Vol.2015-SE-187 No.2
2015/3/12

コーディング規約検査器のための規約記述DSL

伊藤 達也[1,a)]　大久保 弘崇[2,b)]　粕谷　英人[2,c)]　山本 晋一郎[2,d)]

概要：検査ルールを使用者が追加できる C 言語向けコーディング検査器 CX-Checker を対象とするルール記述用 DSL を提案し，その実装を示す．提案した DSL は，従来の検査ルール記述言語で要求される CX-Checker の内部構造に関する知識を必要としない．MISRA-C 2012 Guidelines のルールを用いることで，提案した DSL の記述性を検証した．結果，記述できるルール集合は従来の記述言語と等しく，記述性が制限されていないことを確認した．また，記述するコードの量に関して大きく改善されていることを実証した．記述したルールを用いてルール検査を実行し，メモリ消費量や実行時間は若干のオーバーヘッドを含むが同等であることを確認した．

1. はじめに

ソフトウェアが高機能になるにつれて，ソースコードの量は増加する．例えば，Black Duck Software の調査[*1]によると，Web ブラウザの Firefox は約 1,200 万行，MySQL は約 230 万行である．これらのソフトウェアは現在も開発が進んでおり，機能の追加に伴いソースコード量はさらに増加することが予想される．

また，ソフトウェアの品質を維持するために，ソースコードを適切に管理する必要があるが人手による管理は限界がある．理由として，ソースコード量の増加に伴う管理者の負担増大，管理者ごとに判断基準に差があり，管理基準が一意でないなどがあげられる．この問題を解決するために品質管理の自動化が有効である．

ソースコードの品質管理の項目には，コードの可読性や保守・テストの容易性，コンパイル時の警告数などがある．これらの項目を制御する方策として，ソースコードに対する明確な規則やガイドラインであるコーディング規約がある．コーディング規約には，プロジェクト単位や企業で独自に設定されるものから，組み込みシステム向けの MISRA-C 2012 Guidelines[1] や一般的なソフトウェアのセキュリティ確保を目的とした CERT C Secure Coding Standard[2] のように公的な機関で作成されたものまで様々に存在する．

コーディング規約自動検査器の 1 つに C 言語プログラムを対象とした CX-Checker [4] がある．CX-Checker は XPath や Java を用いて検査ルールを独自に追加できるが，現在のルール記述言語には記述能力が低い，コード記述量が多いなどの問題点があり，ルール記述における負担が大きい．

ルール記述の容易化を目的として，ルール記述用 DSL を提案し実装する．既存のルール記述言語によるルール記述を行う上で，内部データの確認と理解が特に負担が大きい．このステップを省略し簡略化することで，ルール記述にかかる負担を大きく軽減できると考えられる．また，既存のルール記述言語の短所を補うことで，ルール記述の生産性の向上を図る．

2. 背景技術

以降，MISRA-C 2012 Guidelines のルールを『<本文>』(MISRA-C XX-X) と表記する．<本文>はルールの内容で，XX-X は規約番号である．

2.1 コーディング規約

コーディング規約とは，ソースコード記述時に従うべき規則やガイドラインである．『goto 文を使用してはならない』(MISRA-C 15-1) といった単純な規約や，『論理演算子 “&&” と “||” の右オペランドは持続的な副作用を含んではならない』(MISRA-C 13-5) といった複雑な規約，『エラー処理には一貫性のある方針を採用する』(CERT C Secure

[1] 愛知県立大学大学院 情報科学研究科
Graduate School of Infomation Science and Technology, Aichi Prefectural University
[2] 愛知県立大学 情報科学部
School of Infomation Science and Technology, Aichi Prefectural University
a) ito@yamamoto.ist.aichi-pu.ac.jp
b) ohkubo@ist.aichi-pu.ac.jp
c) kasuya@ist.aichi-pu.ac.jp
d) yamamoto@ist.aichi-pu.ac.jp
[*1] https://www.openhub.net/

Coding Standard ERR00-C) のような自動判定できない規約まで，その内容は様々である．

公的な規約文書の例を挙げる．

MISRA-C 2012 Guidelines [1]

MISRA (Motor Industry Software Reliability Association) の定めるソフトウェア設計標準規格．組み込みシステムの安全性と移植性を確保する目的で制定されている．全 159 のガイドラインには自動検査できるルール，適用優先度などの項目が設定されており，119 ガイドラインが自動検査できるとされている．

CERT C Secure Coding Standard [2]

CERT/CC (Computer Emergency Response Team/ Coordination Center) の定めるコーディング規約．ソフトウェアのセキュリティを確保するために必要なルール群を定めている．全 290 ルール．規約には深刻度・脆弱性につながる可能性・修正コストの 3 項目の評価に基づいて 3 段階の優先度が設定されている．利用者は優先度を元に，適用する規約の範囲を決定できる．

2.2 規約検査器

規約検査器とは，ソースコードがコーディング規約に従っているかどうかを検査するためのツールである．例として QA・C や CX-Checker が存在する．

QA・C

QA・C [3] とは，Programming Research が開発した商用の C 言語用静的解析ツールである．MISRA-C 2004 Guidelines に対応しており，複数のメトリクスを用いてソースコードを定量的に評価できるといった特徴がある．

CX-Checker

CX-Checker [4] とは，愛知県立大学と名古屋大学が共同で開発している C 言語プログラムを対象とした規約検査器である．使用者がチェックするルールを追加することができ，社内独自規約やプロジェクト固有のコーディングルールなど，既存の規約検査器ではチェックすることができないルールも検査できるのが特徴である．

C 言語プログラムの解析には Sapid を利用し，出力される CX-Model と呼ばれる XML 形式の解析結果を使用して検査を行う．検査に使用するルールについては 2.3 節で説明する．

2.2.1 Sapid

Sapid (Sophisticated APIs for CASE tool Development) [5] とは，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．C 言語のプログラムを解析して，モジュールよりも細かい構成要素である関数や変数といった情報をリポジトリに格納する．また，リポジトリを使用する CASE ツール作成のための API を提供する．

図 1 C プログラム断片

```
1  #include <stdio.h>
2  #include <signal.h>
3
4  int main() {
5  ...
```

図 2 CX-Model の例

```
1  <?xml version="1.0"?>
2  <!DOCTYPE CX SYSTEM "CX-model.dtd">
3  <CX>
4    <File name="test2.c">
5    <Include>
6      <kw>#include</kw><sp> </sp>
7     <hfile>&lt;stdio.h&gt;</hfile>
8    </Include>
9    <Include>
10     <kw>#include</kw><sp> </sp>
11    <hfile>&lt;signal.h&gt;</hfile>
12   </Include>
13   <Function>
14     <Type><kw>int</kw></Type>
15     <sp> </sp>
16     <ident defid="s33555257"
17       type_id="s83886409">main</ident>
18     <op>(</op><op>)</op>
19     <sp> </sp>
20     <Stmt sort="Block" id="s50331649">
21       <op>{</op>
22 ...
23 <TypeInfo id="s83886409" text="int()"
       sort="function" ref="s83886410"/>
24 ...
```

CX-Model

CX-Model とは，C 言語の抽象構文木を XML で表現したモデルである．CX-Model は 20 種類の非終端要素と 16 種類の終端要素から構成されている．図 1 の C プログラムを解析して生成される CX-Model が図 2 である．なお図 2 は可読化のための省略と整形を行っている．

定義箇所の識別子には一意に決まる ID が付与され，参照箇所の識別子にも同一 ID が付与される．図 2 では 19, 20 行目の ident タグが識別子を表す．一意に決まる ID は `defid` 属性として付与される．さらに，この識別子には型情報を示す型 ID も付与されており，型 ID を用いることで型情報を参照できる．図 2 では ident タグの `type_id` 属性として付与されており，27 行目は対応する型情報である．また，コメントやスペースなどのプログラムの振る舞いに

図 3 完全な構文による hfile タグの指定

```
1 /CX/File/Include/hfile
```

図 4 省略構文による hfile タグの指定

```
1 //Include/hfile
```

直接関係のない要素もモデルに含まれているため，XML タグ部分を除去すると，元のソースコードに戻る．

型情報

型情報とは，識別子が持つ型に関する情報をまとめた要素である．この情報を利用することで, 型や型の参照に関するルールが記述しやすくなる．CX-Model では，TypeInfo タグで表され，属性として型情報を保持する．

2.3 CX-Checker におけるルールの記述

公的な規約文書や社内独自規約などのコーディング規約は人間が読むための文書であるため，このままでは検査を行うことができない．検査できるようにするためには CX-Checker 用のルールファイルを記述する必要がある．

現在 CX-Checker 用のルール記述には，XPath と Java の 2 種類の言語が利用できる．

2.3.1 XPath を用いたルール記述

XPath [6] は W3C(World Wide Web Consortium) により開発された，XML 文書内の位置を指定するための言語である．特徴として，対象の位置を簡潔に指定可能な点が挙げられる．CX-Checker は内部データとして CX-Model を利用しており，ルールを書く上で有効である．

XPath によるルール記述の例を，『標準ヘッダファイル <signal.h> を使用してはならない』(MISRA-C 21-5) により示す．

XPath は XML をルートノードを頂点とする木構造として扱う．ノードの指定方法には，ルートノードから目的のノードまで辿る方法と，省略構文を使う方法の 2 つがある．前者の方法で CX-Model 中の#include プリプロセッサ指令のファイル名記述を指定する記述は図 3 のようになる．CX タグは CX-Model のルートノードであり，File タグは翻訳単位を，Include タグは#include 指令を表す．最後に include するファイル名が hfile タグで囲まれる．

次に省略構文を使用する．省略構文とは，ノードの共通部分を省略することができる．例えば，図 2 では，Include タグはトップレベルにのみに出現するので，途中経路を考慮する必要がない．よって，`/CX/File` 部分を省略することができる．省略構文を用いると図 4 の記述で include するファイル名記述を指定できる．

次に，指定した hfile タグのテキストを確認する．タグのテキストは `text()` で指定し，対象テキストと完全一致するかを確認するには，`text() =` 対象テキストと記述する．

図 5 MISRA-C 21-5 規則の XPath 記述

```
1  <rules>
2    <oneRule>
3      <level> 1 </level>
4      <content> MISRA-C 21-5 </content>
5      <xpath> //Include/hfile[text() =
6        "&lt;signal.h&gt;"] </xpath>
7      <condition> prohibit </condition>
8    </oneRule>
9  </rules>
```

図 6 MISRA-C 7-1 規則の XPath 記述

```
1  <rules>
2    <oneRule>
3      <level>5</level>
4      <content>MISRA-C 7-1</content>
5      <xpath>//macroBody[
6        (starts-with( text(), '0')) and
7        string-length(text()) > 1 ] |
8       //literal[(starts-with(text(),'0'))
9        and string-length(text()) > 1]
10     </xpath>
11     <condition>prohibit</condition>
12   </oneRule>
13 </rules>
```

XPath による記述だけでは，プログラム中に <signal.h> が存在するべきか否かの判断ができない．そこで，ルールを記述する際に XPath で指定した要素がどうあるべきかを指定できる．最終的に図 6 のようになる．

XPath によるルールは XML 形式で記述する．level タグはルールの重要度を 1-5 の範囲で指定し，content タグにはルールの説明を記述する．xpath タグ内に XPath を記述し，condition タグで XPath で指定した要素がどうあるべきかを指定する．condition タグの要素が require なら XPath で指定した要素が存在するべきで，prohibit なら XPath で指定した要素は存在してはならない．

以上のように XPath では検査対象を簡潔に指定できるという利点があるが，位置指定以外に使用できる関数が乏しいという問題がある．例えば，指定したノードの数やテキストの長さ，簡単な演算は行うことができる．例として，『8 進数を使用してはならない』(MISRA-C 7-1) がある．このルールは，文字列の先頭を比較する関数 `starts-with()` と文字列の長さを取得する関数 `string-length()` を使用することで，記述することができる．記述例を図**??**に示す．

プログラム内で 8 進数が登場する可能性がある箇所はマクロの本文とリテラル．その位置に先頭が 0 で長さ 2 以上の数値リテラルが存在しないかを検査している．

表 1　ルール記述言語まとめ

	XPath	Java
ルール記述のしやすさ	○	×
ルール記述に利用できる機能数	少	多
ルールあたりの記述量	20 行前後	100 行以上
記述できるルールの範囲	狭い	広い
ルール記述に必要な知識	XML, CX-Model の構造	

しかし，複数のノードを比べる関数や正規表現によるパターンマッチなどの複雑な処理を行うことができない．例として，『プロジェクト内に未使用な型宣言を含むべきではない』(MISRA-C 2-3) があげられる．このルールを記述するには typedef ノードが変数宣言や構造体宣言のノードの集合に 1 度以上出現するかを確認する必要がある．よって，記述できるルールに制限がある．

2.3.2　Java を用いたルール記述

CX-Checker のルール記述言語には Java が選択できる．Java では DOM を用いて CX-Model を走査する．CX-Model に含まれる情報からチェック可能なルールは全て記述可能である．例えば，MISRA-C 2012 Guidelines において文書内で機械的に検査可能とされている 119 ガイドラインのうち，81 個が記述できる．

残りの 38 個は，CX-Model で複数のファイルにまたがって検査したり，複雑な文章構造を判定できないので，記述できない．

しかし，ルールを記述するためには Java における DOM 操作や CX-Model の構造についての知識を必要とし，1 ルールあたりのコード量が多いといった問題がある．簡単なルールでも 100 行以上記述する必要がある．原因として以下のことを挙げることができる．

- ルール記述に使用するライブラリや定義する必要があるメソッドが多い
- 検査対象ノードを選択するために DOM 操作をする必要があるので，ループ文やループ文の入れ子が多く出現する．
- ノードの選択や処理を行う際に複数の条件式を組み合わせる必要がある

2.3.3　問題点

2 つの言語の特徴を表 1 にまとめる．XPath は簡単にルールを記述することができるが，記述できるルール数が少ない．Java は多くのルールを記述することができるが，ルールを記述するために多くのコードを記述する必要がある．また，どちらの言語もルールを記述する上で CX-Model の構造を理解する必要がある．

3.　ルール記述用 DSL

既存のルール記述言語の短所を補えるような，CX-Checker ルール記述用 DSL の提案し実装する．実装方法は CX-Checker と連携のしやすく既存のライブラリを活用できる内部 DSL を選択した．ベース言語は Java との連携が容易で，DSL の実装を行う上で有用な機能を持つ Scala を選択した．

3.1　設計方針

DSL を設計する上で達成するべき目標を以下に示す．

- 内部データ構造である CX-Model の知識を利用者に要求しない
- 複数の機能を組み合わせて全体の処理を表現できるようにする

1 つ目は，DSL の機能として提供するメソッド名を C 言語の知識から類推できる機能名とする．2 つ目に，内部の処理をメソッドチェインで表現できるようにする．これは，利用者が最終的な出力にのみ集中できるようにするためである．そのために，機能の入出力形式の型を統一し，各機能の入力は明白なので省略できるようにした．

3.2　機能

提案 DSL では検査対象のプログラムを**要素**という単位でとらえ，要素を扱うために提案 DSL が提供するメソッドを**機能**と呼ぶ．

提案 DSL の記述は全て `Target` から始まり，`Target` の次に DSL の動作を決定する機能を指定する動作を決定する機能は 0 個以上の引数を持ち，機能によって引数の数は異なる．DSL の動作内容は，検査対象の CX-Model ファイルの指定，要素の選択，型情報の選択，結果の選択，結果の表示 5 つに分類できる．検査対象の指定には`<+`を指定する．引数として検査対象のファイル名を 1 つ取る．要素の選択と型情報の選択には`:=`を指定する．引数として要素を選択する場合は機能を，型情報を選択する場合は検索機能を 1 つ以上取る．複数の機能，もしくは検索機能を組み合わせる場合は機能結合子を使用する．機能の場合は `in` を，検索機能の場合は `and` を用いて組み合わせる．結果を選択するには `getTarget` を指定する．引数はとらず，要素の集合を返す．結果の表示には `checkLocation` や `defaultShow`, `simpleShow` を指定する．引数はとらず，結果を標準出力に出力する．`checkLocation` を指定した場合，選択した要素の C プログラム上の位置を出力する．`defaultShow` を指定した場合，選択した要素の CX-Model の表現をそのまま出力する．`simpleShow` を指定した場合，選択した要素の CX-Model の表現を一部省略して出力する．

これらの動作を組み合わせてルールを記述する．

提案 DSL によるルール記述の手順を示す．初めに，検査対象の CX-Model を指定する．次に，検査に必要な要素や型情報を選択し，選択した要素に対して処理を行う．最後に，検査結果を表示する．記述するルールの複雑さによって処理を行わなかったり，複数の要素に対して処理を行う場合もある．

図 7 インクルードしたヘッダファイル名

```
1 Target := name in include
```

図 8 可変長配列の型情報か判別する条件式

```
1 (currentNode.getNodeName()
2   .equals("TypeInfo"))
3   && (((Element) currentNode)
4     .getAttribute("sort")
5     .equals("array"))
6   && !(((Element) currentNode)
7     .getAttribute("array_size")
8     .matches("^[0-9]+"))
```

提案 DSL の機能は大きく分けて 3 種類ある．

検査対象の選択

ルール検査に必要な要素を選択する．機能名を C 言語の知識から連想できる名称とし，CX-Model を意識することなく要素を選択することができる．

インクルードするヘッダファイル名を選択する操作を例に説明する．これを XPath では図 4 のように記述した．Java で DOM を用いる場合も，同様に CX-Model の構造に関する知識が必要となる．提案した DSL では，インクルード文を選択する機能である include と，名前要素を選択する機能 name を，包含を表す結合子 in で結合し，図 7 のような記述で XPath と同様の簡潔さでこれが記述できる．

型情報の選択・加工

CX-Model において，型の情報はプログラム本体のマークアップとは別の場所にまとめられている．プログラム中に出現する全ての型の情報がリストアップされ，ユニークな ID が割り振られている．プログラム中の型を持つ要素はその型をこの ID を用いて表す．提案した DSL でも，型の情報はプログラム本体とは別に扱う形態を継承した．

可変長配列の型情報を選択する操作を例に説明する．可変長配列とは，`int x[d]` のような配列のサイズ指定に変数を使用している配列を指す．CX-Model の型情報では，要素の型の種類が配列型で，また配列長が数値リテラルでない要素がこれに対応する．

Java で DOM を用いる場合，型情報を保持する XML 要素は TypeInfo タグを持つ．この要素は XML の末尾に集められているので，その先頭からイテレータで走査することで，目的の型情報を発見する．現在のノード currentNode が可変型配列の型情報か判別する条件式は図 8 のようになる．

相当する例を提案した DSL で記述したものを図 9 に示す．指定した型の型情報を選択する検索機能の tiSort と，配列長の記述が与えられた正規表現にマッチする要素を選択する検索機能の pTiArraySize を and 結合子で結合することで実現している．

図 9 DSL による可変長配列の検索

```
1 Target := tiSort("array") and
2   pTiArraySize("^[0-9]+")
```

この検索機能を実現するため，提案 DSL の実装では型情報を独自のデータ構造に変換し，これに対する検索用機能を型情報が持つ項目に対して網羅的に用意した．

`tiSort` は指定した型の型情報を選択する検索機能で，`pTiArraySize` は配列の長さが正規表現にマッチする要素を選択する検索機能である．この 2 つを `and` 結合子によって結合し，配列型の配列の長さが数字以外の要素を選択する．

選択した検査対象への処理

選択した要素に対して処理を行う．提案 DSL には，選択した要素内に指定したテキストがしきい値以上存在するかや，正規表現によるパターンマッチを行う機能を備える．

処理内容は記述するルールによって変化するので，全ての処理機能を用意することはできない．提案 DSL では，利用者が独自に定義した機能を**拡張機能**として組み込むことができる．機能の記述は提案 DSL のベース言語である Scala で行う．拡張機能の記述を行うには，Scala や CX-Model の構造に関する知識が必要となる．

『プロジェクトは使用されていない型宣言を含むべきでない』(MISRA-C 2-3) を例に説明する．選択するためには型宣言と変数宣言を比較する必要があるが，提案 DSL には 2 つの要素を比較する機能は用意されていない．よって，2 つの要素を比較する機能を追加する．提案 DSL では図 10 のように記述する．

`checkCustomFunction()` 機能 で拡張機能を指定することで，DSL と組み合わせて使用することができる．拡張機能である `diffAttr()` 機能は 7–13 行目に記述されており，Scala の文法に従って記述する．拡張機能を記述するには Scala の XML 操作に関する知識が必要である．拡張機能 `diffAttr()` を使用して型宣言と変数宣言を比較することで，目的の要素を選択する．

ルール記述例

提案した DSL によるルール記述の例を示す．

基本機能のみ

DSL の最も単純な使用方法である．

『goto 文を選択する』(MISRA-C 15-1)，を例として使用方法を説明する．図 11 のように記述する．

初めに，必要なライブラリをインポートする (1, 2 行目)．`dsl._` は制御用ライブラリで，`dsl.DSLDefinition._` は基本機能ライブラリであり，いずれも提案 DSL を使用する上の必須ライブラリである．次に，検査対象の CX-Model を指定する (4 行目)．今回は`"test.c.xml"`を検査対象ファ

図 10 使用されていない型宣言の選択

```
1  Target := typedef
2  val tdNode = Target getTarget
3  Target := checkCustomFunction(
4    diffAttr("id", "type_id", tdNode)) in
       name in cType in variable
5  Target checkLocation
6
7  def diffAttr(attr1 : String, attr2 :
       String, nSeq1 : NodeSeq) = { nSeq2 :
        NodeSeq ⇒
8    val filtered =
9      for (n1 ← nSeq1) yield {
10       if (nSeq2.find(n2 ⇒ (n2 \@ attr2)
             == n1) isEmpty)  Some(n1) else
              None
11     }
12   NodeSeq.fromSeq(filtered.flatten)
13 }
```

図 11 goto 文を選択

```
1  import dsl._
2  import dsl.DSLDefinition._
3
4  Target <+ "test.c.xml"
5  Target := content("goto") in kw
6  Target checkLocation
```

イルとする．次に，検査に必要な要素を選択する．(6 行目) goto は C 言語におけるキーワードである．よって，キーワード要素を選択する機能と，テキストが"goto"な要素を選択する機能を組み合わせる．Target := の後ろに機能を指定する．複数の機能を組み合わせる場合は，in 結合子を使用する．この時 in 結合子の右側に，選択する要素の範囲が広い機能を置く．例えば，ローカル変数の型の名前を選択したい場合，選択する要素の範囲は，ローカル変数 ⊃ 型 ⊃ 型の名前，となるので name in cType in localVar のように記述する．最後に，選択した結果を表示する (8 行目)．checkLocation 機能は選択した要素の，ソースコード上の位置を表示する機能である．

型情報のみ

型情報のみを使用した検査方法である．

『可変長配列』(MISRA-C 18-8) を例として，使用法を説明する．図 12 のように記述する．

基本的な使用方法は基本機能を使用した場合と同じなので，違いのみ説明する．初めに，型情報用のライブラリ dsl.CXTypeInfos._をインポートする必要がある．次に，複数の機能を組み合わせる場合は in ではなく and 結合子

図 12 可変長配列を選択

```
1  import dsl._
2  import dsl.DSLDefinition._
3  import dsl.CXTypeInfos._
4
5  Target <+ "test.c.xml"
6  Target := tiSort("array") and
7    pTiArraySize("^[0−9]+")
8  Target checkLocation
```

を使用する．この時，機能の順序に制約はない．

4. 評価

CX-Checker の既存のルール記述言語と比較して，提案する DSL を評価する．

4.1 記述できるルール数の比較

CX-Checker のルール記述に使用できる XPath や Java，提案した DSL を用いて MISRA-C 2012 Guidelines のチェックルールを記述することにより，それぞれ言語の記述性を比較する．

確認を行うガイドライン数とその理由を以下に示す．

総ガイドライン 143 ガイドライン

規約文書で自動検査可能としているガイドライン 116 ガイドライン

CX-Checker 未対応ガイドライン 35 ガイドライン

ルール記述検討ガイドライン 81 ガイドライン

81 ガイドラインに対して，XPath, Java, 提案 DSL によってルール記述可能かどうかについて検討した．

結果，XPath で 25 ルール，Java で 68 ルール，提案 DSL で 68 ルールとなった．提案 DSL で記述可能な 68 ルールのうち，提案 DSL で用意した機能のみで記述可能なルールが 37 ルール，拡張機能を用意する必要があるルールが 31 ルールである．

提案 DSL 単体で記述可能なルールは Java の約半分だが，拡張機能を用意することで同等のルール数を記述できる．よって，提案 DSL の記述力は Java と同等であるといえる．

4.2 ルール記述に使用する機能数の比較

各言語の機能性を比較した結果を表 2 に示す．ここから，提案 DSL は XPath が対応していない正規表現や型情報を扱う機能を備え，Java と比べて簡潔に記述できると言える．また，提案 DSL では拡張機能を使用しない場合，CX-Model に関する知識を必要としない．以上より，提案 DSL は既存のルール記述言語の短所を補うことができたといえる．

表 2　各言語の機能性の比較

	Xpath	Java	提案 DSL
簡潔さ	○	×	○
ノード選択	○	○	○
語句の指定	○	○	○
否定	○	○	○
正規表現	×	○	○
型情報	×	○	○
複数ノードの比較	×	○	△
機能の拡張性	△	○	△
CX-Model の知識	×	×	△
1 ルールの記述量	20 行前後	100 行以上	2–20 行

4.3　ルール検査におけるメモリ使用量と実行時間の比較

提案 DSL の実行性能の評価の為にルール検査時のメモリ使用量と実行時間の比較を行う．

条件を以下に示す．

検査対象　bison-1.25, dhrystone-2.1, gawk-3.0.1, gnugo-1.2, gzip-1.2.4, patch-2.5 のソースプログラム総計 91 ファイル

ファイルあたりのコード行数　最小：6 行 / 最大：5612 行

総行数　50517 行

検査ルールは以下のルールを用意した．

単純なルール　『goto 式を使用してはならない』(MISRA-C 15-1)

複雑なルール　『プロジェクト内に未使用な型宣言を含むべきではない』(MISRA-C 2-3)

単純なルールは 3 種類全て，複雑なルールは提案 DSL と Java でルールを記述し比較を行った．複雑なルールでは拡張機能の追加を行っている．

ルール検査において，単純なルールは 91 ファイル中 11 ファイル，複雑なルールは 91 ファイル中 8 ファイル検出した．

ファイルの行数に対する，検査時のメモリ使用量及び実行時間 (5 回計測した平均値) について比較を行う．この時，検出したファイルとそれ以外で検査項目の値に挙動の違いがあったので，それぞれ分けて比較を行った．

4.3.1　検出箇所あり

図 13, 14 の縦軸はメモリ使用量 (Byte) と時間 (秒)，横軸はファイルの行数である．図 13, 14 より，以下のことが言える．

XPath　ファイル行数に比例して増加する．

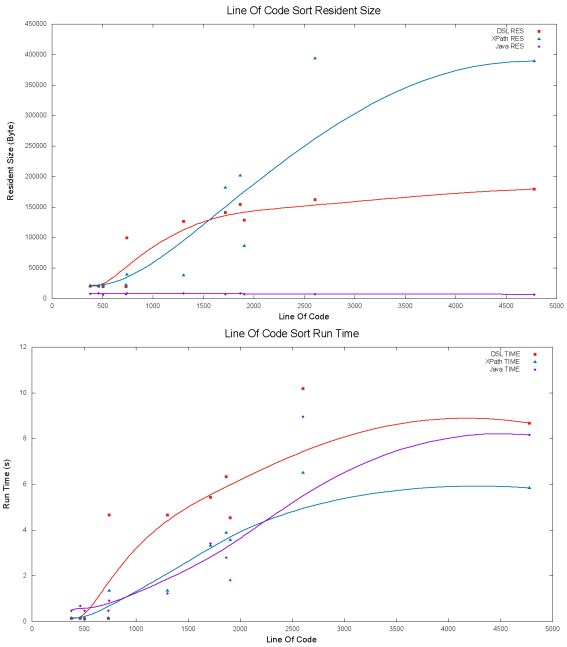
Java　メモリ使用量は一定だが，実行時間はファイル行数に比例して増加する

DSL　ファイル行数に比例して増加するが，ファイル行数等以外の要因が大きい．

4.3.2　検出箇所なし

図 15, 16 の縦軸はメモリ使用量 (Byte) と時間 (秒)，横軸はファイルの行数である．図 15, 16 より，以下のことが言える．

図 13　単純なルール，検出箇所あり



XPath　ファイル行数にかかわらずほぼ一定

Java　ファイル行数にかかわらずほぼ一定

DSL　ファイル行数等との相関がみられない．関係性を見つけるには内部要素のさらなる調査が必要．

4.3.3　言語間の比較

図 13 より，メモリ使用量は概ね XPath と Java の中間あたりと言える．実行時間は XPath や Java と比べて時間がかかっているが，平均 1.5 秒の増加にとどまっている．以上の結果より，提案 DSL は若干のオーバーヘッドを含むが従来の言語と同等であることを確認した．

4.3.4　考察

検出有りの場合，ルール検査を行う過程で検査対象の要素を保持するために余分にメモリを確保する必要がある．ソースコードの行数が増えるにつれて検査対象の要素や処理時間が増加すると考えられる．

検出無しの場合，検査対象の要素を保持する必要がない，もしくは少なくて済むと考えられる．よって，ソースコードの増加の影響を受けにくくメモリ使用量，処理時間がほぼ一定であると考えられる．

ただし，提案 DSL に拡張機能を追加した場合はその限りでない．これは，拡張機能が選択した複数の要素に対して処理を行う機能で，コード行数以外の要因に影響を受けていると考えられる．規則性を発見するためには検査対象のプログラムを構成する要素に関する調査を行う必要がある．

図 14 複雑なルール，検出箇所あり

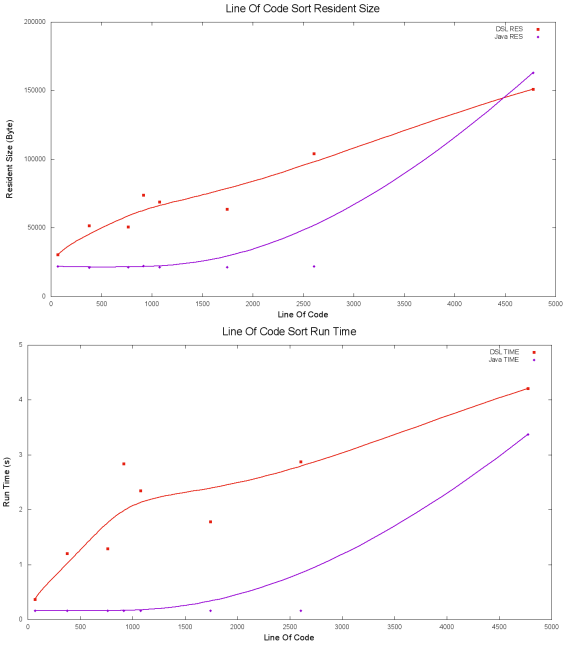


図 15 単純なルール，検出箇所なし

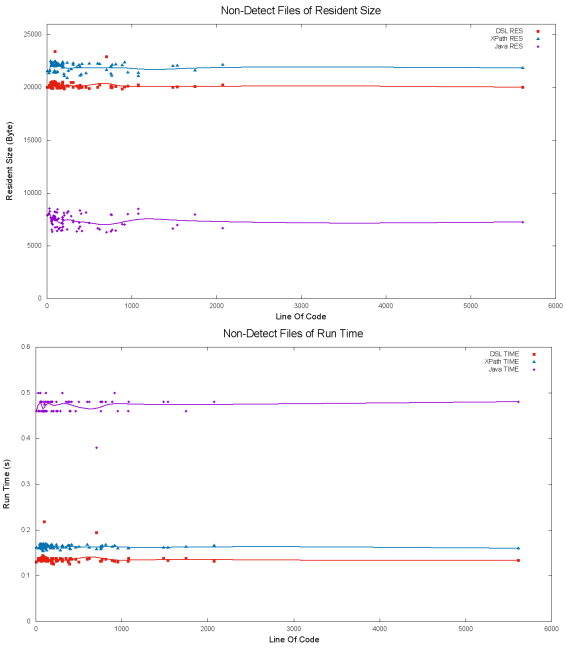
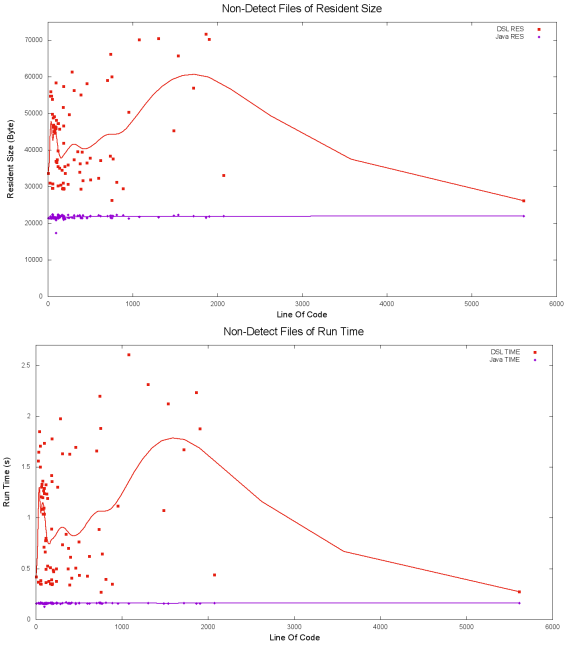


図 16 複雑なルール，検出箇所なし



5. おわりに

5.1 まとめ

CX-Checker のためのルール記述用 DSL の提案と実装を行った．DSL の機能として提供するメソッド名を C 言語の知識から連想できる機能名にすることで，CX-Model やその内部構造を意識せずにルールを記述できるようにした．また，型情報を利用する際に複雑な条件式を構築することなく，簡単に扱えるような検索機能を用意した．さらに，利用者が独自機能を追加することで提案 DSL の拡張性も確保した．

評価として公的な規約文書のルールを記述することで記述性を比較し，既存のルール記述言語の 1 つである Java と同等の記述力を持ち，簡潔に記述可能であることを示した．また，ルール検査におけるメモリ使用量と実行時間を比較することで，提案 DSL が若干のオーバーヘッドを含むが従来の記述言語と同等であることを確認した．

5.2 今後の課題

検査対象の要素に対する処理機能を追加することで記述可能なルールを増やすこと，また，提案 DSL を内部データを意識することなくルール記述を行える形式に発展させるの 2 点が挙げられる．

謝辞 本研究は JSPS 科研費 24300006 の助成を受けたものである．

参考文献

[1] MISRA: *MISRA C 2012 : Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Research Association(2013)
[2] JPCERT, CERT C コーディングスタンダード, `https://www.jpcert.or.jp/sc-rules/`
[3] Programming Research, QA・C, `http://www.programmingresearch.com/products/qac/`
[4] 大須賀俊憲, 小林隆志, 渥美紀寿, 間瀬順一, 山本晋一郎, 鈴村延保, 阿草清滋: CX-Chekcer: 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, 情報処理学会論文誌 Vol.53, No.2, pp.590-600, Feb.
[5] 福安直樹; 山本晋一郎; 阿草清滋.: 細粒度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, 1998, 39.6: 1990-1998.
[6] W3C, XML Path Language (XPath) Version 1.0, `http://www.w3.org/TR/xpath/`
[7] Debasish Ghosh： 実践プログラミング DSL ドメイン特化言語の設計と実装のノウハウ (佐藤 竜一 (監修, 翻訳)), 翔泳社 (2012)