CX-Checker: C 言語プログラムのための カスタマイズ可能なコーディングチェッカ

大須賀俊憲[†1] 小林隆志[†1] 間瀬順一[†2]
渥美紀寿[†3] 山本晋一郎[†4]
鈴村延保[†5] 阿草清滋[†1]

本研究では，ソフトウェアの保守性・再利用性の向上を目的としたカスタマイズ性の高いコーディングチェッカ CX-Checker を提案する．CX-Checker はルールの複雑さに合わせて，XPath を用いたルール，DOM を用いたルール，ラッパーを用いたルールの 3 つのルールの実装方法を持つ．本論文では，CX-Checker の詳細を説明するとともに，MISRA-C と企業における実際のコーディングルールに対して適用した実験をもって実現可能性を評価し，その有用性を示す．

CX-Checker: A Customizable Coding Checker for C

Toshinori Osuka ,[†1] Takashi Kobayashi ,[†1] Junichi Mase ,[†2]
Noritoshi Atsumi ,[†3] Shinichirou Yamamoto ,[†4]
Nobuyasu Suzumura [†5] and Kiyoshi Agusa [†1]

This paper proposes a customizable coding checker “CX-Checker” which aims at improvement of maintainability and reusability of software. CX-Checker supports three type rule description such as XPath base, DOM base and wrapper API base. We introduce the details of CX-Checker and show its effectiveness with feasibility evaluations by adapting MISRA-C and the rules of an embedded software company.

1. はじめに

車載ソフトウェアに代表される組込みソフトウェアの開発では，厳しい時間制約，リソース制約，デバッグの難しさなどから保守性や再利用性を犠牲にせざるを得ない．例えばグローバル変数の使用は，組込みソフトウェア以外のソフトウェアにおいてはモジュールの独立性を下げ，ソフトウェアの品質を下げる原因として避けられるのが一般的である．しかし，グローバル変数はパフォーマンスやメモリ使用量の予測の容易さなどから組込みソフトウェアの開発の現場では使用されることも少なくない．

このようなソフトウェアに起こりうる保守性・再利用性の低下を避けるために，コーディング規約が広く用いられている[1)]．コーディング規約の例として GNU コーディングスタンダード[2)] や MISRA-C[3)] が挙げられる．コーディング規約を遵守することは，組込ソフトウェアの信頼性や再利用性の向上に貢献する．

開発したソフトウェアがコーディング規約に従っているかどうかのチェックはソースコードレビューやインスペクションなどによって人手で行われることもあるが，ソフトウェアの大規模化・複雑化に伴い，コーディング規約に違反する記述を自動的に検出するツールとしてコーディングチェッカが利用されることも多い．QAC[4)] や SQMlint[5)] がその代表である．これらのツールはソースコードを静的解析し，コーディング規約に違反する記述を検出する．コーディングチェッカを用いることでコーディング規約のチェックのコストを大幅に下げることができる．

組込みソフトウェアの用途が多岐に渡り，多くのプロジェクトが独自のコーディング規約を定めている．独自のコーディング規約は既存のコーディングチェッカを用いてチェックすることが困難であるため，現在でもコードレビューなどによる人手のチェックがなさ

†1 名古屋大学大学院情報科学研究科 Graduate School of Information Science, Nagoya University
†2 アイシン・コムクルーズ株式会社 AISIN COMCRUISE Co., Ltd.
†3 南山大学 情報理工学部 Faculty of Information Sciences and Engineering, Nanzan University
†4 愛知県立大学情報科学部 Faculty of Information Science and Technology, Aichi Prefectural University
†5 アイシン精機株式会社 AISIN SEIKI CO,. Ltd

れており，作業コストが問題となっている．

そのため，本研究では，アイシン精機株式会社のマネージメントのもと，カスタマイズ可能なコーディングチェッカ CX-Checker を開発した．CX-Checker は，(1)XPath を用いた方法，(2)DOM を用いた方法，(3)ラッパーを用いた方法の 3 種類のカスタマイズ方法を持ち，単純なコーディング規約から複雑なものまで対応できるという特徴を持つ．更に，XPath を用いたルール記述を容易にする補助インタフェースを持つ．

本論文では，CX-Checker を紹介し，その実用性を示すために行ったカスタマイズ能力の評価実験を説明する. 組込みソフトウェアの開発において広く用いられている MISRA-C のルールに対しては 62%(79/127) のルールが実現可能であった． これは SQMlint と同程度の能力である．アイシン精機株式会社の標準ルールでは，76%(13/17) のルールが実現可能だった．実現可能なルールの半数以上が，XPath により容易に実現可能だった．これらの評価から CX-Checker は，カスタマイズ能力が充分高く，独自のコーディング規約のチェック機能を容易に実現可能であることがわかった．

以下では，まず，既存のコーディング規約とコーディングチェッカを紹介し，3. で CX-Checker の概要と実現しているカスタマイズ機能について述べる．4. において, MISRA-C とアイシン精機株式会社の標準ルールを実際を実装する評価実験とその結果を説明し，CX-Checker のカスタマイズ機能が有効であり, 充分実用的であることを述べる．

2. コーディングチェッカ

開発したソフトウェアがコーディング規約を満たしているかを確認するために，レビューやインスペクションが行われる．しかし，ソフトウェアの規模が大きくなり，コストも増大する．

このため，コーディング規約のチェックを自動化するためのツールとして “コーディングチェッカ” が普及している．コーディングチェッカはソースコード群を入力とし，コーディング規約を満たしているかをチェックする．コーディング規約を満たしていない場合は，違反しているコーディング規約とソースコードの行番号を表示し，開発者に修正を促す．自動化されたツールを使用することにより，コーディング規約を満たしているかの確認をしやすくなる．大規模なソフトウェアに対してコーディング規約のチェックが可能になり，コーディング規約をより厳密に守りやすくなる．

2.1 コーディング規約

コーディング規約とは，ソースコードを記述する際に守るべきルールのことであり，プロジェクトごとで定義されている．コーディング規約の例として命名規約が挙げられる．識別子の命名規約をプロジェクトであらかじめ決めておくことにより，識別子が表す意味を変数名を見るだけで理解することでき，可読性が高くなる．K&R スタイル[6] や GNU Coding Standard 等のコーディングスタイルはコーディング規約の 1 つである．インデントや括弧の位置などを守るべき規約として定めることにより，ソースコードの可読性を高め，保守性や再利用性が向上する．

MISRA-C[3] は，欧州の自動車業界で発足された MISRA によって標準化された，組込みソフトウェアの信頼性向上のためのガイドラインである．MISRA-C は 127 個のルールからなり，環境，文字セット，コメント，識別子，型，定数，宣言と定義，初期化，演算子，変換，式，制御フロー，関数，前処理命令，ポインタと配列，構造体と共用体，標準ライブラリという 17 の観点からルールを定めている．特に，自動車業界では MISRA-C に従った開発を行うことが今後増えていくと考えられている．

2.2 既存ツール

2.2.1 SQMlint

SQMlint[5] はルネサステクノロジ社が開発している MISRA-C 専用のコーディングチェッカである．コンパイラのアドオンとして動作し，MISRA-C のルールを部分的に検査する．同社が提供しているコンパイラに組み込む形で利用でき，コンパイルしなければわからない変数のサイズや，マクロの展開後のソースコードを利用したルールがチェックできることが特徴である．

2.2.2 QAC

QAC[4] は Phaedrus Systems 社が開発している静的解析ツールである．約 1300 項目のコーディング規約をチェックできる．また，MISRA-C のルールの検査機能もアドオンとして提供している．プロジェクト全体を検査範囲とするような難しいルールにも対応し，組込みソフトウェアだけでなく，多くの C 言語で書かれたプロジェクトで利用されているツールである．

2.2.3 RainCode Checker

RainCode Checker[7] は，Ada，C/C++，COBOL 向けのコーディングチェッカである．RainCode Engine と呼ばれるソースコード解析ツールを用いて抽象構文木を作成し，その構文木から違反を検出するルールを作成できるのが特徴である．ルールの記述は，RainCode scripting language と呼ばれる ALGOL ライクのスクリプト言語を用いてプログラミングする．ホワイトスペースやコメントに対するルールは記述できない．

2.2.4 CheckStyle

CheckStyle[8] は，Java 向けのコーディングチェッカである．ホワイトスペースやコメントに対するルールも記述できるのが特徴である．抽象構文木に対して違反を検出するルールを記述することでルールを追加する．ルールは Visitor パターンを用いて Java 言語で記述する．

2.3 既存ツールの問題点

これらのコーディングチェッカは世界中のプロジェクトで幅広く使用されているがカスタマイズ性が高くないことが問題として挙げられる．既存のツールでもルールごとに有効無効を切り替えることができるが，プロジェクト毎に独自のコーディング規約が存在することも少なくない．RainCode Checker や CheckStyle では，独自のコーディング規約をチェックするルールを追加することができるが，ルール記述に利用可能な構文要素が限定される，ルールの記述方法が容易でないなどの問題がある．

3. CX-Checker

2. の最後で述べた問題点を解決するために，柔軟なカスタマイズ機能を有するコーディングチェッカ “CX-Checker” を開発した．CX-Checker は以下の特徴を有する．

- 容易にルールを追加・変更可能な言語を持つ
- ルール追加を補助するインタフェースを持つ
- 複雑なルールにも対応可能なインタフェースを持つ
- 制御フローを使ったルールにも対応可能である

本章では，CX-Checker の概要について述べ，ルールのカスタマイズ方法について述べる．

3.1 概　　要

CX-Checker の概要を図 1 に示す．CX-Checker は検査したいソースコード群とルール群を入力とし，検査を実行する．ソースコードは内部で CASE ツールプラットフォーム Sapid[9] の CX-model[10] に変換される．CX-model は C 言語の抽象構文木を XML で表現したものである．詳細は 3.2. で述べる．CX-Checker では CX-model の XML に対して違反を検出するルールを記述する．ルールの記述方法は 3.3. で述べる．

CX-Checker は CUI のインタフェースと Eclipse プラグインのインタフェースの 2 つを持つ．Eclipse プラグインのインタフェースの動作イメージを図 2 に示す．

中央上部のビューはソースコード上にルールに違反した部分を黄色の波線でハイライトするソースコードビューである．ハイライトされている部分にマウスカーソルを合わせると違反しているルールの説明が表示される．中央下部の問題ビューは検出した違反の一覧とその説明を一覧にして表示する．

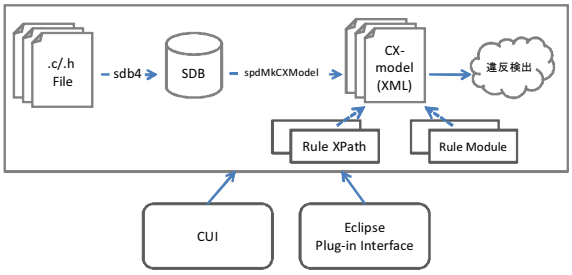

図 1　CX-Checker の概要

CX-Checker は Java 言語と XML で書かれており，規模は約 10,000 行である．利用ライブラリは Eclipse SDK, Sapid, Saxon[11] である．

3.2 CX-model によるコーディング検査

本節では，CX-Checker のコアとなる CX-model について述べる．例として C 言語のソースコードとそれを変換した CX-model の XML を以下に示す．示した XML は読みやすくするため整形してある．

```
1  main() {
2    int a = 0;
3  }
```

```
1  <File id="s8388608">
2    <Function id="s33554432">
3      <ident defid="s33554432">main</ident>
4      <op>(</op>
5      <op>)</op>
6      <op>{</op>
7      <nl line="1" offset="7">
8      </nl>
9      <sp>    </sp>
10     <Local id="s33554433">
11       <Type>
12         <kw sort="type">int</kw>
13       </Type>
14       <sp> </sp>
15       <ident defid="s33554433">a</ident>
16       <sp> </sp>
17       <op>=</op>
18       <sp> </sp>
19       <literal defid="s75497472">0</literal>
20     </Local>
21     <op>;</op>
22     <nl line="2" offset="22">
23     </nl>
24     <op>}</op>
25   </Function>
26   <nl line="3" offset="24">
27   </nl>
28 </File>
```

CX-model は C の前処理前のソースコード上に構文解析結果をマークアップしたものである．そのため，解析対象のソースコードは，マクロ呼び出しを変数あるいは関数呼び出しに置き換えたときに C 言語の

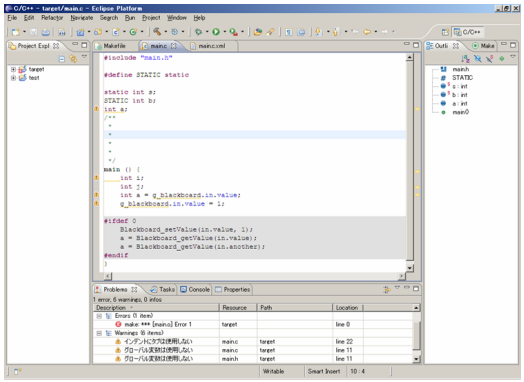

図 2　Eclipse プラグインインタフェースの動作イメージ

文法に従っていなければならないという制限がある．CX-model は 30 個の構文要素 (うち 13 個が終端要素) からなる．定義箇所の識別子には一意の ID がつき，参照箇所の識別子で同じ要素を表す場合には同じ ID がつく．また，構文解析の情報にコメントやスペース，改行など，プログラムの振る舞いに直接関係がない要素もモデルに含まれるという細粒度リポジトリであるため，XML タグに囲まれた部分のみを残し，それ以外を除去すると元のソースコードに戻るという特徴を持つ．

例えば，すべての関数の定義部分を取得する場合には，XML 中のすべての Function 要素を取得すればよい．また，ある関数が呼ばれている箇所を取得する場合には，関数の id 属性の値を defid 属性に持つ Expr 要素を取得すればよい．

3.3　ルールの定義方法

CX-Checker は以下の 3 つの方法で独自ルールを追加できる．

- XPath を用いたルール
- DOM(Document Object Model) を用いたルール
- ラッパーを用いたルール

XPath が最も容易な方法であり，単純なルールに向く．DOM とラッパーを用いたルールは複雑なルールに向く．以下ではそれぞれのルールの実装方法を例を用いながら説明する．

3.3.1　XPath を用いたルール

本節では XPath を用いたルールの実装方法について述べる．XPath は XML Path language の名の通り，XML の要素をパスを表す言語を用いて取得する言語である．

XPath の例を以下に示す．

```
1 /File/Function/Local/ident
```

この例では，ルートの File 要素から順に直下にある Function 要素，その直下にある Local 要素という順番で下り，更にその直下にある ident の要素の集合を取得するという意味である．

XPath には条件を満たす要素のみを取得する述語や関数が使用できる．関数の例は，文字列が指定された文字列から始まるかどうかを starts-with 関数や，文字列の長さを返す string-length 関数などがある．CX-Checker は XPath を用いてルールを記述する．検出したいソースコードのパターンを対応する CX-model の要素を取得する XPath で記述する．

ルールの例を以下に示す．

```
1 //sp[contains(text(),"&#x9;")]
```

この例はインデントにタブ文字を指定している場合に，違反を検出するルールである．タブ文字は，環境によって大きさが違うため，スペースと混在した場合にインデントが乱れて，可読性が下がる恐れがある．その問題を避けるためにスペースを利用するべきであるというルールである．

XPath 中の “//sp” はすべての sp 要素を取得する XPath である．CX-model は sp はホワイトスペースを表す．“[...]” は述語であり，中に書かれた条件を満たす要素を取得する．述語中では contains 関数を用いて，テキストがタブ文字 (“	” はタブ文字の実態参照) を含むものを取得している．

上記のように XPath を用いたルールは，単純なルールに対してわずかな記述でルールを実現できる．

3.3.2　DOM を用いたルール

XPath を用いたルールはコーディング規約が単純な場合に効果を発揮する．しかし，XPath で表現できないコーディング規約も多くある．例えば，正規表現を用いたルールや，仕様を書いた別の XML ファイルを参照しながら検査を行うルールなどは XPath だけでは実現できない．そこで CX-model を DOM で操作するプログラムを記述することでルールを実現する方法を CX-Checker は提供する．

DOM を用いたルールは Java 言語のクラスとして実装する．実装例を以下に示す．

```
1 public class AnRule implements CheckerClass {
2   public List<Result> check(IFile file, CheckRule rule) {
3     List<Result> results = new ArrayList<Result>();
4     Document doc = file.getDOM();
5     ...
6     return results;
7   }
8 }
```

DOM ルールを実装する際には CheckerClass インタ

フェースを実装し，唯一のメソッドである check メソッドを実装する．引数には対象ファイルである IFile インタフェースが渡され，IFile から DOM を取り出してチェックを行う．もう一つの引数である CheckRule クラス は検査に使用されるルールの一覧を保持しているため，複雑なルールを単純なルールの組み合わせとして実現することができる．check メソッドは戻り値として Result クラスのリストを返す．ルールでは，違反を検出した場合に Result クラスのリストに違反したソースコードの位置 (先頭からのバイト数と長さ) の情報を追加していく．

ルールをプログラムで記述することにより，外部のファイルと連携するルールや，正規表現を用いるような条件が複雑で XPath では実現できないようなルールに関しても対応できる．

3.3.3 ラッパーを用いたルール

DOM を用いたルールは，仕組みも単純であり，柔軟性にも優れているが，CX-model に習熟していない開発者にはわかりにくいという問題点がある．例えば，関数定義から関数名を取得するためには Function 要素を取得し，直下の最初に出現する ident 要素を取得し，そのテキストノードを取得するという手順を取る．この方法は CX-model に習熟している開発者には自然であるが，そうでない場合はわかりやすいとはいえない．

そのため，CX-Checker では，これらの手順を隠蔽し，Java 言語のオブジェクトとして扱うためのラッパーインタフェースを提供する．ラッパーを用いたルールでは，まず IFile から取得できる DOM から CFileElement のインスタンスを作る．CFileElement は CElement を継承しており，CElement は関数を表す CFunctionElement など様々な情報を取得するメソッドを持つ．以下にサンプルコードを示す．

```
1 CFileElement cfile = new CFileElement(file.getDOM());
2 for (CFunctionElement function : cfile.getFunctions()) {
3     System.out.println(function.getName());
4 }
```

この例では，ファイルを表現する CFileElement クラスから CFunctionElement のインスタンスを取得した後，getName メソッドを呼ぶことで関数名を取得している．ラッパーを用いることにより，CX-model に習熟していない開発者でもルールを記述することができる．

3.3.4 ルール開発支援機能

CX-Checker は 3 種類のルール開発インタフェースを持つが，以下の 2 つの問題があった．

(1) XPath によるルール開発には XPath 及び CX-model の知識が必要である

(2) 制御フローを用いるような複雑なルールに対応するためには労力がかかる

問題 1 の解決のために，XPath ルール補助インタフェースを持つ．問題 2 の解決のために，PathGraph インタフェースを持つ．以下では，それぞれの機能について詳細を述べる．

XPath ルール補助インタフェース

XPath ルールの作成は単純であるが CX-model の知識が必要である．そこで CX-model に習熟していない開発者の XPath ルールの開発を補助するインタフェースを持つ．図 3 にインタフェースを示す．

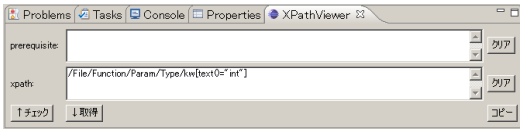

図 3 XPath ルール補助インタフェース

prerequisite 入力ボックスは，複雑になるほど長くなる XPath を理解しやすいように分割するための前提条件を記述することができる．つまり，prerequisite に入力された XPath にマッチしたノードをコンテクストノードとして，そこからの XPath を記述できる．「チェック」ボタンは入力された XPath を用いて検査を実行する．この機能により，ルール実装に必要な試行錯誤を即座に実行できる．「取得」ボタンはソースコードビュー上で，検出したい部分を選択した状態で押すことにより，その部分を取得する XPath を自動生成する．この機能により，ソースコードと CX-model の間を埋めることができる．

PathGraph

CX-model は構文解析結果のみを持つため，制御フローを用いたルールを開発するためには大きな労力がかかる．このため，制御フローを用いたルールを開発する場合に便利なインタフェースを CX-Checker 制御フローを用いたルールの例として MISRA-C のルール 30「すべての自動変数は使用する前に値を代入しなければならない」が挙げられる．以下に MISRA-C ルール 30 に違反する例を示す．

```
1 int foo;
2 if (bar == 0) {
3     foo = 1;
4 } else {
5     bar = 1;
6 }
7 return foo; /* NG */
```

1 行目で宣言された変数 foo は if 文の else 節を通った場合に代入されないまま 7 行目で使用されてしまう．変数の初期値は定められていないため，環境によって

違う場合があり予期せぬ動作を引き起こす原因となる．

CX-Checker は通りうるすべてのパスをグラフにして提供するインタフェースを持つ．上記の例を用いて本インタフェースについて述べる．すべてのローカル変数と文をノードとし，実行されうる順にエッジを張る．この例では図 4 のようにノードが作られ，1 → 2，2 → 3，3 → 4，3 → 5，4 → 6，5 → 6 というエッジが張られる．

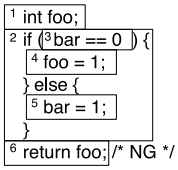

図 4 制御構造を伴うソースコード

ルール開発者はグラフを用いて，すべての実行されうるパスを取得し，すべてのパターンにおいてコーディング規約を満たしているかをチェックすることができる．CX-Checker では if 文だけでなく，for 文，while 文，do-while 文，switch 文などの制御構造に対応しており，制御構造の入れ子にも対応している．ただし，ループは 0 回か 1 回の繰り返しのパスのみを取得する．

PathGraph インタフェースのサンプルコードを示す．

```
1 PathGraph graph = new PathGraph(function);
2 List<List<GraphNode<Element>>> paths = graph.toPathList();
```

開発者は CFunctionElement クラスのインスタンスから，PathGraph クラスのインスタンスを取得する．この例では，PathGraph から実行されうるパスの集合を取得しているが，PathGraph をグラフのまま直接操作することもできる．例えば，ソースコードには含まれるが PathGraph には含まれない文を探せば，到達しない文を検査することができる．

4. 記述性の評価

本章では，CX-Checker のカスタマイズ能力が実用的なレベルであることを証明するために，組込みソフトウェアで利用されることが多い MISRA-C のルールとアイシン精機株式会社の AT(オートマティックトランスミッション) ソフトウェアのコーディング規約を CX-Checker 上に実現できるかを調査した．また，いくつかのルールを実際に実装することにより，ルール開発の容易さを確認した．

4.1 MISRA-C の実装

MISRA-C の全ルール 127 個に対して実現可能性を調査した．主要なコーディングチェッカとの比較結果を表 1 に示す．表中の括弧内の数字は，実現可能なルールのうち，XPath により実現可能と判断したルールの数である．

この結果から，QAC とは対応可能なルールに大きな差があるものの，SQMlint と同程度の対応が可能なインタフェースを CX-Checker が持つことがわかる．また，実装可能なルールのうち，約 6 割が XPath による記述で実現できることがわかった．つまり，実用的なルールのうち，半数がプログラムを記述することなく XPath によるルールで実現できるといえる．

表 1 主要なコーディングチェッカの MISRA-C 対応ルール数

	QAC	SQMlint	CX-Checker
対応可能	118	86	79(45)
対応不可能	9	41	48
合計	127	127	127
割合 (%)	93	68	62

4.1.1 XPath ルールの実装

上記で実現可能と判定したルールのうち，XPath で実現可能であると判断した 45 個を実装した際の所要時間について述べる．実装者は CX-model と XPath について基本的な知識のみを持っている大学院生である．

表 2 に 45 個の XPath ルールの実装にかかった時間を載せる．XPath で実装可能であると判断されたルールのうち約 49%(22/45) のルールが 5 分未満で実装可能だったことがわかる．

表 2 XPath ルールの実装の所要時間

所要時間	ルール数
5 分未満	22
5-15 分	10
15-30 分	6
30-60 分	2
実装できず	5
合計	45

また，ルールの実装ができなかった主な理由は，実装者の知識不足であった．XPath では兄弟ノード同士の順番を判定する際に following-sibling や preceding-sibling というキーワードを用いるが，実装者はこのキーワードを知らなかった．

4.2 アイシン精機株式会社の標準ルールの実装

アイシン精機株式会社の AT ソフトウェアで使用されているコーディング規約の一部について同様に実現可能性を調査した．対象としたのは 17 個のルールである．一部としたのは，MISRA-C と重複したルールを除いており，また MISRA-C ルール 41「選定したコンパイラの整数除算の実装を確認し，文書化し，考慮すべきである」のようにソースコードに対する規約ではなくプロジェクトに対する規約を除いたためである．

実現可能性の結果は，76 %のルールに対して実現可能であった．実現可能なルールのうち 54 %のルールが XPath によるルールで実現可能であった．QAC と SQMlint はこれらのルールに対応していない．

4.3 考　　察

CX-Checker のカスタマイズ性能を MISRA-C のルール及び，アイシン精機株式会社で実際に使われているコーディング規約において実現可能性を調査することにより評価した．MISRA-C で 6 割，アイシン精機株式会社のルールで 7 割のルールに対応可能であることがわかり CX-Checker のカスタマイズ性能は非常に高いものであるといえる．

また，XPath で記述できると判断されたルール 45 個のうち，49%のルールが 5 分以内に実装できるということがわかった．XPath の記述性は充分に高いものであると言える．

対応できなかったルールの原因の多くは以下の 3 つが挙げられる．

- 型情報を用いるルールであった
- マクロの本体の解析を行うルールであった
- 複数ファイルに跨る情報を用いるルールであった

型情報を用いるルールとは，例えば unsigned な型を持つ変数と負の数を比較してはならないといった変数の型情報を用いて検査を行うルールである．C 言語では，型名が違っていても符号とサイズが同じであれば同じ型として扱うことがあるという点が挙げられる．このため，字面上の型名を取得して比較するだけではなく，真の型が同じかどうかを判定する必要もある．これらの理由から CX-Checker は型情報を用いたルールに対応していない．

マクロの本体の解析を行うルールとは，#define A B というマクロがある場合に，B の内容を解析するようなルールである．B の解析にはマクロがどのように展開されるかという情報が必要であるが，CX-Checker は前処理前のソースコードを対象に解析を行うため，マクロの定義に関するルールのうち，マクロがどのような場所に展開されたかという情報を用いるルールについては対応ができない．

しかし，SQMlint は前処理後のソースコードを解析対象としているため，マクロに関するルールは対応していない．前処理前を対象とするか後を対象とするかによって解析するルールが変わるため，一概に弱点とは言えない．

複数ファイルに跨る情報を用いるルールとは，二つ以上のファイルを同時に解析する必要があるルールである．現在の CX-Checker では，実装上の制限により，同時に一つのファイルしか解析できない．そのため，ソースファイル上のある変数の宣言が別のヘッダファイル上にある場合に CX-Checker はその宣言を取得できない．

XPath で記述できるような特定の構文要素に依存するコーディングルールは，Visitor パターンによって構文木を走査し，ルールをチェックする手法も考えられる．Visitor パターンでは複雑な条件を指定することができるが，走査中のノードと先祖ノードとの比較をすることは容易ではない．これに対して，XPath では複雑な条件を指定することはできないが，走査中のノードと先祖ノード，兄弟ノード，従兄弟ノードなどとの比較が容易にできる．これらの手法間におけるルールの記述性に関する評価は今後の課題である．

5. おわりに

本論文では，組込みソフトウェアの再利用性と保守性を向上させるためにコーディング規約が重要であることを述べ，コーディング規約の検査にコーディングチェッカが有用であることを述べた．既存のコーディングチェッカのカスタマイズ性が乏しいことを問題とし，カスタマイズ性の高いコーディングチェッカとして CX-Checker を開発した．

CX-Checker はルールの複雑さに合わせて，XPath を用いたルール，DOM を用いたルール，ラッパーを用いたルールの 3 つのルールの実装方法を持つ．また，MISRA-C とアイシンルールに対して実現可能性を評価することにより，CX-Checker は実際のプロジェクトで使われているルールを実装するために十分なカスタマイズ性を持つことを明らかにした．特に実際のプロジェクトで使われているルールを実現するには 3 つのうち最も単純な XPath によるルールでも十分であることが多いことがわかった．

これらの事実から CX-Checker は実際のプロジェクトに耐えうる高いカスタマイズ性を持つといえる．CX-Checker は組込みソフトウェアの保守性・再利用性の向上を目的として開発したが，組込みソフトウェアだけでなく，その他の C 言語で記述されたソフトウェアにも適用可能だと考えられる．本研究では，組込みソフトウェア以外のソフトウェアのためのコーディングルールに対する記述能力の評価を行っていないため，その有効性を判断することはできない．さらに，XPath によるルールの実装方法は，C 言語に限らず，他の言語にも適用可能である．

5.1 今後の方向性

CX-Checker で対応できなかったルールのうち，型情

報とマクロの展開情報に対応するためには CX-model を拡張する必要がある．Sapid のソースコード解析では，構文木，定義参照関係，型，依存関係，前処理などの情報を解析結果として保持しているが，現在の CX-model では XML で容易に表現可能な，構文木とスコープ規則に従った定義参照関係しか表現できていない．残りの情報を XML で表現することにより，対応できなかったルールに対応することが可能となるが，これらの情報の XML での表現方法は興味深い課題である．以下に対応できなかったルールに対する解決方法を示す．

ファイルを横断する解析の対応

CX-Checker の現在の実装ではファイルを横断する解析ができない．例えば，プロトタイプ宣言をヘッダーファイルでされた場合に，ソースファイルからその宣言を取得することはできない．しかし，コーディング規約はヘッダーファイルも同時に見ながら解析を行わなければならないものも存在する．

そこで，検査対象のファイルの include 宣言の部分にヘッダーファイルのモデルを展開したものを渡す方法が考えられる．この方法を用いれば 1 つのファイルを操作するインタフェースを変えずにファイルを横断する解析を行える．このとき，同じ箇所を何度も違反として検出することを避ける仕組みが必要である．

型情報を取得するインタフェースを追加

CX-Checker の現在の実装では，型情報を取得するインタフェースが無い．この問題に対して，CX-model を拡張するアプローチを考える．CX-model のすべての ident 要素の属性に型情報を付与することにより，変数の型情報を取得できる．

CX-model を拡張するアプローチによって型情報を用いたルールに対応できるようになる．しかし，型情報としてどのような情報を提供するかは難しい問題である．例えばあるローカル変数が型 T で宣言されていてその T は typedef により unsigned char と宣言されているとする．このとき，型情報として T を返すべきか unsigned char を返すべきかという問題がある．この問題については，上記アプローチの他にどのようなインタフェースがあればルールを記述しやすいのかを深く考慮する必要がある．

前処理後のプログラムも検査対象にする

CX-model では，前処理前のソースコードのみを解析対象としている．SQMlint では，前処理後のプログラムを解析対象としているため，マクロに関係するルールには対応できないという弱点があるのに対し，CX-Checker はマクロに関連するルールにも対応可能であるためこの特徴はメリットであるとも言える．しかし，より精密な検査を行いたい場合は前処理後のプログラムに対して検査を行う必要がある．例えば，define キーワードで変数の名前替えをしている場合，前処理後のプログラムを解析対象にできれば，情報を正しく取得できる．

前処理後のプログラムを解析対象とするために，前処理前の C ファイルを解析した CX-model にとは別に，前処理後の I ファイルを解析した結果を “IX-model” として提供するアプローチを考える．IX-model は CX-model の要素のうち，マクロに関係する要素である Define 要素，macroPattern 要素，macroBody 要素，macroCall 要素の 4 つの要素を除いたモデルである．開発者がルール開発時に CX-model と IX-model の両方にアクセスすることができるようになればルールの記述性が向上すると考えられる．

謝辞　本ツールの開発は，アイシン精機株式会社の協力の得て，名古屋大学大学院情報科学研究科 IT スペシャリストコースにおけるソフトウェア工学実践研究 OJL(On the Job Learing) のテーマとして実施された．プロジェクト関係者に深く感謝の意を表す．本研究の一部は科研費 (19700023，20300009) の助成による．

参　考　文　献

1) Pete Goodliffe. Code Craft –エクセレントなコードを書くための実践的技法–. 毎日コミュニケーションズ, 2007.
2) GNU Coding Standard. `http://www.gnu.org/prep/standards/`.
3) MISRA-C 研究会. 組込み開発者におくる MISRA - C-組込みプログラミングの高信頼化ガイド. 日本規格協会, 2004.
4) 静的解析ツール QAC. `http://www.toyo.co.jp/ss/qac/`.
5) ルネサス テクノロジ - MISRA C ルールチェッカ SQMlint. `http://japan.renesas.com/`.
6) BrianW. Kernighan and DennisM. Ritchie. *C programming language 2nd ed.* Prentice Hall, 1988.
7) RainCode Checker. `http://www.raincode.com/checker.html`.
8) Checkstyle. `http://checkstyle.sourceforge.net/`.
9) 福安直樹, 山本晋一郎, 阿草清滋. 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, pp. 1990–1998, 6 1998.
10) 渥美紀寿, 山本晋一郎, 阿草清滋. 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. 情処研報, 2005-SE-149, pp.57–64, 2005.
11) Saxon. `http://saxon.sourceforge.net/`.