# カスタマイズ可能なコーディングチェッカー

高橋 透 † 大久保 弘崇 †† 粕谷 英人 †† 山本 晋一郎 ††

† 愛知県立大学大学院 情報科学研究科 †† 愛知県立大学 情報科学部 情報システム学科

# A Customizable Coding Rules Checker

Tooru TAKAHASHI† Hirotaka OHKUBO†† Hideto KASUYA††
Shinichiro YAMAMOTO††

†Graduate School of Information Science and Technology, Aichi Prefectural University
††Faculty of Information Science and Technology, Aichi Prefectural University

## 1 はじめに

近年，ソフトウェアの大規模化や複雑化が進む一方で，開発期間の短縮化の要求がある．このような中でソフトウェアの品質を高めるためには，効率よく検証を行うことが望まれている．

一般に，ソフトウェアの検証には動的検証と静的検証がある．動的検証とはプログラムを実際に動作させて検査することである．一方，静的検証とは動的検証ではできない網羅的な検査をソースコード解析を用いて行なうことである．このとき静的検証ではツールを用いてコーディング規約の検証を行う．

しかし，検査ツールは広く知られた規約については対応しているが，開発プロジェクト毎の規約についてはそうではない．現状では独自ルールについては開発者がツールを自作して検査している．そのため，効率よくかつ正確に検証を行う技術が求められている．

そこで，本研究ではソースコードに関するコーディング規約の検証ツール (以下，コーディングチェッカーを呼ぶ) を作成する基盤ソフトウェアを開発する．本システムの特徴は，コーディングチェッカーの作成に軽量プログラミング言語を用いる点にある．軽量プログラミング言語とは，Perl[4], Ruby[8], Python[5] に代表されるスクリプト言語の総称である．これらは，rapid prototyping に適した言語であり，様々なコーディング規約に柔軟に対応するために必要である．

また，コーディングチェッカーを作成するためには高度なソースコード解析が必要となる．我々は C 言語 を対象とした CASE ツール・プラットフォーム Sapid[6, 12] を長年にわたり開発しており，細粒度のソフトウェアリポジトリが提供できる．また，Sapid はリポジトリから細粒度情報を取得するための API を用意している．

本稿では，Sapid を用いたコーディングチェッカーの開発環境を提案する．この環境の作成には，ラッパー生成器である，SWIG[7] を用いる．それにより，C 言語で書かれた API が軽量プログラミング言語から利用でき，リポジトリへアクセスが可能となる．また，C 言語で書かれた API と軽量プログラミング言語とでは命名規則が異なるため，効率良く記述するために命名変換を行う．

本稿の構成は，2 章で，カスタマイズ可能コーディングチェッカーについての詳細を述べ，そのために必要な要素を明らかにする．3 章では，コーディングチェッカーを始めとした静的解析ツールの開発に必要な解析情報を提供してくれる CASE ツール・プラットフォーム Sapid について述べる．4 章では，Sapid を用いて MISRA–C[3] のルールを基盤にしたカスタマイズ可能なコーディングチェッカーを作成し，評価する．最後にまとめる．

## 2 コーディングチェッカー

本章では，カスタマイズ可能なコーディングチェッカーの持つべき機能について検討する．そのために，まず広く知られているコーディング規約について述べる．そしてカスタマイズ可能なコーディングチェッカーに必要な機能を明らかにする．コーディング規約はコーディング時に参照するルールの集合によって構成されている．

### 2.1 コーディング規約

#### 2.1.1 MISRA–C[3]

MISRA–C は，自動車用組込みソフトウェアに関して C 言語を安全に使用することを促進するために作られたガイドラインである．分類については，各々のルールに必須・推奨の重要度が設けられている．また，構造化プログラミングの観点から一部制御構造の利用を制限するルールも存在する．コーディングスタイルに関する制約は「59. 制御構造の本文を構成する文を必ず {} で括らなければならない」というルールの他に特に定められていない．これは，スタイル情報については個々のプロジェクトごとに別途定められるべきであるとうい考えに基づいている．

#### 2.1.2 GNU コーディング規約 [1]

GNU パッケージを作成するためのルールを述べたものである．C 言語のプログラミングに関するルールが中心であるが，他

の言語についても適用できる部分が多くある．コーディングスタイルについて細かいルールが定められている．また，Makefile や ドキュメントの書き方も述べられている．以下に例を示す．

1. 条件付きコンパイル
   `#ifdef` による条件付きコンパイルよりも，ツールによって可能なコード経路の全てについて検査ができる`if` 文を用いる．
2. 頑丈なプログラムを書くには
   特定の関数についてエラーチェックの必要性がある．他にも禁止すべきことについて述べられている．
3. ライブラリ関数
   リエントラントになるようにする．ライブラリの命名規則について述べられている．
4. エラーメッセージの形式
   エラーメッセージは “ファイル名:行番号:メッセージ” などの形式にするように述べられている．
5. コマンドラインインタフェースの規約
   コマンドラインオプション には `getopt()` を用いることが推奨されている．また，オプション名は他のツールを参考にすることが推奨されている．
6. ソースコードの整形・構文についての規約
   GNU コーディングスタイルと呼ばれるものである．関数定義においては厳格に定められている．他には文や式についてもスタイルが定められているが，これらを使うより統一したスタイルを用いることが推奨されている．
7. コメント
   ソースコードはプログラムの説明のコメントから始まる．コメントは英語で書く．関数ごとにコメントを書く (引数の用途，返り値)．コメント文の最後にはスペースを 2 文字置く．`#else`,`#endif` の後にはコメントを入れ対応関係が分かるようにする．
8. 名前付け
   変数名は英語で意味を持つような名前にする．名前の中で語を分けるときはアンダースコアを用いる．
9. 互換性
   計算機の型のサイズや浮動小数点の実装に注意し依存したコードを書かないように述べられている．

### 2.1.3 Indian Hill[2]

AT&T のインディアンヒル・コミュニティのための C 言語についてのコーディング規約である．対象分野は特に定められていないがコーディング時に利用できるルールがまとめられている．

1. ファイルの構成
   プログラムは 1. 概要，2. インクルードファイル，3. マクロ，`typedef`，4. グローバル変数，5. 関数 の順で構成すべきであると述べられている．他にもファイル名に関する慣習について述べられている．
2. コメント
   コメントのスタイル以外に，コメントに書くべき内容について触れられている．
3. 宣言
   `char *` 型の変数 `s, t` を宣言する際は，`char* s, t;` でなく，`char *s, *t;` と書く．他には，`typedef`,`enum`,`#define` の使い分けや記述の位置について論じられている．
4. スタイルに関して
   関数宣言，空白，単文，ブロック文についてのスタイルのルールが細かく定められている．
5. 演算子
   演算子とオペランドの間の空白についてのルールについて述べられている．
6. 命名法
   識別子の命名法が定められている．定数についても，`1` と`l` が紛らわしいため利用しないようにと述べられている．
7. 定数
   定数をそのまま書かずに，`#define` や `enum` を用いた意味の分かる使い方が推奨されている．
8. マクロ
   副作用のある式をマクロに引数として渡さないことが推奨されている．他にもマクロに関するスタイルのルールについて述べられている．
9. 条件付きコンパイル，デバッグ
   `#ifdef` の使用目的を改めて確認した上で，`enum`, `assert`, `#ifdef` などを用いたデバッグに適したコードについて述べられている．
10. 移植性，ANSI C
    計算機環境に依存しないコードを書くように努める．そのために，型のビット長や，浮動小数点型へのビット演算による比較，整数 (正負値) に対して「2 の補数」表現であることを仮定しないなど，様々な機種依存なコードの例を理由とともに挙げている．また，可能な限り，ANSI で提唱されている規格内でコードを書き移植性を高くすることが推奨されている．

### 2.1.4 組込みソフトウェア用 コーディング作法ガイド [10]

コーディング規約を作成する際に参考となるルールが記載されている．また，ルールは保守性・信頼性といった品質特性ごとに分けられており，作成するコーディング規約が重視する特性によってルールの数を調整できる．この中で定められているルールの多くは，上 3 つのコーディングルールから選ばれている．また，コーディングスタイルについては特に定められていない．しかし，スタイルは既存のスタイルをなるべく利用すべきであると述べられている．

1. 領域は初期化し，大きさ，生存期間に気をつける．
   初期化時，使用中，開放後についてのルールについて述べられている．
2. データは範囲，大きさ，内部表現に気をつける．
   計算機環境によって精度，実装が異なる型について，型を変換する際の注意が述べられている．
3. 異常値を考慮した書き方に．
   疑わしくないときもエラーのチェックをしてから本処理に入ることが推奨されている．

4. 他人が読むことを意識する．
   以下の制約について述べられている．(1) 演算子についての制約 (2) 宣言についての制約 (3)1 文に副作用は 1 つしか出現してはいけない．
5. 修正しまちがえないように書く．
   制御構造の本体をブロック化することが推奨されている．
6. プログラムはシンプルに書く．
   見通しの良いプログラムを記述するために制限が設けられている．制御構造についての制限．`goto`, `continue` の使用禁止．`break`，`return` 文の制限が挙げられている．
7. 統一した書き方．
   スイタイルに関する制約は定められていない．統一したスタイルを定めることが必要であると述べられている．
8. 試験しやすくなる書き方に．
   デバッグのためのコーディング方法を定めることが必要であると述べられている．また，検査ツールを用いることですべての経路について検査ができるため，マクロ関数よりも関数の利用が推奨されている．
9. コンパイラに依存しない．
   拡張機能については利用しないことが推奨されている．処理系依存な部分については確認し文書化した上でルールを規定することが定められている．
10. 移植性に問題のあるコードは局所化する．
    変更箇所が分散しないようにマクロを用いて局所化することが推奨されている．

### 2.2 カスタマイズ可能なコーディングチェッカー

一般に，コーディング規約の策定は他のコーディング規約からルールを参照し，保守性・信頼性などの品質特性に応じてルールを選択する．そして，選択したルールを基に既存のルールの改変および，新規のルールの追加によってプロジェクト毎の独自のコーディング規約を策定する．しかし，このような細かなルールの差異を全て吸収して対応するような万能なコーディングチェッカーを作成することは不可能である． そのため，異なる部分は開発者が検査ツールを自作しなくてはならない．

この問題について，既存のルールを改変する際には，コーディングチェッカーにパラメータを与えられるようにして開発者の労力を軽減する余地がある．一方，新規に作成したルールに対応する際には，解析器を含む全てを作成しなくてはならない．このとき，CASE ツール・プラットフォームなどの解析情報を提供してくれる基盤があれば，新規ルールに対応するために有用である．また，CASE ツール・プラットフォームを基に作成されたオープンなコーディングチェッカーが存在すれば，ソフトウェアを再利用，参照することでツール作成コストの節約が期待できる．

よって本研究で考えるカスタマイズ可能なコーディングチェッカーとは以下の条件を満たすものとする．

1. 様々なルールを記述するために，ソースコードに関する様々な粒度の解析情報を提供する CASE ツール・プラットフォーム．
2. コーディングチェッカー本体の記述には，様々なルールに柔軟に対応でき，ツールのコスト削減を軽減するような言語が望ましい．
3. 広く知られているコーディング規約については，あらかじめ作成しておき，独自のルールのコーディングチェッカーを作成する際には，他のルールのソースコードが参照できること．

項目 2 について，様々な規則に柔軟に対応するためには宣言的な記法のできる言語を開発することが考えられる．しかし実用性を考慮するとスクリプト言語の利用が妥当である．また項目 3 に挙げたソースコードの再利用・参照を可能とするためにもスクリプト言語を用いることは有用である．

## 3 Sapid

Sapid[12] は，細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォームである．従来，CASE ツール開発者が個別に作成していた基本的な機能である解析器やリポジトリとそれへのアクセスルーチンをあらかじめ用意しておくことで，本質的な機能の開発に専念できる．ここで Sapid の提供するリポジトリのことを SDB (Software DataBase) と呼び，そこへアクセスするための API を AR (Access Routine) と呼ぶ．本研究で作成するコーディングチェッカーも SDB と AR を利用することで効率のよい開発が期待できる．

本章では，2.2 節で述べたコーディングチェッカーに必要な解析情報が SDB によって提供されるのか検討する．また，AR を用いたツール開発をさらに効率化するための方法について考察する．

### 3.1 SDB

SDB は Sapid の解析器からソフトウェアモデルに基づいて得られた実体・関連情報を格納するデータベースである．ソフトウェアモデルとは，ソースコードから得られる構文解析木を目的に応じて取捨したものである．Sapid では，細粒度でソフトウェアを扱うモデルとして，I–model, P–model がある．I–model は前処理語の C 言語の解析木をについてのモデルである．P–model は前処理前のマクロに関するモデルである．ここで言う細粒度とはソースコード中の最小単位であるトークンを意味する．そして，これらを基に作られる抽象度の高いモデルとして Flow–view, Comment–view がある (図 1).

Flow–view はデータフロー，制御フロー解析によって得られるフロー情報に関するモデルである．Comment–view は関数や変数などの識別子とコメントと対応関係を表したモデルである．プログラミング言語に依存しないモデルとして設計されている．C 言語のソースコードを Comment–view で表現するときは，識別子とその直前のコメントを対応づけている．

### 3.2 AR

スクリプト言語によりコーディングチェッカーを記述するために，前節で述べた SDB にアクセスする手段が必要である．Sapid が標準で提供しているアクセス手段は C 言語による API である AR と SDB の内容を XML 化する方法 [13] である．リポジトリを，XML 化することで，DOM, SAX, XPath などの

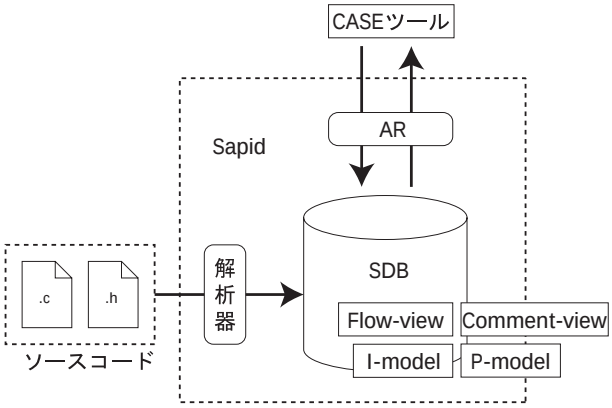


図 1: Sapid の提供するリポジトリの概略

関連 API を通して各種プログラミング言語からアクセスできるようになる．

しかし，SDB を XML 文書化したことによる弊害もある．例えば，ファイル間を跨がった宣言・参照関係の表現が困難になったことである．すなわち，SDB 上で表現できていた関係が XML 文書に変換したことで捨象されている．よって，コーディングチェッカーを作成する際の識別子に関するルールの記述が困難になる．また，AR にある各モデルに依存する高度な API の利用ができなくなるという問題点がある．よって，XML 文書にしたことにより汎用性は高まったが，これらの高度な API を XML 文書に再実装しなくてはならなくなる．

そこで既存の AR の機能を維持し，かつ開発言語に依存しないようなリポジトリへのアクセス方法として，SWIG (Simplified Wrapper and Interface Generator) [7] に注目する．SWIG は，C 言語または C++ で書かれたライブラリを，各種スクリプト言語の拡張ライブラリとして提供するためのラッパー生成器である．SWIG を用いることで AR をスクリプト言語にバインディングでき, スクリプトから SDB へのアクセスが可能となる (図 2).

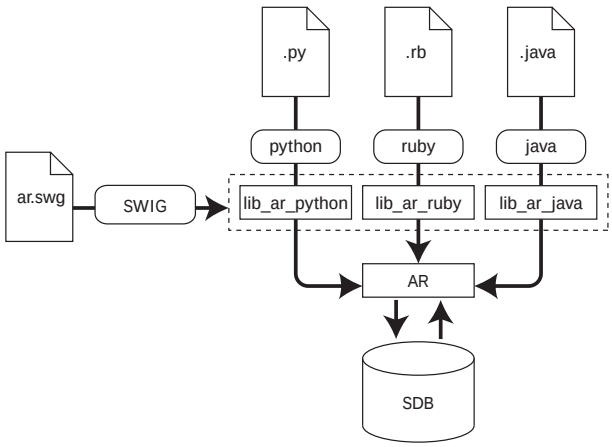


図 2: SWIG を用いたバインディング

# 4 コーディングチェッカー開発環境

スクリプト言語を用いたコーディングチェッカーを作成するために，SWIG を用いてラッパーを作成する．本稿では，スクリプト言語には Ruby を例に考える．Ruby はオブジェクト指向型言語であるため，C 言語の API である AR を オブジェクト指向プログラム化するため C++ でラップする．C++ を基に SWIG により Ruby のラッパーが生成されるが，両言語では命名規則が異なる．この差異を埋めるべく Ruby については Camel 形式からアンダースコアによる単語区切りに変換する機能が動的に働くようにする．さらに，実現したチェッカー開発環境を用いて Ruby によるチェッカーの作成を行う．

## 4.1 SWIG Wrapper

C 言語で書かれた API を C++ から利用できるようにする．Ruby をはじめ SWIG の対象言語の中にオブジェクト指向型言語が多いため C++ に変換することで他の言語向きのラッパーを作成する際の手間が省くことができる．

そこで，一度 AR を C++ 用に作成してから，SWIG に渡す手法をとる．以下のようなクラス群を作成した．

**SDB** SDB から解析情報を読み書き編集するクラス．

**Model** Sapid の表現するソフトウェアモデルを表すクラス．

**SapidClass, Relation** モデルの中で定義されるクラス・関連のクラス．

**SapidObject, Link** SDB の中に実際に格納されるオブジェクトを表すクラス．**SapidObject** は **SapidClass** の，**Link** は **Relation** の実体である．

## 4.2 名前変換

プログラミング言語ごとに，関数名，変数名といった識別子における命名規則が定められている．SWIG によって様々な言語へバインディングができるが，名前については，元の C++ の識別子のままである．例えば，C++ の`obj.getProgramName()` というメソッドは，Ruby ではメソッド呼び出しの'()' を省略し `obj.program_name` のように書く．これは，C++ でのフィールドへのアクセッサメソッドを Ruby ではフィールド値へのアクセスのように書く慣習があることによる．また，関数名は大文字で単語間の区切りを示す Camel 形式ではなく，アンダースコアによる単語区切りを用いる．

このようなメソッド名に関する変換はまず，Camel 形式をアンダースコアと小文字のみの形式に変換する．その後，Ruby の慣習を加味したメソッド名に変換する．そのために，先頭の単語に着目する．

- 動詞の原形から始まるものは，オブジェクトの振舞いを表すメソッドである．多くのメソッドがこの形式をしている．
- 先頭が `is`, 動詞 (三単現) から始まるものは述語関数とみなす．Ruby では真偽値を返すメソッドは名前の最後に '?' を付ける慣習があるため最後に付加する．

- get, set で始まるメソッドはフィールドへのアクセッサメソッドである．Ruby では，getter の get を取り除きフィールド名へのアクセスのように書く．setter は，set を取り除きフィールド名の最後に = を付け加える．すなわち，obj1.setAttr(val) というメソッドは，obj1.attr=(val) のように変換する．ただし実際には，これの糖衣構文である obj1.attr = val を用いる．
- Ruby には，基底クラス Object に to_s, hash, eql? というメソッドが定義されている．これらは連想配列と共に用いる際に呼び出されるメソッドであり適切に定義されているべきである．これら 3 つのメソッド名については個別に名前を変換する．また，Java 風のメソッド名にしておくことで Java バインディングを作成する際に有益となる．

以上を表 1 にまとめる．

表 1: メソッド名の変換例

| Java | Ruby |
|---|---|
| obj1.addAll(obj2) | obj1.add_all(obj2) |
| obj1.isEmpty()<br>obj1.contains(obj2) | obj1.empty?<br>obj1.contain?(obj2) |
| obj1.getAttr()<br>obj1.setAttr(val) | obj1.attr<br>obj1.attr=(val) |
| obj1.toString()<br>obj1.hashCode()<br>obj1.equals(obj2) | obj1.to_s<br>obj1.hash<br>obj1.eql?(obj2) |

### 4.3 ルールについての制限

コーディングチェッカーを SDB に基づいて作成する上で問題がある．Sapid はコンパイラ同様に字句・構文・意味解析を行う．そのためコンパイルができないソースコードについて Sapid は解析対象としていない．よって，MISRA のルール 9 「コメントは入れ子であってはならない」のようなルールについては，解析ができない．そのため今回対象とするコーディングチェッカー作成環境では最低限コンパイルが通ったソースコードを対象とすることとする．

## 5 MISRA–C への適用

本節では，MISRA–C のルール 59 「if, else if, else, while, do...while および for 文の本文を構成する文には必ず中カッコ {} で括られていなければならない」を検査するコーディングチェッカーを作成する．すなわち，図 3 の例において 4–5 行目の else 節が括られていないことを指摘できればよい．

Sapid を用いた CASE ツールは，SDB の中からいかに必要な情報を取得するかが問題となる．本検査ツールでは SDB から制御構造に関する部分を見付け，さらにその中で，{} で括られていない部分を見付けるという手順で必要な情報を取り出す．

図 4 に 図 3 のプログラムの I–model 表現 を示す．図中の矩形が SDB 内部でオブジェクトと呼ばれるものである．オブジェクト同士を繋いでいる辺が関連である．SDB はオブジェクトと関連のグラフ構造をしている．オブジェクトの一番上に書かれている名前が属しているクラス名になる．以下はオブジェクトの属性値である．点線で囲まれた部分の (a) が if 文に，(b) が then 節に，(c) が else 節に対応している．

```
1  int func(int n)
2  {
3      if (n == 0) {
4          return 0;
5      } else
6          return 1;
7  }
```

図 3: 検査対象プログラム

今回のルールに必要なクラスは，I–model の **block** クラスである．**block** クラスは (1) 制御構造，(2){} で括られた実際のブロック (**composite block**)，(3) {} で括られていない文の集合 (**basic block**) を表している．また，**block** にはそれ同士を結ぶ関連 **blk_blk** がある．この関連は，関数の本体を表すブロックと内部の，if 文を結ぶ関連といった構文上の入れ子関係を表す．

よって，if 文を表す **block** 構造の直下にある，**composite block** でない **block** オブジェクトを見つけることで，{} で括られている個所を見付けられる．図 5 のプログラムでは 11–13 行目で**block** の中から制御構造を抽出し (図 4 点線内の (a))，13–17 行目でそこを基点として直下の**basic block** を抽出している (点線内の (b)(c) のうち (c) だけを抽出)．

### 5.1 検証

既存のコーディング規約を基にしたチェッカーを多く提供することが作成環境に必要な要素の 1 つであったため，MISRA–C のコーディングチェッカーを作成する．

表 2 は，今回作成したコーディングチェッカーによって記述することのできたルールの内訳である．127 個あるルールのうち，93 個を記述することができた．大半が I–model を用いることで記述することができた．

記述できなかったルールの内訳は (1) 明確でないルール，(2) ソースコード以外に関するルール，(3) 解析できないルール の 3 つに分類できる．

**明確でないルール** ソースコードを解析することで関数の表面的な内容は捉えることができるが，その関数がどのような意図で呼び出されているかについては把握できない．このような関数の意図や役割については，仮定やユーザによって役割が与えられないとすべてについて解析することになる．例えば，ルール 86「関数がエラー情報を返り値とする場合，エラー情報はテストされるべきである.」というルールについて考える．このルールは関数がエラー情報を返り値として返すということが，ソースコードからだけではわからない．そのため，あらかじめそのような関数を挙げておき，それの関数についてのみエラーチェックがなされているか調べるという手法を取った．

**ソースコード以外に関するルール** MISRA のルールには，ANSI C の未定義な規格についての文書化の義務，利用するコ

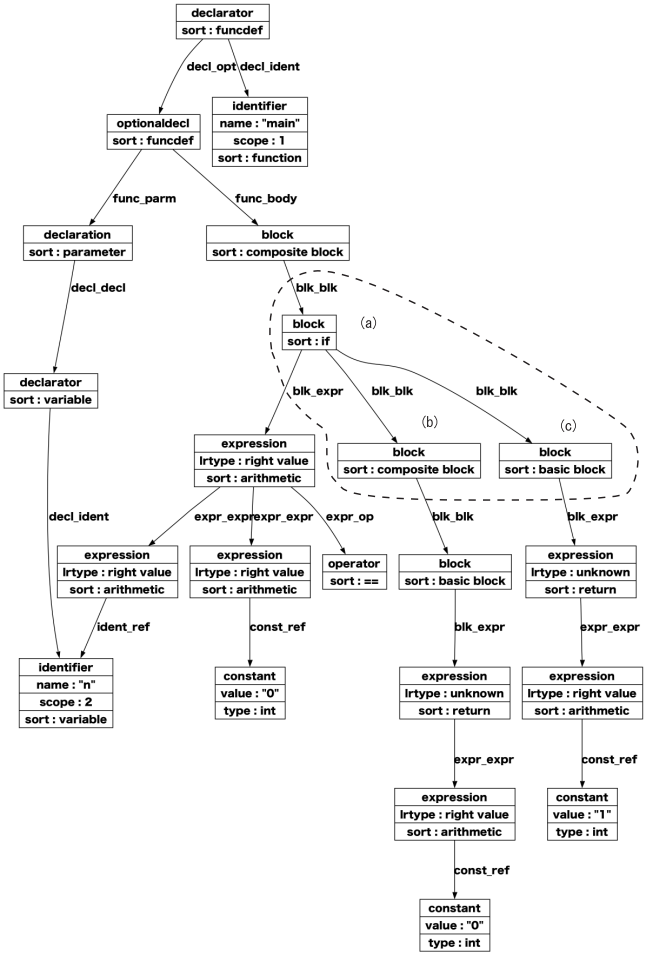


図 4: 図 3 の プログラムの I–model 表現

```
1  #!/usr/bin/env ruby
2
3  require 'sapid/ar5'
4  include Sapid::AR5
5
6  Imodel = Model.named('I-model')
7  IM_block = Imodel.sapid_class('block')
8  IM_blk_blk = Imodel.relation('blk_blk')
9
10 def get_uncomposite_blocks(sdb)
11    sdb.extent_of(IM_block).select do |b|
12       b.attr_enum_val_s('sort').to_s =~ /if|while|do|for/
13    end.collect do |b|
14       b.child_objects(IM_blk_blk).select do |cb|
15          cb.attr_enum_val_s('sort') == :"basic block"
16       end
17    end.flatten
18 end
19
20 def main(args)
21    SDB.open do |sdb|
22       sdb.read(Imodel)
23       p get_uncomposite_blocks(sdb)
24    end
25 end
```

図 5: MISRA 59 の検査プログラム

ンパイラ・リンカについての検査項目も含まれている．前者については，文書化を促すメッセージの表示の有無といった切替えをするという手法を取った．後者については人手での検査が妥当である．

**解析できないルール**　組込みソフトウェアでは必須のインラインアセンブラについては，Sapid では解析できない．また，ルール 10「コード部はコメントアウトしてはならない」については，コメントを解除してから再解析するといった近似解を提供するに留まっている．さらに，コンパイル可能かどうかの確認で検査ができるルールも多く存在する．これらについては対象としない．

表 2: MISRA–C のルールの対応状況 (モデルの利用は重複している)

| 対応 | 利用したモデル | ルール数 | |
|---|---|---|---|
| 記述可能 | I–model | 81 | 93 |
| | P–model | 14 | |
| | Flow–view | 6 | |
| | Comment–view | 0 | |
| 一部記述 | I–model | 14 | 15 |
| | P–model | 2 | |
| | Flow–view | 1 | |
| | Comment–view | 1 | |
| 記述できない | | | 8 |
| コンパイラを用いて発見 | | | 11 |

# 6　むすびと今後の展望

本稿では，スクリプト言語を用いたカスタマイズ可能なコーディングチェッカーを提案した．そのために，コーディングチェッカーに必要な解析情報を明らかにした．また，その解析情報を用いるために SWIG に着目し拡張ライブラリとしてスクリプト言語に提供することで，スクリプト言語の特長である rapid prototyping に適するという特性を生かせるコーディングチェッカーの作成環境が実現できた．さらに，実際に MISRA–C のコーディングルールの大半を記述できることを示した．

今後の課題は，他のコーディング規約についての対応である．本研究で作成したコーディングチェッカーは MISRA のルールを対象としたため，スタイル情報に関するルールが少ない．そのため，スタイルに関するチェッカーの作成が課題である．

他には，Sapid の提供するリポジトリが細粒度であるために必要な情報に辿り着くまでにノードを 1 つずつ辿らなかった．そのためソースコードに冗長な記述が増えるという問題があった．これについては，まず今回の経験を基に抽象度の高いモデルを作成することが考えられる．これとは別にグラフ上を走査する言語を組込むことが考えられる．これについては，XML の関連技術として XPath[11] や SeRQL[9] が挙げられる．

まず XPath は XML 文書中の要素の位置を記述するための言語である．しかし用途はそれだけでなくパターンマッチに用いることができ，XML 文書中から XPath 言語によってマッチする要素の集合を取得することができる．

次に，SeRQL は RDF (Resource Description Framework) に対するクエリ言語である．XPath が木を対象とした言語であるのに対し，SeRQL は RDF 同士を結ぶの関連によってできたグラフを対象とした言語である．そのため，SDB の走査言語への応用に参考なると考えている．

**謝辞** 御指導して頂いた愛知県立大学情報科学部の稲垣康善教授に感謝します．また，本システムの完成に多大な貢献をした同大学情報科学研究科の大橋台地氏に感謝します．また，本研究のベースとなる Sapid プロジェクトの皆様に感謝します．

# 参考文献

[1] GNU Coding Standards.
`http://www.gnu.org/prep/standards/`

[2] Indian Hill C Style and Coding Standards.
`ftp://ftp.cs.utoronto.ca/doc/programming/ihstyle.ps`

[3] MISRA. `http://www.misra.org.uk/`

[4] Perl. `http://www.perl.com/`

[5] Python Programming Language.
`http://www.python.org/`

[6] Sapid. `http://www.sapid.org/`

[7] SWIG (Simplified Wrapper and Interface Generator).
`http://www.swig.org/`

[8] オブジェクト指向スクリプト言語 Ruby.
`http://www.ruby-lang.org/`

[9] The SeRQL query language, rev. 1.1, November 2004.
`http://www.openrdf.org/doc/sesame/users/ch06.html`

[10] 組込みソフトウェア用 C 言語 コーディング作法ガイド (0.8 ドラフト版), 3 月 2005.
`http://www.ipa.go.jp/software/sec/download/files/report/200504/coding_guide.pdf`

[11] J.Clark. XML Path Language (XPath) Version 1.0, W3C Recommendation, 16 November 1999.

[12] 福安直樹, 山本晋一郎, 阿草清磁. 細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid. 情報処理学会論文誌, Vol. 39, No. 6, 1998.

[13] 渥美紀寿, 山本晋一郎, 阿草清磁. XML 記述によるソフトウェアリポジトリを用いたコード検索. 情報処理学会ソフトウェア工学研究報告, Vol. 2005, No. 149, pp. 57–64, 2005.