書換えパターン記述によるコーディング規約適用支援に関する研究

2006MI033 堀田 淳司　　2006MI041 石井 健司

指導教員　野呂 昌満　蜂巣 吉成

1　はじめに

ソフトウェアの開発効率を向上させる方法に，他のプロジェクトやオープンソースのソースプログラムの再利用がある．しかし，他のソースプログラムの一部を開発中のソースプログラムに取り込むと，可読性や拡張性の低下といったプログラム品質の低下が起きやすい．その原因の一つが，コーディング規約 [1, 2] の不一致である．コーディング規約は開発現場ごとに異なり，コーディング規約に合わない記述が混入することで可読性が低下する．取り込んだソースプログラムを開発現場の規約に適合するように修正する作業は，単純作業の繰り返しであり，人為的な誤りが混入する恐れもある．

本研究はコーディング規約の適用支援を目的とし，コーディング規約適用支援ツールの提案と実装を行なう．コーディング規約の適用作業をツールによって自動化することにより作業効率の向上や誤り混入を防止できる．ただし，開発現場ごとにコーディング規約が存在するので，各利用現場で必要に応じて規約の定義を簡単にカスタマイズできることが必要である．本研究では，書換えパターンを用いて規約を定義する仕組みを導入する．

2　コーディング規約

コーディング規約とは，実装工程で品質管理を行なう際の品質基準である [2]．複数の開発者による協働作業としてコーディングを行なう場合，各々の流儀でソースプログラムを書くと他人の書いたソースプログラムの理解や修正が難しくなる．ソースプログラムの記述形式を規約として統一しておくことにより，ソースプログラムは一定以上の可読性，保守性が得られる．コーディング規約は，コーディングスタイル，命名規則，禁止規則に分類できる．コーディングスタイルとは，構文の記述スタイルを統一する規約や，コメントの表記方法に関する規約である．命名規則とは，識別子やマクロの名前のつけ方に関する規約である．禁止規則とは，グローバル変数を使用しないといった，演算の誤差や人為的な誤りを防ぐための規則に関する規約である．

文献 [2] のコーディング規約は 66 個あり，8 個の命名規則，20 個の禁止規則，38 個のコーディングスタイルに分類した．

3　コーディング規約適用支援ツール

本研究は，コーディング規約の適用作業を自動化するための適用支援ツールを提案する．コーディング規約は企業ごとに異なっているので，利用場所に応じて規約の定義を書換えパターンとして記述する方法を提案する．

3.1　適用支援ツールの概要

適用支援ツールの概略を図 1 に示す．適用支援ツールは対象となるソースプログラムとコーディング規約を記述した書換えパターンを入力とし，書換え済みソースプログラムを出力する．

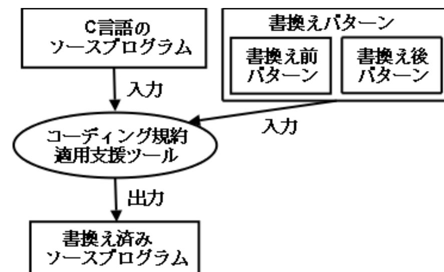

図 1　適用支援ツールの概略

3.1.1　入力

図 1 に示すように，適用支援ツールの入力は対象とするソースプログラムと書換えパターンである．本研究で対象とするソースプログラムは，C 言語で記述されており，再利用したソースプログラムが混入していることを前提としている．

本研究の書換えパターンとは，トークン列に対する書換えの記述であり，コーディング規約に合致しないトークンの並びを表現する書換え前パターンと，規約に合致するよう書換えた後のトークンの並びである書換え後パターンで構成する．各パターンは，トークンをアルファベットとする正規表現と等価である．各パターンは，Perl や Sed などが持つ正規表現による文字列置換をトークン列に応用したものと捉えることができる．詳細は 3.2 節に示す．

本研究で対象となるコーディング規約は 2 節の 3 つの分類のうち，禁止規則とコーディングスタイルに関する規約とする．命名規則はプログラムの意味に依存しており，書換えパターンとして記述できないので，本研究では対象としない．禁止規則は，トークンの位置や出現に関する制約と解釈でき，トークンの並びに対するパターンとして表現できる．トークンの並びに対するパターンとして表現できる規約は，書換えパターンとして記述できるので，本研究の対象とする．コーディングスタイルは，構文の記述スタイルを統一する規約であれば禁止規則と同様にトークンの並びに対するパターンとして表現できる．

コーディング規約を適用するにあたり，規約同士が循環しない停止性や，適用順序に関係なく最終的に同じ結

果を得られる合流性を考える必要がある．停止性と合流性については 4.4 節で検証する．

3.1.2 出力

適用支援ツールの出力は，コーディング規約が適用された書換え済みソースプログラムとなる．書換え済みソースプログラムは適用前と同様の動作をすることを前提とする．適用支援ツールを適用したときに，利用者の意図に合う書換えが行なわれることを保証するのは難しいので，書換え済みソースプログラムを元に戻せるようにする．具体的には，前処理命令の`#ifndef`，`#else`，`#endif` を用いて書換え前を無効の状態で残しておき，マクロ定義の変更で書換え前を有効にすることで元に戻す．

3.2 書換えパターンの表現と記述方法

書換えパターンは書換え前パターンと書換え後パターンで構成されており，各パターンは，論理的にはトークンと空白文字列が混在したリストで表現する．ただし，各パターンを単純にトークンと空白文字列のリストとして記述すると，可読性や保守性が低くなる．そこで，ソースプログラムの一部をパターンとして記述し，字句解析によってリストに変換する．また，書換え前パターンの空白文字列は，任意の空白文字列に適合するものとして扱う．

コーディング規約「c1）1 つの宣言文で宣言する変数は 1 つにする」と「c2）条件分岐はブロック化する」の書換えパターンをソースプログラムの一部で記述した例を図 2 に示す．

c1) 1つの宣言文で宣言する変数は1つにする

```
#BEFORE
int count, ind;
#AFTER
int count;
int ind;
```

c2) 条件分岐はブロック化する

```
#BEFORE
if (count<10)
  count++;
#AFTER
if (count<10) {
  count++;
}
```

図 2　ソースプログラムの書換えパターンの記述

書換えパターンの記述は図 2 のように`#BEFORE`，`#AFTER` それぞれの後に，書換え前パターンおよび書換え後パターンを記述する．なお，図 2 の状態でパターンマッチングを行なうと，書換え前パターンで記述したトークン列がソースプログラムの対応する箇所と完全一致しないとマッチングしない．規約 c1）の書換えパターンでは，対象ソースプログラムに int の型宣言で count，ind という変数名の宣言が記述されていることが前提となる．規約 c2）の書換えパターンも同様に完全一致しないとマッチングしない．

図 2 のように，変数名や型を直接指定すると汎用性がないので，抽象化した書換えパターンの記述方法が必要である．そこで，本研究では表 1 に示すメタ記号を定義し，この表現を用いて書換えパターンの記述を行なう．

表 1 の（A）は，トークンの字面ではなく，`@`*token* と同じ種類の任意のトークンにマッチングする．トークンの種類とは，トークンそのものではなく，字句解析と構文解析によって求められる終端記号の種類である．例えば，変数，型，数値などである．（B）は，書換えパターンで記述された箇所に，*token* を含まないトークンにマッチングする．（C）は（B）に加えて，書換えパターンで記述された箇所に，*token* を含むトークンにもマッチングする．

表 1　メタ記号の定義

	メタ記号	意味
（A）	`@`*token*	同じ種類の任意のトークン
（B）	`@\`*token*`\@`	トークンが出現しない
（C）	`@?`*token*`?@`	トークンの出現が 0 か 1 回
（D）	`$Expr`	式
（E）	`$eStmt`	式文
（F）	`$cStmt`	式文か制御ブロック
（G）	`$Stmts`	文の連続

トークンのみで汎用的な書換えパターンを記述することは難しいので，構文単位を表わすメタ記号が必要になる．本研究では，式，式文，式文と制御ブロック，文の連続の構文単位を表わすメタ記号を導入した．（D）は，演算子，オペランド，区切り子の任意の並びで構成されるトークン列である．（E）は，式にセミコロンが付加されたトークン列である．（F）は，式文か，制御ブロックを表したトークン列である．（G）は，制御ブロックと式文の集まりで表わしたトークン列である．

表 1 のメタ記号を用いた書換えパターンの例を図 3 に示す．図 3 は，図 2 に対応した書換えパターンの記述である．

c1) 1つの宣言文で宣言する変数は1つにする

```
#BEFORE
@type @var1,@var2;
#AFTER
@type @var1;
@type @var2;
```

c2) 条件分岐はブロック化する

```
#BEFORE
if ($Expr_1)
$cStmt
#AFTER
if ($Expr_1) {
$cStmt
}
```

図 3　メタ記号を用いた書換えパターンの記述

図 3 の規約 c1）の書換え前パターンに記述されている`@`type は，解析するとトークンの種類が変数宣言の型と推測され，対象ソースプログラムの int や char といった型にマッチングする．`@`var1，`@`var2 は変数名と解析されるので，変数名を表す任意のトークンとマッチングする．書換え後パターンの`@`type は，書換え前パターンの`@`type のトークンの種類と対応し，書換え前で int が適合した場合，書換え後は`@`type が int に置き換わる．また，図 3 の規約 c2）の書換え前パターンで記述されている`$Expr_1` は，if 文の条件式にマッチングする．`$Expr_1` は`$Expr` の特殊表現で，同じメタ記号を複

数使うとき，それらを区別するためにメタ記号の後ろに“ _*id* ”を追加できる．`$cStmt` は複合文以外の文なので，中括弧がない場合にマッチングする．

3.3 適用支援ツールの設計

図 4　コーディング規約適用支援ツールの流れ

コーディング規約適用支援ツールは図 4 に示すように 5 種類の処理を組合わせて構成している．これらの処理は Perl で実装している．なお，以下の説明では，空白文字の連続も 1 つのトークンとして扱う．

1) **字句解析**

字句解析では，対象ソースプログラムと書換えパターンをトークンごとに分解し，対応するトークンの種類を属性値として割当てる．

2) **構文解析**

構文解析では，対象ソースプログラムと書換えパターンのトークン列に構文情報を与える．トークンの並びのみでは書換えパターンを表現できないことがあるからである．一般的に構文解析は式単位まで解析を行なっているが，本研究で対象とする規約では文の区別が付けば十分であるので，解析の粒度を文までとした．構文情報は，各文の前後に開始と終了を表す仮想のトークンを加える．なお，以下では，仮想のトークンを両端字句と呼ぶ．

3) **パターン変換**

パターン変換では，トークン列で表現される書換えパターンを文字列に対する正規表現に変換する．また，空白文字は 0 個以上の空白の連続を表わす正規表現に変換する．

4) **書換えパターン適用**

書換えパターン適用は，トークン列になった対象ソースプログラムに対して，文字列の正規表現に変換された書換えパターンを適用し，文字列置換を行なう．

5) **字句結合**

字句結合は，文字列置換を行なった書換え済みソースプログラムのトークン列を，空白や改行を整えてテキストに変換する．

3.4 コーディング規約の適用例

適用支援ツールが書換えパターンとして記述した規約をソースプログラムに適用できるか検証した．検証方法は，個々の書換えパターンについて，対象のソースプログラムに対して適用できるか簡単なテストを行なった．例を図 5 に示す．図 3 の 2 つの書換えパターンで実際にコーディング規約を適用した．対象ソースプログラム `sample_before.c` に規約 c1）と c2）を適用したところ `sample_after.c` が得られた．

sample_before.c

```
#include<stdio.h>
main(){
  int x,y;
  x=y=1;
  if(x==y)
    x=2;
  return 0;
}
```

sample_after.c

```
#define RULE_1_1
#define RULE_2_1
#include<stdio.h>
main(){
#ifndef RULE_1_1
  int x,y;
#else
  int x;
  int y;
#endif
  x=y=1;
#ifndef RULE_2_1
  if(x==y)
    x=2;
#else
  if(x==y){
    x=2;
  }
#endif
  return 0;
}
```

図 5　コーディング規約の適用例

4 評価と考察

4.1 書換えパターンの適用可能性

文献 [2] の規約のうち，2 節で述べた対象となる規約に対して書換えパターンの適用可能性を検証した．3.1.1 節より対象となる規約は 41 個あり，そのうち 8 個について記述できた．記述できた規約は，本研究で定義したメタ記号であるトークンの種類，トークンの出現回数，構文単位より，書換えパターンの適用がソースプログラムに対して行なえた．適用ができなかった 33 個の規約については，情報や表現を追加することで適用が可能になると考える．詳しくは 4.5 節で考察する．

4.2 適用支援ツールの動作の検証

適用支援ツールの処理が意図した結果になるか検証を行なった．構文解析で意図した処理が行なわれるか，また規約の適用が意図した結果になるか 100 行程度のソースプログラムに対して適用した．結果として構文情報が必要になる規約以外は適用でき，意図した通りに対象ソースプログラムを書換えることができた．できなかった規約については，構文解析器の機能に問題があったので，一部の両端字句の順序を手動で補正した．補正した

解析結果を使用したところ，意図した通りの書換えができることを確認した．

4.1 節と本節から，本研究の目的であるコーディング規約の適用支援は，書換えパターンを用いた適用支援ツールにより，意図した規約の適用ができ達成される．

4.3 停止性・合流性について

書換えパターンの適用ができたコーディング規約の停止性と合流性の検証は，項書換え系の考え方を基に検証を行なった．停止性は規約の適用が循環しないか検証し，合流性はソースプログラムに対してを複数の規約を適合するときに，適用順序に関係なく同じ結果が得られるか検証した [3]．結果，規約を適合させるときの停止性・合流性はともに満たしていた．

4.4 適用できなかった規約について

本研究の適用支援ツールで，コーディング規約の適用ができなかった 33 個の規約に関して考察を行なう．コーディング規約の適用ができなかった原因は，以下の 4 つの組み合わせとして分類した．

(1) 型情報
(2) プログラムの意味の解釈
(3) プログラムの文脈の条件
(4) 構文単位を表わすメタ記号に対して，トークンの置換と削除

原因の（1）は，解析後の情報が不足していることの問題である．(2)，(3)，(4）は，コーディング規約を書換えパターンとして記述する方法の問題である．

書換えパターンの記述にあたり，型の制約が必要となる規約が存在する．(1）の型情報を与えることによって，変数や式の型に関する規約を書換えパターンとして記述できる．これは，「キャストする場合は，演算結果ではなくオペランドをキャストする」など計 7 個の規約に関係している．そのうち，5 個の規約が記述可能となる．

書換え後パターンの記述にあたり，(2）が必要となる規約が存在する．意味を解釈できないと書換え後パターンを記述できない規約があるが，部分的に記述できる規約も存在する．その場合, 書換え可能な部分のみ記述し，残りの部分はユーザに修正を促すような雛型に置換えることで作業量の軽減を行なう方法がある．この方法は，「浮動小数点の比較は許容範囲を考慮する」など計 16 個の規約に関係している．例えば「浮動小数点の比較は許容範囲を考慮する」は許容範囲を機械的に修正できないので，ユーザに許容範囲を指定する記述を追加してもらう必要がある．

（3）は，本研究で定義した構文単位を表わすメタ記号に対して，トークンの出現条件を書換えパターンに記述することで表現できる．これは，「マクロ定義では，使われる仮引数はすべて小括弧で括る」など計 7 個の規約に関係している．そのうち，3 個の規約が記述可能となる．例えば「マクロ定義では，使われる仮引数はすべて小括弧で括る」は，“ 小括弧で括られていない ”という文脈の条件を満たす識別子が書換え対象になり，書換えパターンとして記述する必要がある．

（4）は，書換え処理が必要になる規約が存在する．この処理ができれば，式文から式といった構文要素を変更することができる．これは，「分岐やループの制御部と処理は分離する」など計 5 個の規約に関係している．そのうち，3 個の規約が記述可能となる．例えば「分岐やループの制御部と処理は分離する」は，制御に関する処理のプログラムを制御部に移すために構文要素を変更しなければならないので，この書換え処理が必要である．

これら 4 つの原因を解決できれば，25 個の規約が書換えパターンとして記述でき適用可能になる．残りの 8 個の規約は，特別な表現や処理が必要になる規約である．例えば，複数のファイルを対象とする処理の追加を行なえば，適用可能であると考えている．

他のコーディング規約ガイドライン [1] についても分析を行なった．文献 [1] には対象になる規約が 98 個あり，(2）は，54 個の規約に関係しており，解決することで 34 個の規約が書換えパターンとして記述できる．(3) は，38 個の規約に関係しており，解決することで 22 個の規約が適用できる．(4）は，7 個の規約に関係しており，解決することで 5 個の規約が適用できる．このように原因が混在していることを確認し，原因を解決することにより 98 個の規約のうち 76 個の規約が適用可能になると判断した．他のコーディング規約ガイドラインの書換えパターンの記述と適用が可能になることから，開発現場が異なっていても対応できる．

5 おわりに

本研究では，コーディング規約適用支援のために，汎用性を考えた書換えパターンとして記述する方法を提案し，コーディング規約適用支援ツールの設計と実装を行なった．実装したコーディング規約適用支援ツールの評価を行い．対象とした 41 個のコーディング規約のうち，8 個の規約を書換えパターンとして記述し適用できた．残りの規約の 33 個の規約について，書換えパターンとして記述に必要な要件を考察し，25 個の規約について解決策を示した．今後の課題は，考察した解決策を適用支援ツールに実装することである．

参考文献

[1] David D Ward, and Peter H Jesty, *MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Research Association, 2004.

[2] 福岡知的クラスタ (第 1 期) 組込みソフト開発プロジェクト，組込み現場の「C」プログラミング 標準コーディングガイドライン，技術評論社，2007.

[3] 外山芳人，“ 項書き換えシステム入門，” 信学技報，SS98-15，pp.31-38，1998.