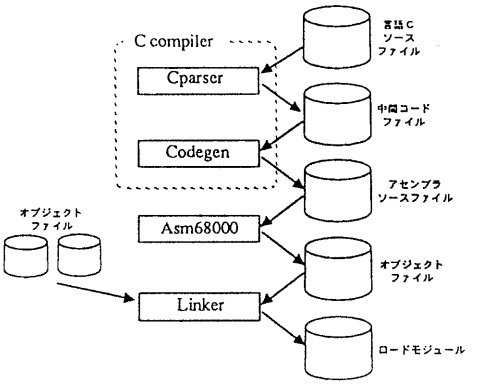

図 1：CAT の構成図

2．2　言語C処理系「知紙」の構成

知紙は OS/omicron 第４版のシステム記述言語に利用することから，早期実現と，高い信頼性が求められる．しかし，新規に言語処理系を最初から作ることは，実現と信頼性確保に多くの時間を必要とする．

一方，前述したように CAT はそれ自身を利用して他のプロセッサ用の言語C処理系をクロスコンパイラの形で実現しやすい構成となっている．CAT を利用してクロスコンパイラを実現した場合，パーザを共有することが可能なので，パーザの実現時間を短縮でき，なおかつ信頼性の高いパーザを利用できる．さらに，新規に実現する部分にデバッグの対象を絞ることができる．そこで，知紙は CAT のプロセッサ依存部を新たに実現し，プロセッサ独立部を再利用することで実現する．

CAT から再利用を行う部分としては，パーザ全体とコードジェネレータの解析木生成部までであ

る．コードジェネレータはプログラムの一部となることから，ソースレベルでの再利用となる．知紙の構成と，再利用部分を図 2 に示す．

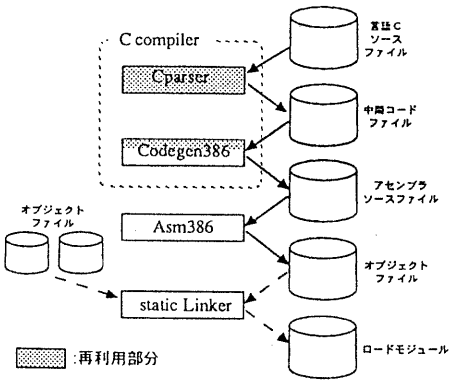

図 2：知紙の構成図

3．言語C処理系「知紙」の実現における CAT の再利用

CAT は M68000 用をベースに，現在 80386/486 用と PA-RISC 用がある．本報告では，80386/486 用 CAT である「知紙」のコード生成部 codegen386 について述べる．

3．1　再利用の方法

codegen386 の実現では CAT のコードジェネレータ codegen からソースの再利用を一部について行う．codegen386 の全体構成と再利用を行う部分を図 3 に示す．中間コードから解析木作成までの部分で再利用を行うが，そのまま利用することはできない．再利用部とはいえ多少の処理変更，追加，削除が必要となる．そのため，その作業においてエラーを混入する可能性がある．また，新規に作成する部分については当然エラーが発生する．そこで，実現については次のような方針を立てた．

（1）二段階に分けて実現を行う

コードジェネレータにおける再利用では，ソースに多少の処理変更や追加，削除が必要となるので，再利用作業でエラーを混入してしまう可能性がある．その場合，信頼性の低下につながるので

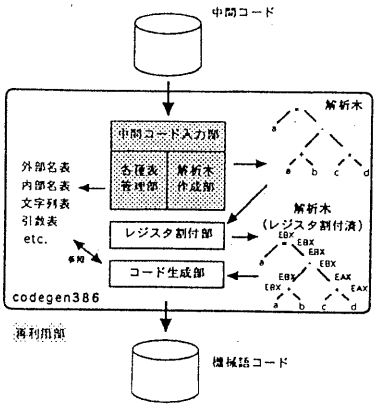

図 3：codegen386 の構成図と再利用部分

好ましくない．そこで，一度再利用を行うモジュールに追加，変更を加えた段階で実行可能なプログラムを実現し，エラーを可能な限り取り除いた後に新規作成となるレジスタ割付け部とコード生成部の実現を行う二段階の実現を行った．このように実現することで，実現した処理系の信頼性を確保することを考えた．また，ソースの再利用の効果をより正確に測定できると考えた．

（2）ツールによる関数インタフェースのチェックを行う

CAT は言語Cで記述され，知紙も CAT を用いて開発を行ったことから，言語Cで記述されている．言語Cでは，Ada のパッケージ等，関数仕様の不整合性をチェックする機構が存在しないことから，この部分でのエラーが発生する．

知紙の実現環境である OS/omicron 第2版上には，acp（Argument Check Pass）ツールという関数呼出しのインタフェースをチェックするツールが存在する．このツールは，パーザの出力した中間コードを用いて関数サイズ，引数サイズ，引数個数等の相違を検出する．関数参照部と関数定義においてこれらが異なっていることはバスエラーの発生原因となり，プログラムを実行するまで現われないので，発見することが難しい．そこで，このツールを利用してリンク・実行前にチェックを行い，実行前に削除可能なエラーをすべて削除することとした．

3．2　再利用

codegen の再利用にあたり，まず再利用を行う部分の識別子の日本語化を行った．CAT 第1版を開発した処理系の制限から，をcodegen のソース中の識別子はほとんどがアルファベット 8 文字で記述されている．このため，ソースの可読性が非常に悪い．そこで，ソースの可読性に有効な日本語識別子 [3] に変換することによって可読性の向上を計り，処理系の保守を容易にしようと考えた．

変換を行った量は 36 ファイルで，ファイル行数は変換前で 7949 行であった．変換に要した期間は 2 か月であった．

日本語識別子化の後にソースの変更，追加，削除を行い，いくつかのモジュールを追加し実現した（以後，この段階を第一段階と呼ぶ）．実現は一人で行い，デバッグも含めて 2 か月を要した．この段階で，ファイル数は 53，総行数 9127 行となった．

再利用部の実現の後に，レジスタ割付け部とコード生成部の実現を行った（以後，この段階を第二段階と呼ぶ）．実現は一人で行い，設計とコーディングに 4 か月，デバッグに 4 か月を要した．デバッグを終了とした時点でのファイル数は 97，総行数は 22179 行となった．

4．実現手法の評価方法

codegen386 の実現では，信頼性確保と早期実現の観点から codegen からのソースの再利用を行った．また，実現において acp ツールを用いた関数インタフェースチェックを行い，関数インタフェースにおけるエラーを実行前に取り除いた．実現で行ったこれらの手段が，はたして信頼性の確保や早期実現役立ったのかを定量的に評価する必要がある．

また，acp ツールのような関数インタフェースをチェックするツールについては，その効果は経験的には知られているが，定量的な評価がなされていない．そこで，codegen386 の実現で利用した結果から，その効果について定量的に評価する．

4．1　評価方法

ソースの再利用の効果については，第一段階と第二段階とで，それぞれに発生した各種エラーの数を比較し，再利用を行った場合の信頼性を評価する．また，その差分のエラー数を除去するのに必要な時間を算出し，再利用によって短縮できた時間を評価する．

acp ツールの検出能力は，実行時テストによって検出したエラーとツール検出エラーの合計量に占めるツール検出エラーの量から評価する．acp ツールが検出するエラーは，ツールを利用しなければ，実行するまで判明しないエラーである．そのため，その数字がそのまま acp ツールの検出能力の指数となる．また，ツールの検出したエラーと同数のエラーを実行時テストでデバッグする場合の時間を推定し，その時間効果を評価する．

これらの評価を行うために，実現中に次のデータを収集した．

（1）コンパイルエラー数

ソースをそのまま再利用するならばコンパイルエラーは発生しないが，codegen386 では codegen のソースに処理の変更や追加・削除，識別子の日本語化等を行ったことから，コンパイルエラーが発生した．そこで，第一段階と第二段階とでコンパイルエラーの数を測定し，それらをソース再利用の評価材料として利用する．また，日本語識別子化の影響についても調査する．

（2）関数インタフェースのエラー数

acp ツールによって検出したエラーの結果を各段階で集計した．この数と，次の実行時テストで発見したエラー数から acp ツールの能力を評価する．また，ソース再利用の評価材料としても利用する．

（3）実行時のエラー数

実行時にテストデータを使って各モジュールのチェックを行い，それによって検出したエラーに

ついてその原因を記録し，原因別に集計を行った．これを用いてソースの再利用の評価や acp ツールの評価を行う．

（4）デバッグ時間

ソースの再利用や acp ツールの使用によってどの程度の時間を短縮できたのかを知るには，評価基準となるデータが必要である．そこで，第二段階の実行時テストにおいて，デバッグ作業にかかった時間を集計した．

4．2　測定結果

測定した各データの集計方法と集計結果を次に示す．

（1）コンパイルエラー数

コンパイルにおいてパーザの出力するエラーメッセージの発生原因を記録し，その原因別集計を行った．記録作業はコンパイルと同時に行い，その記録を元に集計を行った．集計結果を表 1 と図 5 に示す．

表 1：コンパイルエラー数

エラーの種類	第一段階	第二段階
セミコロン忘れ	11	8
ヘッダファイル忘れ／ファイル名間違い	5	16
タイプミス	5	17
マクロ名違い	6	22
識別子の宣言／定義間違い	8	74
括弧類付忘れ	4	15
識別子変換間違い	33	0
引数並びの間違い	0	9
その他	33	19
合計	105	180

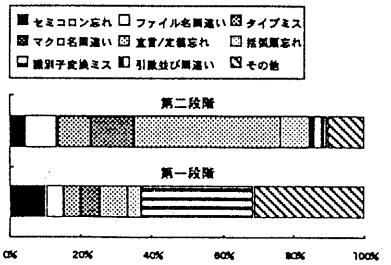

図 5：コンパイルエラー数

（2）関数インタフェースのエラー数

acp ツールが検出したエラーについて，その出力結果を元に，検出原因別に集計した．集計は第一段階と第二段階の両方を集計した．結果を表 2 と図 6 に示す．

表 2：関数インタフェースエラー数

種類	第一段階	第二段階
引数のサイズ間違い	5	39
引数の個数の間違い	2	17
関数の宣言型の間違い	9	31
未定義シンボル	8	7
関数名間違い	3	84
合計	27	178

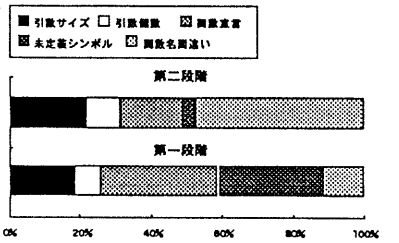

図 6：関数インタフェースエラー数

（3）実行時のエラー数

実行時テストにおいて発生したエラーを，それぞれの段階で原因別に集計した．原因別の集計結果を表 3 に示す．第二段階で発生したエラーについては，その発生数が多いことから，原因別の割合も図 7 に示す．また，第二段階についてはエラーの発生日別の集計も行った．集計結果を表 4 に示す．

表 3：実行時エラー数

原因	第一段階	第二段階
文中の条件式の条件間違い	0	7
演算子の間違い	0	3
必要な処理の記述をしていなかった	4	33
余計な処理が入っていた	0	4
処理順序の間違い	0	5
識別子名間違い	0	4
変数定義型の間違い	0	2
引数並び順の間違い	0	71
引数自体の間違い	0	25
定数値の間違い	1	8
最適化処理が入っていなかった	0	13
プロセッサ仕様の誤解	0	9
その他	3	27
合計	8	211

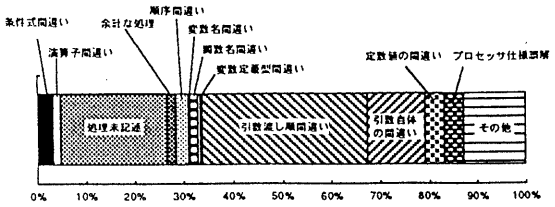

図 7：第二段階の実行時エラー数

（4）デバッグ時間

デバッグ時間の収集は，OS/omicron 第2版ファイルシステム上に実現されている追記型光ディスクを用いた世代管理機能[4]を利用して行った．この機能は，本来書換え不可能な追記型光ディスクを仮想的に書換え可能に見せる仮想ディスク方式で実現されている．仮想ディスク全体を単位とした世代管理を行っており，世代情報として世代番号，世代終了時の日時，任意の文字列（128 文字）を得ることができる．これらの世代情報から次のような方法でデータの収集を行った．

1)テストの開始時に作業開始を示す文字列を記録する

2)終了時に作業終了を示す文字列を記録する

3)デバッグが終了した時点でこれらの情報からテスト時間を算出する

このようにして収集した作業時間と累計作業時間を表 4 に示す．また，作業時間 1000 時間ごとの検出エラー数の累計を表 5 と図 8 に示す．

このデータと実行時テストで検出したエラー数からエラー一つあたりの平均デバッグ時間を単純に計算すると，約 58 分となった．

表 4：デバッグ時間と検出エラー数

日付	個数	作業時間	累積個数	累積時間	日付	個数	作業時間	累積個数	累積時間
7/ 6	4	420	4	420	9/ 6	2	155	151	6177
7/ 7	1	400	5	820	9/ 7	0	266	151	6443
7/ 8	69	296	74	1116	9/ 8	1	182	152	6625
7/ 9	2	247	76	1363	9/ 9	7	277	159	6902
7/12	0	58	76	1421	9/14	0	358	159	7260
7/13	16	539	92	1960	9/16	1	13	160	7273
7/14	2	273	94	2233	9/20	4	384	164	7657
7/15	0	34	94	2267	9/21	2	296	166	7953
7/18	0	133	94	2400	9/24	9	305	175	8258
7/20	8	337	102	2737	9/25	1	106	176	8364
7/21	2	93	104	2830	9/26	1	146	177	8510
7/23	4	169	108	2999	9/27	0	401	177	8911
7/27	2	161	110	3160	9/29	0	82	177	8993
8/12	17	257	127	3417	10/ 2	2	153	179	9146
8/13	1	207	128	3624	10/ 3	4	280	183	9426
8/23	5	336	133	3960	10/ 4	3	242	186	9668
8/24	0	326	133	4286	10/ 5	5	270	191	9938
8/25	3	373	136	4659	10/ 6	0	264	191	10202
8/27	2	254	138	4913	10/ 8	7	265	198	10467
8/29	2	75	140	4988	10/12	10	443	208	10910
8/30	2	280	142	5268	10/13	0	322	208	11232
8/31	1	130	143	5398	10/14	0	347	208	11579
9/ 1	3	266	146	5664	10/15	0	252	208	11831
9/ 2	0	88	146	5752	10/16	2	232	210	12063
9/ 3	3	270	149	6022	10/18	1	13	211	12076

時間の単位：分

表 5：1000 時間単位の検出エラー数

時間（分）	個数	累積
0 ～ 1000	5	5
1000 ～ 2000	87	92
2000 ～ 3000	16	108
3000 ～ 4000	25	133
4000 ～ 5000	7	144
5000 ～ 6000	6	146
6000 ～ 7000	13	159
7000 ～ 8000	7	166
8000 ～ 9000	11	177
9000 ～ 10000	14	191
10000 ～ 11000	17	208
11000 ～ 12000	0	208
12000 ～ 12076	3	211

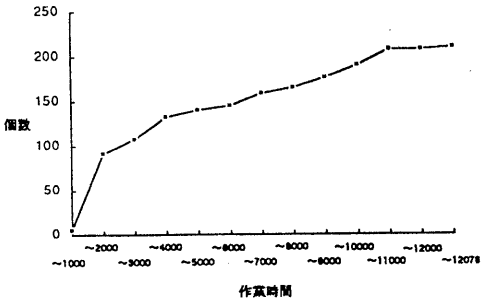

図 8：1000 時間ごとのエラー数の推移

これらの結果から，実現手法等の評価を行う．

4．3　ソース再利用の効果

第一段階と第二段階について，コーディングエラー数，acp ツール検出エラー数，実行時エラー数の比較を行った．第一段階と第二段階ではソースの総行数が大きく異なるので，ソース行数をエラー数で割った比で比較を行った．結果を表 6 に示す．

表 6：エラー数の比較

種類	第一段階	第二段階
コーディングエラー	75.3(9127/105)	123.2(22179/180)
関数インタフェースエラー	338.0(9127/ 27)	124.6(22179/178)
実行時エラー	1140.9(9127/ 8)	105.1(22179/211)

まず，コーディングエラーについては第一段階のほうが多く発生していることがわかる．これは，再利用時に識別子の日本語化を行ったことが影響している．変換作業の間違いから発生したコンパイルエラーは全体の 31.4% で，原因別では一番多い．また，この数を抜いて同じ比を計算すると

109.8 という値になるが，それでも第二段階での測定結果のほうが良いデータが出ている．この結果からは，再利用するソースプログラムでも人の手が加われば，コーディングエラー数は新しく記述した場合と大差なくなってしまうことがわかる．

次に，acp ツールの検出結果を元に比較する．今回の結果では，第一段階で検出したエラー数は，第二段階の検出量の 1/3 近くになっている．この結果では，再利用した部分は新規に作成した部分よりもエラー数が少なく，信頼性が高いと言える．

実行時テストで発見したエラー数では，その差がより大きくなる．同様の方法で第一段階と第二段階の両方を比較すると，第一段階でのエラー発生率は第二段階の 1/10 以下である．このように，acp ツールでの検出結果と実行時エラー数を比較した結果からは，エラーの発生数を押さえるという点ではソースの再利用は有効だったと言える．

実現時間の短縮という点でも，ソースの再利用は有効であった．第二段階の実行時エラーの数から予測すると，再利用した規模のソースを新たに記述した場合に発生するエラーの数は，約 87 個となる．実際に第一段階で実行時に発生したエラーの数は 8 個なので，その差は 79 個である．この数のエラーをデバッグするには 4582 分，約 76 時間の時間が必要である．これだけの時間を短縮できたと考えることができる．

以上の結果から，ソースの再利用という実現方法は，信頼性の確保と早期実現という二つの要求に十分に応えたと言える．

4．4　acp ツールの性能

acp ツールが検出するエラーは，関数インタフェースで発生する次の３つのエラーである．

・関数の宣言サイズと定義サイズの違い
・引数の宣言サイズと定義サイズの違い
・引数個数の違い

これら以外に，未定義シンボルなども検出可能であり，今回の測定では実際に検出している．しかし，未定義シンボルはリンカでも検出できるので，acp ツールを使って初めて検出できるのは上記の３種類と考えてよい．

関数インタフェースのエラーは，関数のプロトタイプである程度防ぐことが可能なエラーである．しかし，参照位置と定義位置の比較ではないことから，完全に取り除くことは不可能である．また，言語仕様ではプロトタイプの記述は省略可能であることから，関数プロトタイプによるチェック機能だけでは完全とは言えない．

（１）検出能力の評価

acp ツールはプログラムの実行前に 87 個のエラーを検出している．これらが実行時のエラーに混入されていたと仮定すると，実行時テストで検出するエラーの 29.2% を検出できたと考えることができる．

また，これだけの量のエラーを自分で取るとなれば，その作業時間も非常に多くなる．実行時テストにおいて一つのエラーを取るのにかかった平均時間 58 分から計算すると，acp ツールが検出した数のエラーを取り去るのに必要な時間は 50 46 分，84 時間 6 分である．一日に 8 時間デバッグ作業を行うとすると 11 日分の時間を節約できたことになる．しかし，実際に実行デバッグをするとなると，それはプログラムの実行もままならない状態で行わなければならいので，もっと多くの時間を節約できたと考える．

（２）ツールの問題点

今回の結果からも，acp ツールの効果は大きく，特に初期のデバッグでは非常に有効であることが言える．しかし一方で，関数インタフェースのエラーでありながら検出できずに実行時テストまで残ったものがある．例えば，新規作成部分の実行時デバッグでは acp ツールを通したにも関わらず，引数並びの間違いが 71 件も発生している．これは，定義側と参照側で，同サイズで型が違う引数（構造体へのポインタ等）を積む順序が逆になっていたものがほとんどである．acp ツールは

サイズの相違しかチェックしないので，このような型の違いだけのエラーは発見することができない．より厳密なチェックを行うには，引数型の違いまで識別するツールが必要である．

5．考察

5．1　言語Cプログラムにおけるエラーの傾向

第二段階で発生したエラーで，関数インタフェースのエラー数と実行時エラー数を見ると，関数インタフェースのエラーが非常に多いことがわかる．その二つのエラーを併せた数では，全体の47.0% を関数呼出しでの間違いが占めている．この結果から，関数インタフェースのエラーは非常に発生しやすいエラーであったことがわかる．

関数インタフェースのエラーはバスエラーや誤動作の元になるので，言語Cによるプログラミングでは注意すべき点と言われている．このエラーの防止方法としてはツールによる事前チェックが有効である．今回利用した acp ツールや UNIX の lint のような型チェックツールはぜひとも利用すべきである．

5．2　codegen の再利用性

codegen の再利用性を評価するために，codegen386 の全ソースに占める再利用ソースの割合（再利用率）と codegen の全ソースに占める被再利用ソースの割合（被再利用率）を測定した．測定結果を表 7 に示す．

表 7：ソースの再利用率

種類	総行数	（被）再利用部行数	（被）再利用率
codegen	36454	6601	18.1%
codegen386	22179	7908	35.7%

被再利用率からは codegen の再利用性はそれほど高くないように見える．しかし，これは CAT の機能の，複数の浮動小数点数表現方式[5]を再利用対象としなかったためである．

一方，再利用率を見ると，総プログラム数の35.7% を再利用したソースによって構成することができた．新規の処理系開発においてこれだけの量のソースを再利用できた．また，PA-RISC用の CAT における再利用率でも，ほぼ同じ再利用率が得られた．

6．終わりに

本稿では，言語C処理系 CAT/x86「知紙」の実現において行ったソフトウェアの再利用について，その定量的な評価を行った．また，関数呼出しのインタフェースチェックツールの有効性についても定量的に評価した．

再利用を行ったことによって開発期間を短縮し，実現時に発生するエラーの数を低減することができた．また，ツールの検出能力が優秀であり，利用した場合の効果もエラーの発生原因の傾向から非常に高いことがわかった．

今後は，知紙を実際に利用し，その過程で発生したエラー数を CAT 利用時のデータと比較することで，処理系利用時の信頼性の評価を行う．

参考文献

[1]早川，他：“手書きインタフェースを支援する OS OS/omicron 第４版の構成”，情報処理学会コンピュータシステムシンポジウム論文集 Vol92, No.7, pp35-42 (1992).

[2]並木，他：“OS/omicron 用システム記述言語C処理系 Cat のソフトウェア工学的見地からの方式設計”，電子情報通信学会論文誌 D, Vol.J71-D, No.4, pp.652-660 (1988).

[3]中川，他：“日本語プログラムの可読性の評価と検討”，情報処理学会ヒューマンインタフェース研究会 (1992)

[4]横関，他：“追記型光ディスクの仮想的な書換えと世代管理機能の実現”，電子情報通信学会論文誌 D-I, Vol.J72-D-I, No6, pp.414-422 (1989)

[5]中原，他：“複数の浮動小数点表現法を処理するシステム環境の設計と実現”，情報処理学会論文誌, Vol.33, No.4, pp.481-490 (1992)