情報処理学会論文誌 Vol.53 No.2 590–600 (Feb. 2012)

CX-Checker：柔軟にカスタマイズ可能なC言語プログラムのコーディングチェッカ

大須賀 俊憲[1] 小林 隆志[2,a)] 渥美 紀寿[2] 間瀬 順一[3] 山本 晋一郎[4]
鈴村 延保[5] 阿草 清滋[2]

受付日 2011年5月30日, 採録日 2011年11月7日

概要：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．

キーワード：コーディングチェッカ，コーディング規約，静的解析，プログラム解析

CX-Checker: A Flexibly Customizable Coding Checker for C

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

Received: May 30, 2011, Accepted: November 7, 2011

Abstract: 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.

Keywords: coding checker, coding rule, static analysis, program analysis

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，応答性能などの厳しい時間制約，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えない場合もある．たと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，組込み分野の開発においては，通常のデバッガが利用できず ICE（in circuit emulator）などの特殊なデバッグ環境を用いる必要があり，ICE によって値を状態信号として確認するために，アドレスが固定された変数としてグローバル変数が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] があげられる．コー

1 株式会社ネットレックス
NETREQS Co., Ltd., Koutou, Tokyo 135–0044, Japan
2 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University, Nagoya, Aichi 464–8601, Japan
3 アイシン・コムクルーズ株式会社
AISIN COMCRUISE Co., Ltd., Nagoya, Aichi 450–0002, Japan
4 愛知県立大学情報科学部
Faculty of Information Science and Technology, Aichi Prefectural University, Nagakute, Aichi 480–1198, Japan
5 アイシン精機株式会社
AISIN SEIKI CO., Ltd., Kariya, Aichi 448–8650, Japan
a) tkobaya@is.nagoya-u.ac.jp

ディング規約を遵守することは，組込み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・複雑化にともない，ソースコードを静的解析することで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であるコーディングチェッカが利用されることも多い．QAC や SQMlint がその代表である．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わたり，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，開発プロジェクトごとに容易に独自コーディング規約を運用する支援を目的として，カスタマイズ可能なコーディングチェッカ CX-Checker を提案する [4]．提案手法の特徴は，C ソースコードを XML 表現することにより，ソースコードに関する特徴を，XML 関連の標準技術を利用して記述・発見することを可能にする点である．このことにより，利用者が独自のコーディング規約を容易に運用することが可能となる．

提案手法では，空白文字やコメントまでを含めたテキスト情報をすべて保存しながら，型情報や構文情報のタグ付けを行う CX-model [5] を利用して XML 形式で表現する．XML 形式を用いることで，広く一般的に利用されている XML 関連の標準技術が利用できるため，ソースコードを操作するために特別な記法を修得する必要がないという利点がある．また，CX-model ではソースコードの代表的な特徴である構文木に関する情報を中心にタグ付けが行われるため，XPath を用いて少ない記述量で構文木における位置情報を表現することができるという利点もある．

CX-Checker は，(1) XPath を用いた方法，(2) DOM を用いた方法，(3) ラッパを用いた方法の 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をユーザが独自に定義することができる．さらに，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代表的な商用ルールと同等の 80.3%（102/127）のルールが実現可能であった．さらに，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．これらの評価から CX-Checker は，カスタマイズ能力が十分高く，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分かった．

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する．提案手法の基盤となる C ソースコードの XML 表現 CX-model に関して 3 章で説明し，4 章で 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．5 章において，MISRA-C とアイシン精機株式会社の標準ルールを実際に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，十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すか確認するために，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るに従い，そのコストも増大する．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あげられる．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，変数名を見るだけで識別子が表す意味を理解でき，可読性が高くなる．K&R スタイル [6] や GNU Coding Standard などのコーディングスタイルもコーディング規約である．インデントや括弧の位置などを規約として定めることで，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 既存研究・ツール

SQMlint 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分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．

QAC は Programming Research 社が開発している静的解析ツールである．約 1,300 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．

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．

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という特徴を有する．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ことが可能である．ルールは Visitor パターンを用いて Java 言語で記述する．

岩手県立大学の研究グループが開発しているツール [9] では，C ソースコードの構文木情報の S 式表現である SXML 記述 [10] に対してコードパターンの発見を行うことができる．パターン記号と呼ばれる，代表的な構文要素に対する正規表現に似た記述言語を導入することにより，利用者がコーディングパターンを定義することができる．

2.3 既存研究・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あげられる．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，プロジェクトごと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，専用の手法や言語を用いるためルールの記述方法が容易でないなどの問題がある．

3. CX-model: C ソースコードの XML 表現

3.1 必要となる情報

プログラムソースコードに対して，タグを付けることによって XML 表現を行う研究は数多く行われている [11], [12], [13], [14], [15], [16]．これらの多くの研究では，構文要素をその種類ごとにタグ付けし，属性を用いて定義参照関係を表現しているのみであり，CX-Checker のような CASE ツール開発に有用な情報を十分に表現できていない．たとえば，識別子のデータ型に関する情報は，その識別子の定義箇所を調べ，そのデータ型を参照する必要がある．また，データフローや制御フローの情報は表現されておらず，それらの情報を取得するためには XML で表現されたソースコードを解析しなければならない．

本研究の対象である C ソースコードの XML 表現では，GCC-XML [16] や srcML [15] のような粗粒度の解析しかしないツールや ACML [14] のように前処理後のコードを対象に解析するツールが多く，前処理前のコードを細粒度にマークアップするツールは少ない．

我々が提案する CX-model [5], [17] では，C 言語の前処理前のコードに対して，構文構造と定義参照関係，データ・制御フロー情報，型情報を XML を用いて表現する．CX-model では，ソースコードを解析するうえで最も基本的な特徴である構文木情報に加え，空白やコメントなどソースコード中に存在するテキスト情報を重視し，ソースコードの構造を保持したままマークアップする．このため，XML 要素を取り除くことによって，元のソースコードに復元することができる．Badros はタグを付加するだけでマークアップした表現では，テキスト部分を再度構文解析する必要があると指摘している [18]．我々はこの問題を解決するために，細粒度にタグ付けし，いくつかの属性を付与している．

以下では，構文構造，フロー情報，型情報についての概要を述べる．CX-model の詳細な定義は DTD [19] を参照されたい．

3.2 構文構造の表現

構文構造は `File` 要素で表現し，構文要素の種類ごとに XML 要素が定義されており，各要素にはそれぞれいくつかの属性が定義されている．`sort` 属性は要素の種類を示す．`defid` 属性は識別子の定義要素の識別 ID を示し，この ID によって定義参照関係を表現する．`type_id` 属性は識別子のデータ型を示し，その詳細の情報は `TypeInfos` 要素で表現される．

図 2 は **図 1** を CX-model に変換した結果の一部である．図 2 の `Local` 要素は図 1 の 2 行目のローカル変数の宣言を，`Stmt` 要素は 5 行目の “`ans += tmp`” を表している．

```
1  int sum(int x) {
2     int ans = 0, tmp;
3     while (x > 0) {
4         tmp = x--;
5         ans += tmp;
6     }
7     return ans;
8  }
```

図 1 サンプルコード
Fig. 1 A sample code.

```
1  <File>...
2    <Local id="s34">
3      <Local id="s36">...
4        <ident defid="s34" type_id="s81">ans</ident>...
5        <ident defid="s35" type_id="s81">tmp</ident>...
6      </Local>...
7      <Stmt id="s66">
8        <Expr id="e66" sort="AssignADD">
9          <Expr id="e64" sort="VarRef">
10           <ident defid="s34" type_id="s81">ans</ident>
11         </Expr><sp> </sp><op>+=</op><sp> </sp>
12         <Expr id="e65" sort="VarRef">
13           <ident defid="s35" type_id="s81">tmp</ident>
14         </Expr>
15       </Expr>
16     </Stmt>...</File>
```

図 2 サンプルコードの CX-model の構文構造表現（抜粋）
Fig. 2 A part of CX-model representation of the syntax structures for the code.

```
1  <flow id="f01" stmt_id="s50" next="s49" sort="branch_true"/>
2  <flow id="f02" stmt_id="s50" next="s68" sort="branch_false"/>
3  <Stmt sort="While" id="s50"><!-- while (...) {...} -->
4     ...
5     <flow id="f004" stmt_id="s49" next="s63" sort="control_normal"/>
6     <Stmt sort="Block" id="s49">   <!-- {...} -->
7          <flow id="f08" stmt_id="s63" next="s66"
8                sort="data_dependence"  expr_id="e65" dep_id="s32"/>
9          <Stmt id="s63">...</Stmt><!-- tmp = x--; -->
10         ...
11         <Stmt id="s66">...</Stmt><!-- ans += tmp; -->
12    </Stmt>...
13    <Stmt id="s65">...</Stmt><!-- return ans; -->
14 </Stmt>
```

図 3 サンプルコードに対する CX-model のフロー情報表現（抜粋）
Fig. 3 A part of CX-model representation of the flow information for the code.

3.3 フロー情報の表現

データ・制御フローは構文要素と密接に関連しているため，File 要素中に flow 要素を埋め込んで表現する．データフローは制御フローをたどり，制御フロー上で変数定義の文とその変数を使用している文を求め，それらの間の関係を表現する．配列はいずれかの要素を定義あるいは使用している場合，配列全体が定義あるいは使用されたととらえる．同様に構造体・共用体変数もいずれかのメンバ変数が定義あるいは使用している場合，構造体・共用体変数が定義あるいは使用されたととらえる．

flow 要素は Stmt 要素の兄弟要素として表現し，全 Stmt 要素に対して情報を付与する．**図 3** は図 1 のソースコードにおけるフロー情報を表している．1 行目と 3 行目の flow 要素は図 1 の 3 行目の while 文に対するフロー情報を表している．1 行目の flow 要素は while 文の条件式が真の場合の，3 行目は偽の場合の制御フロー情報である．7 行目の flow 要素は式 “ans += tmp” が式 “tmp = x--” に依存していることを示している．

3.4 型情報の表現

型情報は構造体や列挙体など複雑な構造の型を File 要素中に埋め込んで表現すると，構文構造の表現と混在して複雑になるため，TypeInfos 要素として分けて表現した．各識別子には型の識別 ID を属性 type_id に付与し，TypeInfo 要素の型情報との対応関係を表現する．

TypeInfo 要素は 1 つの型を表す．sort 属性はこの型が構造体，共用体，列挙体，typedef 型，基本型，ポインタ型，配列型，関数型，可変引数のいずれかを示す．ref 属性は，この型が typedef 型である場合には本来の型，この型がポインタ型である場合にはそれが指す型，この型が配列型である場合には格納するデータの型，この型が関数型である場合には戻り値の型の ID を示す．この型が構造体，共用体，列挙体である場合はこの型のメンバ，この型が関数型である場合にはこの型の引数を typeRef 要素によって表現する．

図 4 は図 1 のソースコードにおける型情報とそれに関連する識別子の情報を表している．12 行目は関数 sum の返り値，引数 x，ローカル変数 ans と tmp の型 int に関する情報を表している．13–15 行目は関数 sum に関する型情報を表しており， 返り値の型情報を ref 属性によって，引数の型情報を子要素 typeRef によって示している

3.5 CX-model の利点

CX-model は多くの CASE ツールで必要になる，定義参

```
<File>
  <Function id="s33">...
    <ident defid="s33" type_id="s80">sum</ident>
    <op>(</op><Param id="s32">...
       <ident defid="s32" type_id="s81">x</ident>
    </Param><op>)</op>...
    <Local id="s36">...
      <ident defid="s34" type_id="s81">ans</ident>...
      <ident defid="s35" type_id="s81">tmp</ident>...
</File>
<TypeInfos>
    <TypeInfo id="s81" sort="standard" type="int" const="false" volatile="false"/>
    <TypeInfo id="s80" sort="function" ref="s81">
       <typeRef sort="argument" ref="s81"/>
    </TypeInfo>
</TypeInfos>
```

図 4 サンプルコードに対する CX-model の型情報表現（抜粋）
Fig. 4 A part of CX-model representation of the type information for the code.

照関係，データ・制御フロー情報，型情報を表現している．また，構造が複雑になりすぎないよう，構文構造と型情報を分割している．そのため，DOM，SAX，XPath，XSLT，XQuery などの XML 関連技術を用いて容易に必要な情報を取得することが可能である．

4. CX-Checker

4.1 概要

2 章で述べた問題点を解決するために，本研究では，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」を提案する．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．

CX-Checker の構成を図 5 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 [20] の CX-model [5], [17] に変換される．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 4.2 節で述べる．また，CX-Checker は CLI と Eclipse プラグインのインタフェースの 2 つを持つ．

Eclipse プラグインインタフェースの実行画面を図 6 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とその説明を一覧にして表示する．

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 である．

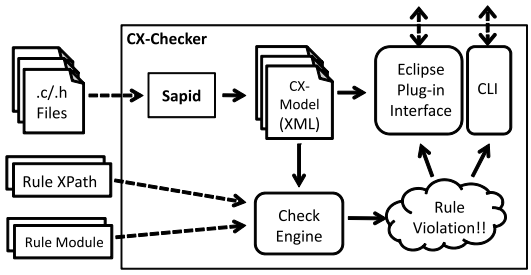

図 5 CX-Checker の概要
Fig. 5 The outline of CX-Checker.

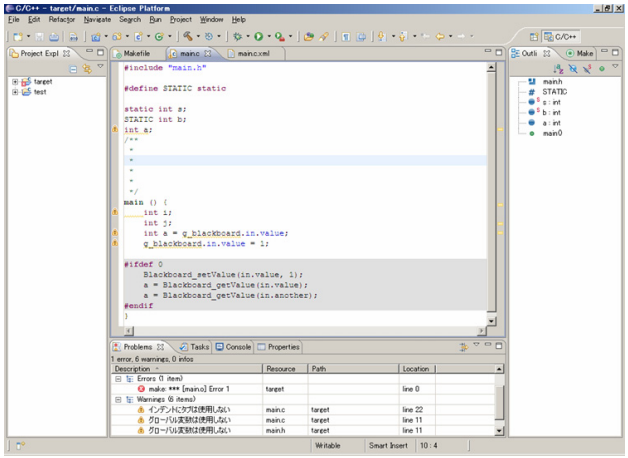

図 6 Eclipse プラグインインタフェースの実行画面
Fig. 6 A screenshot of our Eclipse plug-in interface.

4.2 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 とラッパを用いたルールは複雑なルールに向く．以下では例を用いながらそれぞれのルールの実装方法を説明する．

表 1 追加した拡張 XPath 関数
Table 1 Defined XPath extension functions.

名前	説明
get-type-info	引数の型の TypeInfo 要素を返す
get-solved-type-info	引数の型の TypeInfo 要素を返す．typedef は解決する
get-pointee-type-info	引数のポインタ型（typedef は解決する）がポイントする型の TypeInfo 要素を返す
get-type-info-by-tag-name	タグ名から TypeInfo 要素を探して返す
get-type-sort	引数の型の分類を “standard” などの文字列で返す
get-solved-type-sort	引数の型の分類を “standard” などの文字列で返す．typedef は解決する
get-pointee-type-sort	引数のポインタ型（typedef は解決する）がポイントする型の分類を “standard” などの文字列で返す
get-sign	引数の型の符号を “true” などの文字列で返す．typedef は解決する
get-type-length	引数の型のビット幅を返す．typedef は解決する
get-pointing-level	引数のポインタ型のポイント段階数を返す．typedef は解決する
is-void	引数が void 型か否かを判定する．typedef は解決する
is-floating	引数が浮動小数点型か否かを判定する．typedef は解決する
is-void-function	引数が void を返す関数型か否かを判定する．typedef は解決する
is-function-pointer	引数が関数ポインタ型か否かを判定する．typedef は解決する
is-signed	引数が signed か否かを判定する．typedef は解決する
is-unsigned	引数が unsigned か否かを判定する．typedef は解決する
can-be-floating	引数が浮動小数点型か否かを判定する．共用体のメンバを考慮する．typedef は解決する
may-be-boolean	引数が実質的ブール型か否かを判断する．typedef は解決する
equals	引数の型が等しいか否かを判定する
is-compatible	引数の型に互換性があるか否かを判定する
is-lossy	第 1 引数の型から第 2 引数の型へ情報の損失を伴う変換が起き得るか否かを判定する．typedef は解決する
is-assign	引数が代入式か否かを判定する
is-arithmetic	引数が算術演算式か否かを判定する
is-call	引数が関数呼び出し式か否かを判定する
is-bit	引数がビット演算式か否かを判定する
is-shift	引数がシフト式か否かを判定する
is-logical	引数が倫理演算式か否かを判定する
is-comparison	引数が比較演算式か否かを判定する
is-parenthetic	引数が括弧式か否かを判定する
is-explicit-cast	引数が明示的キャスト式か否かを判定する
is-implicit-cast	引数が暗黙的にキャストされる式か否かを判定する
matches	第 1 引数の文字列が第 2 引数の正規表現にマッチするか否かを判定する

4.2.1 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 の名のとおり，パスを表す言語を用いて XML の要素を取得する言語である．XPath の例を以下に示す．

```
/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，さらにその直下にある 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．

```
//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（“	” はタブ文字の実体参照）を含むものを取得している．

さらに，本研究では，型情報を利用したルールを記述する支援として型情報に関する拡張 XPath 関数を用意している．XPath では XML 木構造上の離れた場所の情報を参照することが難しいため，型に関する情報を容易に取得できるよう，**表 1** に示す関数を定義している．

拡張 XPath 関数とは，XPath 評価器に標準で用意されている XPath 関数のほかに，独自に定義した XPath 関数である．CX-Checker では，拡張 XPath 関数の追加に Java XPath API を使用している．

拡張 XPath 関数の設計にあたっては，現時点で取り扱った型情報に関する処理を分析し，特定のルールを記述するためではなく，型情報に関して必要となる汎用的な関数を用意している．

拡張 XPath 関数を用いることで，たとえば，「MISRA-C ルール 37：符号付き整数へのビット演算は禁止」は以下のように記述することが可能となる．

```
//Expr[cx:is-bit(.)] /Expr[not(cx:is-unsigned(.))]
```

なお，CX-Checker では，対象ソースコードの実行環境での型情報を設定項目としている．そのため，CX-Checker の実行環境と対象コードの実行環境とで型のサイズが異なっていても，上記の拡張 XPath 関数を用いて解析できる．

4.2.2 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．たとえば，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．

```
public class ARule implements CheckerClass {
  public List<Result>
    check(IFile file, CheckRule rule) {
      List<Result> results
        = new ArrayList<Result>();
      Document doc = file.getDOM();
      ...
      return results;
  }
}
```

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 1 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合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．

4.2.3 ラッパを用いたルール

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分かりにくいという問題点がある．たとえば，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とる．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，そうでない場合は分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インタフェースを提供する．ラッパ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．実装例を以下に示す．

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

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．

4.2.4 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，XPath によるルール開発には XPath および CX-model の知識が必要である．

XPath は W3C が定める XML に関する標準技術であるが，ルールの記述を行ううえでは，意図するプログラムコードに関する特徴を，CX-model の構造で表現する必要

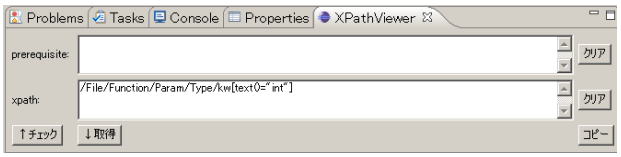

図 7 XPath ルール補助インタフェース
Fig. 7 A screenshot of our XPath rule assistant interface.

がある．

そこで CX-Checker では，CX-model に習熟していない開発者の XPath ルールを定義しやすいように，ソースコードと CX-model の間を埋めるための 補助インタフェースを提供している．**図 7** にインタフェースを示す．

prerequisite 入力ボックスには，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．前提条件を分離して記述することで，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 を記述することができ，複雑な構造特徴を指定することを支援する．補助インタフェースは CX-model の DTD 情報および，検査対象の XML 文書中の出現状況を分析し，候補となりうる CX-model 要素を補完する機能 [21] も有している．

「チェック」ボタンは入力された XPath を評価し，XPath 式が正しいかどうか，正しい場合は，該当する XML ノードを検索し，ソースコード中の該当部分に波線を表示する．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．

さらに，「取得」ボタンでは，「チェック」ボタンとは逆に，指定したソースコードから XPath 式を作成することができる．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，その部分のみを取得する XPath を自動生成する．この機能を利用し取得した XPath を参考とすることで，コーディングルールとして表現したい一般的な構造を記述することが容易となる．

5. 記述性の評価

本研究では，CX-Checker のカスタマイズ能力が実用的なレベルであること，ルール開発が容易であることを評価した．以下でそれぞれを詳しく述べる．

5.1 MISRA-C の実装

組込みソフトウェアで利用されることが多い 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 2** に示す．

この結果から，CX-Checker が，代表的な商用ツールである SQMlint や QAC と同程度の対応が可能なインタフェースを持つことが分かる．また，実装可能なルールのうち，約 5 割が XPath による記述で実現できることが分かった．CX-model による構造文書化と，拡張 XPath 関数の提供により，実用的なルールの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 2 主要なコーディングチェッカの MISRA-C 対応ルール数
Table 2 Number of supporting MISRA-C rules of major coding checkers.

	QAC	SQMlint	CX-Checker
対応可能	118	86	**102**
（XPath で記述可能）	-	-	（**61**）
対応不可能	9	41	**25**
合計	127	127	127
割合（%）	93	68	**80**

表 3 XPath ルールの実装の所要時間
Table 3 The amount of time required to implement XPath rules.

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

上記で実現可能と判定したルールのうち，拡張 XPath 関数を用いずに XPath で実現可能であると判断したルールのうち，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 3 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分かる．

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 では兄弟ノードどうし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．

5.2 アイシン精機株式会社の標準ルールの実装

組織内の独自ルールの例として，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．実際に企業で運用され

表 4 CX-Checker が対応できないルールの理由と個数
Table 4 Number of rules which CX-Checker cannot support and its reasons.

理由	個数	詳細
ルールの再定義が必要なもの	6	ルールの詳細が不明確で定義ができない（#3，#4，#46，#114）
		プログラム解析だけでは判定不能（#1，#99）
CX-model 表現能力に起因するもの	5	前処理後の情報が必要（#90）
		ファイルをまたがった解析が必要（#25，#27，#106，#109）
Sapid の解析能力に起因するもの	5	前処理記述に関する情報が必要（#94，#97，#100）
		ポインタ解析が必要（#103）
		検出対象が構文違反である（#9）

ているルールのうち，市販ツールが標準では対応していないルールが存在し，そのうちの多くに対応できている点は，カスタマイズ性が有効であったことを示していると考える．また，XML に関する標準技術である XPath を用いて 54%が記述できた点も，開発現場において柔軟にルールを定義できることを示しており，CX-model を用いた提案手法のアプローチの優位性を示すことができたと考える．

5.3 考察

CX-Checker のカスタマイズ性能を MISRA-C のルールおよび，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 80%，アイシン精機株式会社のルールで 76% のルールに対応可能であることが分かり 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，拡張 XPath 関数を用いずに 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分かった．XPath の記述性は十分に高いものであるといえる．

MISRA-C のルールうち，型情報を用いるものが 22 ルールあった．そのうち 21 ルールは，拡張 XPath 関数を用いて XPath で記述可能であり，提案手法での拡張 XPath 関数が有用であったことが確認できた．一方でフロー情報を利用したルールは，「到達不能なコードが存在しない」などの 2 つのみであった．今後，フローの情報を利用する高度なルールを実装するなどしてフロー情報の有用性を議論する必要がある．

QAC が対応しているが，現状の CX-Checker では対応できない 16 のルールの内訳を**表 4** に示す．

対応できなかったルールの原因の多くとして以下の 2 つがあげられる．

- マクロの本体の解析を行うルールであった．
- 複数ファイルにまたがる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，一概に弱点とはいえない．

複数ファイルにまたがる情報を用いるルールとは，2 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 1 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．

6. おわりに

6.1 まとめ

本研究では，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 を提案した．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，また，MISRA-C とアイシン精機の独自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分かった．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

マイズ性を持つと結論付ける．

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る．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．

6.2 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．たとえば，プロトタイプ宣言をヘッダ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．現在，XREF-model [5] として，複数ファイルにまたがる情報を整理できる記法およびツールを整備したため，今後この情報を用いてルールを記述することを検討している．

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いえる．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．たとえば，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 本ツールの開発の一部は，アイシン精機株式会社の協力を得て，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 Learning）のテーマとして実施された．上原伸介氏，蛭牟田英治氏，林英志氏，日高隆博氏をはじめとする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（19700023，20300009，21700027）の助成による．

参考文献

[1] Goodliffe, P.: 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, available from ⟨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] 大須賀俊憲，小林隆志，間瀬順一，渥美紀寿，山本晋一郎，鈴村延保，阿草清滋：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，ソフトウェアエンジニアリング最前線 2009，pp.119–126 (2009).
[5] Atsumi, N., Yamamoto, S., Kobayashi, T. and Agusa, K.: An XML C source code interchange format for CASE tools, *Proc. IEEE Computer Software and Applications Conference* (*COMPSAC2011*), pp.498–503 (2011).
[6] Kernighan, B.W. and Ritchie, D.M.: *C programming language*, 2nd edition, Prentice Hall (1988).
[7] RainCode Checker, available from ⟨http://www.raincode.com/checker.html⟩.
[8] Checkstyle, available from ⟨http://checkstyle.sourceforge.net/⟩.
[9] 堀江佑太，福原和哉，今井信太郎，新井義和，猪股俊光：XML 形式を用いた C 言語用静的解析ツールの開発，第 73 回情報処理学会全国大会予稿集 5L-4 (2011).
[10] 福原和哉，猪股俊光，新井義和，曽我正和：コードパターンの検出に適した C 言語前処理系解析器の開発，情報処理学会研究報告，Vol.2010-EMB-17, No.10 (2010).
[11] Aguiar, A., David, G. and Badros, G.J.: JavaML 2.0: Enriching the markup language for Java source code, *Proc. XML: Aplicações e Tecnologias Associadas* (*XATA2004*) (2004).
[12] 吉田 一，山本晋一郎，阿草清滋：XML を用いた汎用的な細粒度ソフトウェアリポジトリの実装，情報処理学会論文誌，Vol.44, No.6, pp.1509–1516 (2003).
[13] Maruyama, K. and Yamamoto, S.: A CASE tool platform using an XML representation of Java source code, *Proc. IEEE International Workshop on Source Code Analysis and Manipulation* (*SCAM'04*), pp.158–167 (2004).
[14] Kawashima, H. and Gondow, K.: Experience with ANSI C markup language for a cross-referencer, *Proc. 36th Annual Hawaii International Conference on System Sciences* (*HICSS'03*), Track 9 (10pages) (2003).
[15] Collard, M.L.: Addressing source code using srcML, *Proc. IEEE International Workshop on Program Comprehension* (*IWPC'05*) (2005).
[16] Brad King. GCC-XML (2011), available from

⟨http://www.gccxml.org/⟩.
[17] 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，情報処理学会研究報告，Vol.2005-SE-149, No.8 (2005).
[18] Badros, G.J.: JavaML: A markup language for java source code, *Proc. International World Wide Web Conference* (*WWW2000*), pp.159–177 (2000).
[19] Document type definition of CX-model, available from ⟨http://www.sapid.org/DTD/cx-model.dtd⟩.
[20] 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 (1998).
[21] 林　英志，日高隆博，山本晋一郎，小林隆志，上原正太，間瀬順一，鈴村延保，阿草清滋：メタ情報とコンテキスト情報を用いた入力補完機能と XPath 入力への応用，情報処理学会研究報告，Vol.2010-SE-167, No.28 (2010).

大須賀 俊憲 （正会員）

2008 年名古屋大学大学院情報科学研究科博士前期課程修了．同年日本電気株式会社入社．2011 年株式会社ネットレックス入社．現在に至る．修士（情報科学）．平成 22 年度山下記念研究賞受賞．

小林 隆志 （正会員）

2004 年東京工業大学大学院情報理工学研究科博士課程修了．2002 年同大学助手．2007 年名古屋大学大学院情報科学研究科特任助教授．2009 年より准教授．博士（工学）．ソフトウェア設計方法論，ソフトウェア理解等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，日本データベース学会，ACM，IEEE 各会員．

渥美 紀寿 （正会員）

2007 年名古屋大学大学院工学研究科博士課程後期課程修了．2005 年南山大学数理情報学部情報通信学科講師．2010 年名古屋大学大学院情報科学研究科研究員．2011 年より特任助教．博士（工学）．プログラム解析，ソフトウェア保守，ソフトウェア再利用等の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

間瀬 順一

1997 年名古屋大学大学院理学研究科博士課程後期課程単位取得中退．2002 年アイシン精機株式会社入社．2007 年アイシン・コムクルーズ株式会社に出向．現在同社技術 1 部グループマネージャー．技術士（情報工学）．自動車搭載用のコンピュータ開発に従事．

山本 晋一郎 （正会員）

1991 年名古屋大学大学院博士課程修了，同大学助手．1996 年同大学講師．1998 年愛知県立大学情報科学部助教授．2007 年同大学准教授．2010 年より同大学教授．博士（工学）．プログラミング言語処理系，ソフトウェア開発に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会各会員．

鈴村 延保

1977 年広島大学工学部電子工学科卒業．1977 年アイシン精機（株）入社．現在，同社ソフトウェアセンター主査．自動変速機やアクティブ・サスペンション，ABS 等のコンピュータ，ソフトウェア，IC の開発・製品化に従事．自動車技術会/機能安全 ISO26262 規格審議委員．

阿草 清滋 （正会員）

1972 年京都大学大学院工学研究科修士課程修了．同博士課程へ進学．1974 年より同情報工学助手．同講師，助教授を経て，1989 年より名古屋大学教授．工学博士．ソフトウェア開発方法論，知的開発環境，仕様化技法，再利用技法の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．