情報処理学会論文誌　プログラミング　Vol.9　No.4　1–15 (Sep. 2016)

規則違反コードの構造を反映した木パタンを用いるコード検査器

中村 真也[1,a)]　鵜川 始陽[2,b)]　馬谷 誠二[3,c)]

受付日 2016年1月28日, 採録日 2016年4月25日

概要：プログラマはコーディング規約や規則に従わねばならない．ライブラリには，API 呼び出しの手順などに特有の規則を定めているものがあり，規則に従わないプログラムは正しく動作しない．また，ソフトウェアの脆弱性の原因となる危険なコードパタンや，それを避けるためのコーディング規約がよく知られている．このような規則や規約に違反したコードを発見するために，多くの検出ツールが開発されている．しかし，規則や規約ごとに特化した検出ツールを作成するのは労力がかかる．検出モジュールだけ分離して開発できる機構を提供しているツールも存在するが，その多くは (1) 検査対象コードの内部表現が専用のデータ構造で表現されており，可視化ツールを使わなければ構造を確認できない，(2) 内部表現には API 経由でアクセスしなければならない，(3) 検出モジュールは手続的なプログラムとして記述する必要があるという特徴があり，その開発は容易ではない．そこで，我々は，違反しているコードの構造を木パタンで表現し，ソースコード中からパタンにマッチする箇所を探すことで違反を検出するツールを提案する．このツールは，Clang コンパイラをもとに作った抽象構文木（AST）を生成する部分と，Clojure で記述された木パタンマッチ部で構成されている．規則違反パタンの作成者は，規則違反を含むソースコードの AST を見ながら規則違反パタンを記述できる．さらに，検査対象の AST と規則違反のパタンのどちらも同じ形式の S 式で表現されており，AST から一部を切り出して修正することで規則違反パタンを容易に作成することもできる．さらに，提案ツールの有用性を確認するため，実際にセキュアコーディング違反と JNI コーディング規約違反のうちいくつかのパタンの記述を行った．

キーワード：デバッグ，バグパタン，セキュリティ，JNI，S 式，木パタン，パタンマッチ

A Code Checker That Uses Tree Patterns Reflecting the Structures of Rule Violation Code

Shinya Nakamura[1,a)]　Tomoharu Ugawa[2,b)]　Seiji Umatani[3,c)]

Received: January 28, 2016, Accepted: April 25, 2016

Abstract: Programmers must obey some coding rules or coding standards. For example, some libraries require the programmers to obey their own rules, such as the order of API calls. If a program violates one of these rules, it does not work correctly. As another example, many coding patterns leading to vulnerability and coding rules to avoid such coding patterns are widely known. Various tools that detect violations of these rules and standards are developed. However, it is costly to develop such checking tools for each individual violation. Although some tools allow users to develop violation detection mechanisms as separate modules, the development of detection modules is not straightforward because (1) these tools have the target source code in their own internal representations, and so we need some visualizers to look at the structure of the source code, (2) we also need to access the structure through their own API, (3) detection modules are expected to be written in an operational style. We propose a rule violation detection tool for C and C++, in which the structure of the code fragments violating a rule is represented as a tree pattern. Our tool searches for locations of source code that matches with the specified patterns. This tool comprises two parts; the abstract syntax tree (AST) generator built on the Clang compiler, and the tree pattern matcher written in Clojure. Pattern writers can look at the AST of a source code containing the rule violation while writing the pattern. Moreover, since both the AST and the pattern are represented as S-expressions, the pattern writers can compose a pattern by modifying a fragment extracted from the AST. To verify the usability of our tool, we wrote several patterns selected from secure coding standards and JNI coding rules.

Keywords: debug, bug pattern, security, JNI, S-expression, tree pattern, pattern match

[1] 有限会社がんばりや
GANBARIYA CO., LTD., Tokushima 770–0865, Japan
[2] 高知工科大学
Kochi University of Technology, Kami, Kochi 782–8502, Japan
[3] 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[a)] nakamura@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp
[c)] umatani@kuis.kyoto-u.ac.jp

1. はじめに

プログラムを書く際にはプログラム中に誤りがないように注意をする必要がある．正しく設計されたプログラムでも，プログラムを書く際に誤りが混入して正しく動作しなかったり，ソフトウェアの脆弱性の原因になったりすることはよくある．このようなプログラム記述の際に混入する誤りの中でも，プログラミング言語の文法には従っており，型違反もないような誤りは，一般的なコンパイラが自動的に検出することができず厄介である．本論文では，文法や型の規則には従っている誤りを意味的誤りと呼ぶことにする．たとえば，C 言語で，if 文の条件部に 2 つの値が等しいかを判定しようとして，誤って代入をするコードを書いてしまうと，意味的誤りとなる．

近年ではソフトウェアの脆弱性が大きな問題となり，安全なプログラムを書くためのガイドラインが研究されている [13]．CERT [*1]では，安全な C 言語プログラムを書くためのコーディング規約を公開しているが，そこに列挙される規則の多くは，意味的誤りにつながるコーディングを禁止するものである．

プログラムでライブラリを利用する場合にも，意味的誤りに気をつける必要がある．ライブラリを利用してプログラムを書く場合，そのライブラリの API 呼び出し順序などの規則に違反した使い方をしてしまうと正しく動作しない．しかし実際には，ライブラリの API 仕様を正しく理解していなかったり，不注意によって API が正しく使われないことはよくある [5], [8]．

図 1 は，Android バージョン 2.3.7 のフレームワーク中にある意味的誤りを簡略化したコードである．読みやすさのために，誤りの本質でない箇所を削り，変数名や関数名を付け替えている．このコードは C 言語で Java のメソッドを記述するためのライブラリである JNI（Java Native Interface）ライブラリを使っている．このコード中には，JNI ライブラリの利用規則に違反する点が 1 カ所と，違反につながりやすい点が 1 カ所存在する．

JNI ライブラリでは，Java のオブジェクトを指す参照を `jobject` 型で扱う．Java のオブジェクトを指す参照が C 言語で定義したメソッドに渡されるとき，その参照はメソッドの実行中でのみ有効な局所参照として渡される．したがって，その参照を static 変数などのメソッドの寿命を越えて生存する変数に保存し，後の呼び出しでその参照を使って Java のオブジェクトにアクセスすると，正しく動作しない恐れがある．図 1 では 4 行目で局所参照を static 変数に保存している．局所参照を保存するだけで，後の呼び出しで利用しなければプログラムの誤動作にはならないが，保存すれば後に利用する可能性は高い．また，

[*1] http://www.cert.org/

```
1  void f(jobject obj) {
2    static jobject prev = NULL;
3    if(prev != obj) {
4      prev = obj;
5    }
6  }
```

図 1　文献 [16] に掲載された Android のフレームワーク中の意味的誤りを簡略化したコード

Fig. 1　Simplified code containing semantic errors from Android framework. Reported in Ref. [16].

参照どうしが同じオブジェクトを指しているかどうかは `IsSameObject` 関数を使って調べる必要があるが，3 行目では C 言語の演算子を使って調べている．

安全なプログラムを書くためのコーディング規約には，一般的なコンパイラでの検出が難しいものが多い．また，ライブラリの仕様違反はライブラリごとの仕様に依存するため，原理的に，一般的なコンパイラで自動的に検出することは不可能である．

そこで，API の規則やコーディング規約に違反したソースコードを発見するために，多くの検出ツールが開発されている [3], [5], [9], [16]（2 章で詳細を示す）．本論文では，これらの中でもプログラムの構文や型など，コンパイラのフロントエンド程度の解析で発見できる規則違反を対象に考える．たとえば，著者らは図 1 のような JNI の規則違反を検出するツール SEAN [9] を開発した．SEAN は，Java オブジェクトへの参照を static 変数に代入している箇所を検出して報告する機能を持っている．しかし，規則や規約ごとに特化した検出ツールを作成するのは労力がかかる．規則違反を検出するプログラムをツール本体と分離して開発できる機構を提供している検出ツールも存在するが，その多くは

- 検査対象ソースコードの内部表現が専用のデータ構造で表現されており，可視化ツールを使わなければ構造を確認できない，
- 内部表現には，そのツールが提供する専用の API 経由でアクセスしなければならない，
- 検出モジュールは手続的なプログラムとして記述する必要がある，

という問題があり，規則違反の検査プログラムを開発するのが容易ではない [12]．

本研究では，C/C++言語のプログラムを対象とした，検出モジュールの記述が容易であるという特徴を備える規則違反検出ツール ASTgrep を提案する．検査モジュールの記述を容易にしているのは，ASTgrep の以下の設計である．

まず，ASTgrep は検査対象となるプログラムを構文解析し，Clojure 言語の S 式（マップやベクタが利用でき，JSON に近い）で表現された抽象構文木（AST）を生成する（以降，S 式で表現された抽象構文木を S-AST と呼ぶことにする）．S-AST は人間が読むのに適しており，また，

```
1  [{:kind "Funcdef" :name "f"
2    :type ···
3    :parm [{:kind "Parm" :name "obj" :scope "local"
4            :type [{:kind "typedef-type" :typedefname "jobject"}}]]
5    :body [{:kind "Var" :name "prev" :scope "local"  :static "true"
6            :type [{:kind "typedef-type" :typedefname "jobject"}]}
7           {:kind "If" :condition {:kind "Binop" :op "!="
8                                   :type [{:kind "typedef-type" :typedefname "jobject"}]
9                                   :LHS {:type [{:kind "typedef-type" :typedefname "jobject"}]}
10                                  :RHS {:type [{:kind "typedef-type":typedefname "jobject"}]}}
11            :then [{:kind "Binop" :op "=" :type [{:kind "typedef-type" :typedefname "jobject"}]
12                    :LHS {:kind "VarRef" :name "prev" :scope "local" :static "true"
13                          :type [{:kind "typedef-type" :typedefname "jobject"}]}
14                    :RHS {:kind "VarRef" :name "obj" :scope "local"
15                          :type [{:kind "typedef-type" :typedefname "jobject"}]}}]}]}]
```

図 2　図 1 のプログラムの S-AST
Fig. 2　S-AST generated from the program shown in Fig. 1.

専用の API を呼び出さずとも，表示するだけでその内部構造が一目瞭然なため，検出モジュールの開発者は S-AST を見ながら検出モジュールを開発できる．**図 2** に図 1 から作った S-AST を示す．ただしスペースの都合で，一部を省略している．11 行目から 15 行目が，局所参照の static 変数への代入（図 1 の 4 行目）に対応している．

次に，検出モジュールの開発者は，検出したい誤りのコード片の形，すなわち，S-AST のパタンを表す木パタンで宣言的に記述する．したがって，検査モジュールを手続的なプログラムとして記述するよりも直観的に書くことができる．

さらに，検出モジュール中に記述する木パタンは，S-AST と同じ文法の S 式である．したがって，検査モジュールの開発者は，誤りを含むプログラムから S-AST を作り，その中の誤りに相当する箇所を切り出してきて，それをもとにして規則違反のパタンを作ることができる．図 1 の 4 行目に対応する部分を切り出すと，

```
1  {:kind "Binop" :op "="
2   :type [{:kind "typedef-type"
3           :typedefname "jobject"}]
4   :LHS {:kind "VarRef" :name "prev"
5         :scope "local" :static "true"
6         :type [{:kind "typedef-type"
7                 :typedefname
8                  "jobject"}]}
9   :RHS {:kind "VarRef" :name "obj"
10        :scope "local"
11        :type [{:kind "typedef-type"
12                :typedefname
13                 "jobject"}]}}
```

となる．このうち，規則違反の特徴をとらえている箇所だけを残すと，

```
1  {:kind "Binop" :op "="
2   :LHS {:static "true"}
3   :RHS {:type [
4         {:typedefname "jobject"}]}}
```

となる．これがそのまま「static 変数に jobject 型の変数を格納してはならない」という規則の違反を検出するパタンになる．

このように，検査対象を見ながら，その一部を抽出・改変しながらパタンを作って検査することができるので，ASTgrep は Unix の grep コマンドのように簡単に使うことができる．

本論文の貢献は以下のとおりである．

- 木パタンマッチを行う Clojure ライブラリである bc-macro に機能を拡張し，パタンのデバッグを容易にした（3 章）．
- 誤りのパタンを記述しやすい違反検出器 ASTgrep を設計した（4 章）．
- Clang コンパイラの抽象構文木を利用したソフトウェアを開発するためのフレームワークである LibTooling を利用して，S-AST の生成部を実装した（5 章）．
- SEAN [16] が検出する違反のパタンを ASTgrep 用に記述し，パタンの記述が容易であることを確認した．このパタンは，文献 [16] と同じ Android バージョン 2.3.7 のソースコードに適用して，同じ違反を検出できることを確認した（6 章）．
- CERT の Web ページで公開されているセキュアコーディング規約のうち，式に関する規約の違反を検出する ASTgrep のパタンが容易に記述できるかどうかを検討した（6 章）．

2. 関連研究

2.1 コーディング規約

ライブラリを正しく利用するのは難しく，意味的誤りの原因になりやすい．たとえば，C 言語の標準ライブラリにある strcpy 関数が文字列長を引数にとらないので，バッファオーバフロー脆弱性の原因になりやすいということはよく知られており，一部の C コンパイラでは，このような脆弱性の原因になる関数の利用を警告するオプションがある．Java のメソッドを C 言語や C++言語で記述するためのライブラリである JNI は，仕様が複雑で，プログラマは意味的誤りのあるプログラムを書きやすい．JNI のプログラマーズガイド [8] では，プログラマが陥りやすい意味的

誤りのパタンが 15 個列挙されている．また，JNI を正しく使っていない箇所を検出する研究もさかんに行われている [2], [6], [7], [9], [16].

西脇らは JNI ライブラリを用いたプログラミングにおける参照の扱いに関する規則違反を研究し，規則違反につながる可能性が高いコードのパタンとして，次の 3 つをあげている [16].

(1) 局所変数以外への `jobject` 型の値の代入
(2) C 言語の演算子（`==`と`!=`）を使った `jobject` 型の値の比較
(3) `jobject` 型の値から他の型への型キャスト

Seacord は，C 言語で脆弱性を含まないプログラムを書くためのコーディング規約をまとめている [13]．これらのコーディング規約のほとんどは，意味的誤りか，意味的誤りにつながる可能性が高いコーディングを禁止するものである．このコーディング規約は JPCERT [*2]でも翻訳して公開しており，広く認知されている．

2.2　規則違反検出ツール

プログラムの構文上のパタンから意味的誤りを検出するための研究は，これまでにも多くなされている．SEAN は，上記の JNI ライブラリを使ったプログラムの参照の扱いに関して，規則違反につながる可能性が高いコードのパタンを検出するツールである [9], [16]．SEAN は Clang コンパイラ [4] を改変して作られており，構文解析直後に，Clang が内部的に保持している構文木をビジターを使ってたどって，問題となる箇所を探す．このように，コンパイラを改変すれば，フロントエンドが構文解析した結果の構文木にアクセスでき，規則違反を探すことができる．

この考えを推し進めて，コンパイラインフラストラクチャを規則違反検出ツールのフレームワークとして用いるシステムもある．そこでは，構文木にアクセスして規則違反を探すプログラムを，独立したソースファイルで記述できるようになっており，高い拡張性を提供している．たとえば，CERT Rosecheckers [*3]は ROSE コンパイラインフラストラクチャ [11] を使った，そのようなシステムであり，ソフトウェアの脆弱性の原因になりそうな箇所を探すのに用いられている．

しかし，コンパイラインフラストラクチャの構文木を直接操作して規則違反を探すプログラムを作成するのは簡単ではない．**図 3** に，文献 [12] に掲載されている，Rose コンパイラインフラストラクチャで default 節がない switch 文を探すためのプログラムを示す．ただし，スペースの都合により多少改変してある．このプログラムでは，構文木をたどるビジターを定義している．switch 文に対応するノードであれば（7 行目），その子ノードのイテレータを取得し

```
 1 #include "rose.h"
 2 class visitorTraversal
 3   : public AstSimpleProcessing {
 4 public:
 5 virtual void visit(SgNode* n) {
 6   SgSwitchStatement* s =
 7     isSgSwitchStatement(n);
 8   if(s) {
 9     SgStatementPtrList& cases =
10       s->get_body()->get_statements();
11     bool switch_has_defaults = false;
12     SgStatementPtrList::iterator i =
13       cases.begin();
14     while (i != cases.end() &&
15            !switch_has_default) {
16       if (isSgDefaultOptionStmt(*i))
17         switch_has_default = true;
18       i++; }
19     if (!switch_has_default)
20       s->get_startOfConstruct()
21        ->display("Error"); }}};
```

図 3　Rose 用規則違反検査モジュール（文献 [12] より）
Fig. 3　Error detection module for Rose compiler (reported in Ref. [12]).

(12 行目)，各子ノードが default 節であるかどうかを調べている（16 行目）．このプログラムを作成するためには，ドキュメントや，場合によってはコンパイラインフラストラクチャのソースコードを調査して，構文木の構造やそれにアクセスするメソッド（`isSgSwitchStatement` など）を探す必要がある．また，手続的な記述なので，どのようなパタンを探しているのかが直観的に分かりにくいという欠点もある．

これに対して Quinlan らは，Rose コンパイラインフラストラクチャの構文木の中間表現を Datalog のデータベースに格納することで，規則違反のパタンを Prolog のプログラムとして宣言的に記述して検査できるようにした [12]．Quinlan らは，構文木に直接アクセスする手法は「うんざりする」が，Datalog を使うと非常にシンプルに宣言的に記述できる（ただし Datalog の宣言的な記法は一般のプログラマに広く普及しているとはいえない）と結論づけている．Clang コンパイラも，宣言的な記述で Clang の構文木にパタンマッチできるドメイン特化言語である ASTMatchers を提供している．Ramanathan らは ASTMatchers を利用して，C++のテンプレートライブラリである STL の誤用を検出するシステムを開発した [5]．これらの研究では，規則違反を検査するプログラムを見通しよく宣言的に記述することができる．しかし，構文木の構造や構文木にアクセスするために使う述語は，やはりドキュメントを調べる必要がある．一方で，本研究で開発した ASTgrep では，規則違反を含むプログラムを入力すれば，その構文木の S 式表現が得られるので，構文木の構造は自明に推測できる．さらに，その一部を切り取って規則違反を検出するためのパタンを作成できるので，構文木にアクセスするためのメソッドや述語を調べる必要もない．

Coccinelle [1], [10] は C プログラムに対する誤りの検出や修正，リファクタリングを容易に行うためのツールである．

[*2] http://jpcert.or.jp/
[*3] http://rosecheckers.sourceforge.net/

コードに対するパタンマッチによって規則違反を検出する点はASTgrepと同様であるが，(1) マッチング対象はプログラムの構文木ではなく制御フローグラフである，(2) パタンはCコードの断片を組み合わせたようなドメイン特化言語により記述する，(3) マッチしたパタン中の一部を自動で書き換えられる，という違いがある．(1), (2) については設計方針の違いであり，特に，対象が構文木か制御フローグラフかにより表現可能な規則違反の種類にも違いがある（それぞれに長所・短所がある）と考えられるが，系統的な比較はまだ行っていない．しかし，たとえばASTgrepでは，後述のとおり，構文木の各ノードに型情報が埋め込まれており，その情報をうまく利用して簡潔なパターンを記述可能である．一方，Coccinelleの制御フローグラフやパタン言語には，任意の位置に型情報を埋め込む機能は備わっていない．

2.3 構文木のテキスト表現の利用

SC [17] はS式の構文とC言語の意味論を備えたプログラミング言語である．S式で表現されたC言語のプログラムを，Lispのマクロを使って変換するために使われる．SCはプログラマが直接S式のコードを記述することを念頭に設計されており，C言語とほぼ一対一に対応した，プログラマが記述するのに適した構文になっている．一方で，本研究で用いるS-ASTは，各ノードにノードの種類や型情報を持たせるなど，局所的な情報のみで様々なパタンマッチができるように工夫している．

プログラムをXMLに変換して，XMLの操作でプログラム変換を実現する研究もある．Takizawaらは，ROSEコンパイラインフラストラクチャを使って，入力プログラムの構文木を生成しXMLとして出力した [14]．これに，XSLTで宣言的に記述された変換規則を適用してプログラム変換を行った．Takizawaらの手法は，本研究で開発したASTgrepと同様に，変換規則の開発者が変換対象となる構文木を見ることができ，変換規則を宣言的に記述できるという利点がある．しかし，構文木を表すXMLの一部を切り取って，そのまま変換規則のパタンマッチ部として利用することは想定していない．

3. bcmacro

本章では，提案ツールのパタンマッチ部に用いられるClojure用木パタンマッチライブラリbcmacroについて述べる．3.1節では，構文木を対象にした規則違反やコーディング違反の検出の際に頻繁に用いられるbcmacroの基本的な機能について説明し，3.2節では，ASTgrepのインタラクティブ性を向上させるために追加した新たな機能について説明する．bcmacroの詳細については，文献 [15] を参照されたい[*4]．

[*4] 本論文中に現れるパタンの記法は，文献 [15] で説明されている古いbcmacroのものと若干異なる．

3.1 基本機能

1章で例示した簡単なパターンのように，bcmacroではClojureの基本データ型である文字列，数，記号，キーワード，ベクタ，マップを用いて表現された木構造を対象とする．たとえば，

```
(def tree {:v [1 2 3] :m {"pi" 3.14 "e" 2.7}})
```

と定義されるマップ`tree`は，キー`:v`，`:m`に対応する値（ベクタと別のマップ）を子ノードとする木である（文字「:」で始まる識別子はキーワード）．また，`:v`に対応する子ノードのベクタは，その子ノードとして整数値`1`，`2`，`3`を含んでいる．

bcmacroの木パタンの記法は，上記のClojureの基本データ型を表すためのリテラルと同じである．たとえば，パタン`2`は整数値`2`とマッチし，パタン`[1 2 3]`や`{"pi" 3.14 "e" 2.7}`は，`tree`の2つの部分木にそれぞれマッチする．さらに，ベクタやマップを任意に入れ子に組み合わせたものもパタンとなるため，

```
{:v [1 2 3] :m {"pi" 3.14 "e" 2.7}}
```

は`tree`全体とマッチする．なお，マップパタンではすべてのキー/値の組を指定する必要はなく，たとえばこの場合だと`{"pi" 3.14}`もマッチする．

Clojureの記号をパタンとして用いることもでき，任意の値とマッチする．さらに，その記号の名前を持つ変数がマッチした値へと束縛され，後述の書き換え規則を記述する際に利用することができる．たとえば，

```
{:v [1 x 3] :m y}
```

のようなパタンは`tree`にマッチし，変数`x`，`y`はそれぞれ`2`，`{"pi" 3.14 "e" 2.7}`に束縛される．

ベクタに対するパタンを記述する際，要素の並びの一部を省略したい場合がある．そのような場合のため，bcmacroでは省略パタン`...`が用意されている．たとえば，`[1 ...]`は先頭要素が`1`である長さ1以上のベクタにマッチし，`[... 3]`は最後の要素が`3`である長さ1以上のベクタにマッチする．`...`は任意の位置に好きな数だけ埋め込めるため，たとえば，`[... 2 ... 3 ...]`と書くと，`2`と`3`を（左からこの順に）含むベクタにマッチする．

さらに，任意の深さの入れ子を省略するための省略パタン`#nest`も用意されている．たとえば，

```
{:v #nest 2}
```

は，キー`:v`に対応する値（木）の子孫ノードとして`2`が含まれていればマッチする[*5]．

bcmacroでは次のようにパタンを定義する：

(**defbcmacro** ⟨ルール名⟩ ⟨パタン⟩ ⟨置き換え式⟩)

ここで⟨ルール名⟩はパタンに付ける名前でありパタンマッチの際に使用する．⟨パタン⟩は上述の記法で書いた

[*5] 正確には，パタン中の`#nest`を書ける位置には，次に述べる`@`と関連する制約があるが，本論文ではその点については詳しく触れない．

パタンである．〈置き換え式〉は任意の Clojure 式であり，次に説明する木の書き換えで使用される．

bcmacro は本来，木（の一部）を自動的に書き換えるのに用いるよう設計されている．たとえば，

```
(defbcmacro m1 @[1 x 3] [x x])
```

によって定義される書き換え規則では，パタン `[1 x 3]` にマッチする部分木を，`[x x]` によって生成されるベクタと置き換えることを指定している．ここで，パタン中の`@`は，マッチした部分木のどの部分を置き換えるかを指定するプレフィックスである．上の例では，マッチした部分木全体を置き換えている．また，木の生成には，任意の Clojure コードを書くことができる．`@`プレフィックスをつける位置を動かすことで，置き換え対象を変えることもできる．たとえば，

```
{:v [1 x 3] :m @y}
```

というパタンでは，キー`:m` に対応する部分木だけが書き換えられることを指定している．

なお，ASTgrep は構文木中のマッチする部分木の検出のみが目的であるため，bcmacro の書き換え機能については，これ以上触れない．実際には，ASTgrep は，`defbcmacro` 中の木生成コードにおいて，（適当な木への置き換えの）副作用として違反コードを検出した旨のメッセージを表示する．

上記に加え，bcmacro のパタンは以下のような便利な機能も持つ．まず，(p `:as` v) と書くことで，任意の部分パタン p にマッチする部分木に，変数 v を束縛できる．また，(p `:when` *pred*) と書くことで，p にマッチし，なおかつ，Clojure 式 *pred* が真であるような場合にのみマッチするパタンをつくることができる．これらを使用した具体的なパタンは，6 章の例のいくつかに含まれている．

3.2　インタラクティブなデバッグのための追加機能

前節で述べたとおり，bcmacro のパタン記法は Clojure のリテラル記法と同じなため，意味的誤りを含むプログラムの S-AST を表示しつつ，それにマッチするパタンを作成するのが容易である．また，パタン自身が S 式で表現されているため，パタンの構造に従ってその一部の無効化を行うことができれば，規則違反を正確に表現しているかをインタラクティブにデバッグする際，非常に便利である．

パタンの一部を無効化する 1 つの方法として，Clojure のリーダに標準で備わっている `#_` を用いることが考えられる．プログラム中に `#_` を書くと，直後の S 式が読み飛ばされる．たとえば，以下のような（図 2 の一部にマッチするはずの）複雑なパタン：

```
1  [{:kind "Funcdef"
2    :parm
3    [{:kind "Parm"
4      :name n
5      :type [
6        {:typedefname "jobject"}}]]
7    :body
8    [{:kind "Var"
9      :scope "local"
10     :static "true"
11     :type [{:kind "typedef-type"
12             :typedefname "jobject"}]}
13    ...
14    {:kind "If"
15     :then @[{:kind "Binop"
16              :op "="
17              :LHS {:static "true"
18                    :type [
19                     {:typedefname
20                        "jobject"}]}
21              :RHS {:kind "VarRef"
22                    :name n}}]}]}]
```

を規則違反を表現するパタンとして書き下してみたものの，想定している意味的誤りコードの構文木にどうしてもマッチしない場合，1，8，17 行目に`#_`を挿入することで，より簡単なパタン：

```
1  [{:parm
2    [{:kind "Parm"
3      :name n
4      :type [{:typedefname "jobject"}]}]
5    :body
6    [...
7     {:kind "If"
8      :then @[{:kind "Binop"
9               :op "="
10              :RHS {:kind "VarRef"
11                    :name n}}]}]}]
```

へと容易に変更できる．

このように，S 式中の部分式を無効にする機能は一時的なパタンの単純化を行うのに非常に有用であるが，逆に，ある部分式のみを有効にできるなら，より柔軟にパタンの単純化が可能である．しかしながら，Clojure のリーダにはそのような機能は備わっていない．そこで，我々は bcmacro のパタン中に`#use` を挿入すると，その直後の S 式が表す部分パタンのみが有効になる機能を追加した．たとえば，先程の例に`#use` と`#_`を挿入して，

```
1  [{#_:kind #_"Funcdef"
2    :parm
3    [{:kind "Parm"
4      :name n
5      :type [
6       {:typedefname "jobject"}}]]
7    :body
8    [#_{:kind "Var"
9      :scope "local"
10     :static "true"
11     :type [{:kind "typedef-type"
12             :typedefname "jobject"}]}
13    ...
14    {:kind "If"
15     :then #use @[{:kind "Binop"
16              :op "="
17              #_:LHS #_{:static "true"
18                    :type [
19                     {:typedefname
20                       "jobject"}]}
21              :RHS {:kind "VarRef"
22                    :name n}}]}]}]
```

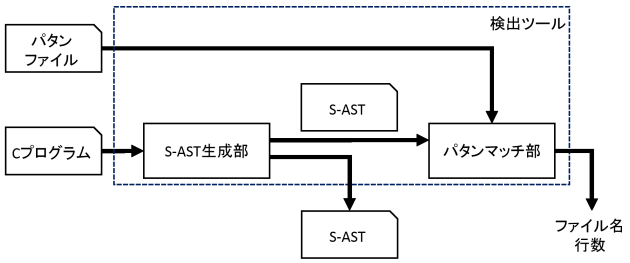

図 4　ASTgrep の全体像
Fig. 4　Overview of ASTgrep.

とすれば，単純なパタン：

```
1  @[{:kind "Binop"
2     :op "="
3     :RHS {:kind "VarRef"
4           :name n}}]
```

となる．

4. ASTgrep

本研究では，C/C++言語で記述されたプログラムコードを対象とする．検査対象の C/C++言語プログラムと規則違反のパタンを入力として受け取り，検査対象のプログラム中の，規則違反がある位置を報告するツール ASTgrep を開発した．

4.1 設計

ASTgrep は，与えられた C/C++言語のプログラムを S 式で表した抽象構文木（S-AST）に変換する **S-AST 生成部**と，S-AST と与えられた規則違反のパタンを用いてパタンマッチを行う**パタンマッチ部**の 2 つのモジュールから構成されている．

ASTgrep の全体像を図 4 に示す．ASTgrep への入力は，検査対象の C/C++言語のプログラムと，規則違反のパタンを記述したパタンファイルである．S-AST 生成部は，与えられたプログラムを Clang コンパイラを用いて構文解析して，S-AST を生成する．ここで生成された S-AST は一時ファイルとして保存される．パタンマッチ部は Clojure のプログラムであり，S-AST 生成部が生成した S-AST を対象に，bcmacro を用いて与えられたパタンのパタンマッチを行い，規則違反の箇所を検出する．パタンにマッチした場合には，もとのソースコードでのパタンの出現位置（ファイル名と行数）が出力される．また，パタンの開発者が，規則違反のあるプログラムの S-AST を見ながらパタンを記述できるように，S-AST を保存したファイルを出力する機能も備える．

4.2 パタンファイル

パタンファイルは，Clojure プログラムで記述する．規則違反の単純なパタンは，

`(defrule` 〈ルール名〉〈パタン〉`)`

によって記述できる．パタンファイルには，このような記述を羅列する．ASTgrep は，パタンにマッチする箇所を見つけると，

`Rule` 〈ルール名〉`violation:`〈ファイル名〉`:`〈行数〉

のように，どのファイルの何行目がどのルールに該当したのかを警告する．

パタンファイルは Clojure のプログラムなので，単純なパタンでは表せない規則違反も記述することができる．たとえば bcmacro には，否定のパタンは存在しない．しかし `:when` のパタンで，ある条件にマッチしないときに真になる Clojure のプログラムを書くことで，否定のパタンを表現できる．また，たとえば木の 2 カ所に同じ形の部分木が現れるというようなパタンも直接は書けない．このような複合的なパタンのパタンマッチは，パタンマッチを繰り返し使うことで，現実的な複雑さの記述で（計算時間はかかる可能性はあるが）実現できる．

5. S-AST 生成部

S-AST 生成部では，Clang コンパイラを用いて構文解析を行い必要な情報を取り出しパタンマッチしやすい形の木構造である S-AST へと変換する．

本研究では，LibTooling を用いて S-AST を生成する．LibTooling とは，Clang コンパイラの部品を使いつつ，Clang コンパイラとは独立したツールを作成するための C++インタフェースである．LibTooling を用いることで，Clang が解析した構文木の持つすべての情報にアクセスすることができる．S-AST 生成部は，その情報の中から必要なものだけを選択したり，情報を補ったりしてパタンマッチをしやすい S-AST を生成する．

5.1 設計

S-AST はベクタとマップを用いて，以下のような方針で表現する．

- 構文木のノードは，マップで表す．
- ノードの詳細情報である属性は，マップの要素で表す．
- 可変個のノードの集合や順序に意味がある列は，ベクタで表す．

ノードの属性には，ノードの種類，変数名やスコープ，型情報，もとのプログラム中での出現位置などが含まれる．また，ベクタは関数本体などのブロックや関数の引数の列，プログラム全体（宣言または定義の列）などに用いる．

具体的に，図 2 の S-AST を見ながら説明していく．この S-AST は図 1 の関数定義の部分を変換したものである．関数定義は図 2 の 1 行目にあるようにマップで表され，ノードの種類（**`:kind`**）が関数定義を表す`"Funcdef"`であり，ほかに関数名（**`:name`**），関数の型（**`:type`**），引数（**`:parm`**），関数本体（**`:body`**）などについての属性を要素として持つ．関数本体は，図 2 の 5 行目にある**`:body`** で

表されている．ここでは，関数本体に変数宣言と if 文が記述されている．それぞれもまた，マップで表されており，ノードの種類（**:kind**）がそれぞれ"Var"（5 行目），"if"（7 行目）となる．

5.2　工夫点

Clang の構文木は，基本的に DECL（宣言），STMT（文），EXPR（式）の 3 種類のノードで構成されている．構文木は，TRANSLATION_UNIT というノードを根に持ち，DECL，STMT，EXPR のノードは，子孫になる．変数宣言や関数定義の部分などが DECL となっており，変数の参照や関数呼び出しなどの部分は EXPR となる．STMT には，if 文や while 文などが当てはまる．このように構成される木構造を，ビジターを利用することで深さ優先でたどることができる．S-AST 生成部は構文木をたどりながら必要な情報を取得し，パタンマッチしやすい S-AST を生成する．ここでは，S-AST を生成する上で工夫した点について説明する．

5.2.1　Clang の構文木の情報の保存

ASTgrep で生成する S-AST には，Clang の構文木が含んでいる情報をできるだけ多く，マップの要素としてそのままを持たせるようにした．多くの情報を S-AST に残すことで，パタンを記述する際に細かい指定ができるため，余計な部分を省いて本当にマッチさせたいところにだけマッチさせることができる．

5.2.2　プログラム中の出現位置

パタンマッチを行った結果，マッチした箇所を出力するために，各ノードにはプログラム中の出現位置を表す属性を持たせる．ここで持たせる情報は出現するファイル名と行数，カラムである．これらは，**:loc-begin**，**:loc-end** として，例外的にベクタを使って表す．たとえば，sample.c というファイルの 23 行目の 2 列目から 25 行目の 1 列目に現れるノードの場合であれば S-AST は次のようになる．

```
1  {:kind ···
2   :loc-begin ["sample.c" 23 2]
3   :loc-end ["sample.c" 25 1]}
```

5.2.3　型情報

Clang では，ビジターがたどる構文木に型情報は現れないが，型をパタンマッチに利用できれば便利であると考え，ASTgrep では S-AST の各ノードに型情報を持たせた．Clang では，getType 関数で型を表す木構造がとり出せる．それを S 式に変換して，各ノードの**:type** 属性に持たせた．型キャストがある場合を考慮して，型情報はベクタで表した（詳細は 5.2.4 項参照）．

たとえば，char 型のポインタを表す場合には，次のようになる．

```
1  :type [{:kind "Pointer-type"
2          :Pointee
3           {:kind "Char-type"}}]
```

また，int 型変数を 2 つ引数に取り値を返さない関数の型は次のように表す．

```
1  :type [{:kind "Func-type"
2          :ParamsType [
3           {:kind "Int-type"}
4           {:kind "Int-type"}]
5          :RetType
6           {:kind "Void-type"}}]
```

関数型は引数の型と返り値の型をそれぞれ**:ParamsType** と**:RetType** の属性で表す．

5.2.4　型キャスト

型キャスト（明示的なものも暗黙的なものも含む）は，Clang の構文木では，子ノードにキャストされる式を持つノードとして表される．しかし，直観的には型キャストはプログラムの構造とは考えにくいので，キャストの情報をノードの属性として持たせることにした．

キャストがあった場合には，型を表すキー（**:type**）の属性として表す．型はベクタで表すことにし，その要素の数と位置でキャストの情報を表す．キャストが起こらない場合には，要素は 1 つだけである．キャストが起こると要素が複数になりキャストの回数だけ要素が増える．また，ベクタの最初の要素はそのノードそのものの型を表し，右の要素になるにつれてキャスト先の型を表す．つまり，先頭の要素がそのノードそのものの型で，最後の要素がキャストされた最終的な型となっている．たとえば，int 型の値を double 型の変数に代入しようとする場合，つまり，int 型から double 型へのキャストが起こる場合，S-AST は次のようになる．

```
1  {:kind "Binop" :op "="
2   :LHS {:kind "VarRef"
3         :type [{:kind "Double-type"}]}
4   :RHS {:kind "VarRef"
5         :type [
6          {:kind "Int-type"}
7          {:kind "Double-type"}]}}
```

ここで，5 行目と 7 行目のベクタが int 型から double 型へのキャストを表している．

このキャストの表現方法を使えば，キャストがある式でもキャストがない式でも，キャストをする前の式の型が τ であるというパタンは [τ ...]，キャストした結果が τ であるというパタンは [... τ] というように統一的に記述できる．また，型 τ から別の型へのキャストは，bcmacro の変数パタンを使って，[τ x ...] と記述できる．ここで x は変数である．

5.2.5　ラベル

ラベルも直観的にはプログラムの構造ではないため，ノードの属性とした．ラベルの情報は，ラベルが付いている部分のノードに**:label** という属性で持たせる．加えてラベルは，連続して現れる場合があるためベクタで表す．

例として，ラベルが付いたプログラム片

```
1   L1: L2: x++;
```

を考える．これは，2 つのラベルが x++; に付いているプログラム片である．これを S-AST に変換すると

```
1  {:kind "Unop" :op "++"
2   :label [{:kind "Label"
3            :labelname "L1"}
4           {:kind "Label"
5            :labelname "L2"}]
6   :operand ···}
```

のようになり，ラベルはノードの属性として表される．

5.2.6　可読性のための属性

5.2.3 項で型情報は，木構造にして S-AST の各ノードに持たせると述べた．それではパタンマッチには便利であるが，パタン開発者には読みにくい．そこで，ノードに **:displaytype** 属性として，型のプログラム上での表現も持たせることにした．

たとえば，サイズが 5 の char 型のポインタ配列を宣言しているプログラム片の S-AST は次のようになる．

```
1  {:kind "Var"
2   :displaytype "char *[5]"
3   :type [{:kind "Array-type"
4           :ArraySize "5"
5           :elementtype
6           {:kind "Pointer-type"
7            :Pointee
8            {:kind "Char-type"}}}]
9   ···}
```

このように，**:type** を見ただけでは分かりにくい型であっても，**:displaytype** を見ることで，どんな型であるかが簡単に分かる．

6.　ケーススタディ

本節では，SEAN [9] が検出する，JNI の参照に関する規則違反について，ASTgrep でうまくパタンを記述できるかを確かめ，さらに，規則違反を含む箇所を実際に検出できるかを検証した．さらに，JPCERT で紹介されているセキュアコーディング規約[*6]を対象に，ASTgrep で検出可能かどうかを検討した．

6.1　JNI 規則違反

JNI 規則違反については，

(1) jobject 型の値の static 変数への代入
(2) C/C++言語の演算子を用いた参照どうしの比較
(3) jobject 型の値から他の型へのキャスト

の 3 種類の規則違反について ASTgrep による検出を行う．JNI では，メソッドに渡された Java の参照を表す jobject 型の参照はそのメソッド内でのみ有効な局所参照となる．しかし，その局所参照を static 変数などのメソッドが終了しても値が保存されるような変数に代入しておくと，渡されたメソッド以外で利用される可能性があるため意味的誤りとなる危険性がある．そのため，SEAN は (1) を規則違反としている．SEAN は，static 変数のほかにも，大域変数や構造体のフィールドへの代入も規則違反としているが，ここでは static 変数への代入のみをとりあげる．(2) については，参照どうしの比較には，比較演算子ではなく IsSameObject 関数を用いる必要があるため，規則違反としている．jobject 型の値を他の型にキャストしてから，static 変数に代入したり，C/C++言語の演算子を用いて比較すると (1) や (2) の方法では検出できないので，SEAN は (3) の jobject 型の値を他の型にキャストすることも規則違反としている．

6.1.1　jobject 型の値の static 変数への代入

SEAN を使って実際に Android バージョン 2.3.7 のソースコードのバグを発見し，修正した文献 [16] では，**図 5** のプログラムに規則違反が見つかったと報告されている．図 5 は Android のフレームワークライブラリ中の frameworks/base/core/jni/android_text_format_Time.cpp（691 行のソースコード）の 187 行目から 305 行目にある関数定義である．図 5 の 8 行目の代入が (1) の規則に違反している．

ASTgrep を使って android_text_format_Time.cpp から S-AST を生成すると，3,894,069 バイトのファイルが生成された．これは，S-AST にはインクルードされているすべてのヘッダファイルの中に記述されたプログラムも含まれているからである．実際，S-AST の先頭から約 69%付近までは，

```
{:kind "Structdef" :name "_IO_FILE"
 :loc-begin ["/usr/include/stdio.h" 44 1]
 :loc-end ["/usr/include/stdio.h" 44 8]
 :Member []}
```

のようにヘッダファイル中の定義で占められていた[*7]．生成した S-AST から，もとのプログラムの行数や js_locale という変数名を使って絞り込むと，(1) の規則違反は，**図 6** に示す付近に存在することが分かった．特に，図 6 の 3 行目にある 237 行目という情報は，ソースコード中位置に対応するノードを探すのに有益であった．

(1) の規則違反を検出するためには，static 変数に jobject 型の局所参照を代入しているということが分かればよい．図 6 で，それに対応する情報は，

- 代入演算であること（1 行目），
- 代入先（左辺）が static 変数であること（8 行目），
- 代入元（右辺）が jobject 型の局所参照であること（17 行目）

の 3 つである．これだけの情報を残すと (1) の規則違反を検出するパタンは次のように作れる．

```
1  {:kind "Binop" :op "="
2   :LHS {:static "true"}
3   :RHS {:type
4         [... {:typedefname
```

[*6] https://www.jpcert.or.jp/sc-rules/

[*7] S-AST を保存したテキストファイル中で，**:loc-begin** 属性に android_text_format_Time.cpp が含まれる最初のノードの出現位置で判定した．

```
1  static jstring android_text_format_Time_format(JNIEnv* env, jobject This, jstring formatObject) {
2    static jobject js_locale_previous = NULL;
3    ...
4    jobject js_locale = (jobject) env->GetStaticObjectField(timeClass, g_localeField);
5    ...
6    if (js_locale_previous != js_locale) { /* 元のプログラム 205 行目 */
7      ...
8      js_locale_previous = js_locale; /* 元のプログラム 237 行目 */
9      ...
10   }
11   ...
12 }
```

図 5　JNI 規則違反 (1) と (2) を含んだソースコード（文献 [16] に掲載）
Fig. 5　Source code containing JNI rule violation (1) and (2) (reported in Ref. [16]).

```
1  {:kind "Binop" :op "="  :type ···
2   :loc-begin
3    ["···_text_format_Time.cpp" 237 9]
4   :loc-end
5    ["···_text_format_Time.cpp" 237 30]
6   :LHS {:kind "VarRef"
7         :name "js_locale_previous"
8         :scope "local"  :static "true"
9         :type [{:kind "Typedef-type"
10                :typedefname "jobject"
11                :typedeftype ···
12                :displaytype "jobject"}]
13        :loc-begin ··· :loc-end ···}
14   :RHS {:kind "VarRef" :name "js_locale"
15         :scope "local"
16         :type [{:kind "Typedef-type"
17                 :typedefname "jobject"
18                 :typedeftype ···
19                 :displaytype "jobject"}]}
20         :loc-begin ··· :loc-end ···}
```

図 6　規則違反 (1) に該当する部分の S-AST
Fig. 6　S-AST corresponding to the part containing the rule (1) violation.

```
5                  "jobject"}]}}
```

右辺の型については，キャスト後の型が jobject 型になっていることが重要である．

実際，このパタンを利用して android_text_format_Time.cpp を検査すると，文献 [16] が報告していた 237 行目が検出された．

6.1.2　C/C++言語の演算子を用いた参照どうしの比較

文献 [16] の報告によれば，図 5 の 6 行目で比較演算子（!=）をもちいて参照どうしを比較しており，(2) の規則に違反している．S-AST を生成し，(1) の規則違反のパタンを作成したときと同様に，行数や変数名をたよりに，S-AST から該当箇所を探すと，**図 7** に示す付近が対応することが分かった．

(2) の違反を検出するためにパタンに持たせる情報は，比較演算子が使われていること，比較対象が jobject 型であることの 2 つである．比較演算では，演算子の両辺の型が一致するように，必要に応じて暗黙のキャストが行われる．そのため，jobject 型であるかどうか検査するのは片方の型だけでよい．しかし，Clang の仕様が分からなければ，キャストされると自信を持つことができないかもしれない．このときは，パタンを 2 つ書けば安心できる．

```
1  {:kind "If"
2   :loc-begin ··· :loc-end ···
3   :condition
4     {:kind "Binop" :op "!=" :type ···
5      :loc-begin ··· :loc-end ···
6      :LHS {:kind "VarRef"
7            :name "js_locale_previous"
8            :type [{:kind "Typedef-type"
9                    :typedefname "jobject"
10                   ···}]
11           ···}
12     :RHS {:kind "varRef" :name "js_locale"
13           :type [{:kind "Typedef-type"
14                   :typename "jobject"
15                   ···}]
16           ···}}
17  ···}
```

図 7　規則違反 (2) に該当する部分の S-AST
Fig. 7　S-AST corresponding to the part containing the rule (2) violation.

図 7 からこれらの情報だけを残して，パタンは次のように作れる．

```
1  {:kind "Binop" :op "!="
2   :LHS {:type
3         [...
4          {:typedefname "jobject"}]}
```

6.1.3　jobject 型の値から他の型へのキャスト

文献 [16] によると**図 8** のプログラムに (3) の規則違反が見つかったとある．図 8 は，Android のフレームワークライブラリ中の frameworks/base/core/jni/android_util_Binder.cpp（1717 行のソースコード）の 967 行目から 994 行目にある関数定義である．図 8 の 4 行目で行っている linkToDeath 関数の呼び出しの第 2 引数は jobject 型だが，8 行目にある linkToDeath 関数の定義を見ると，第 2 引数の型は void ポインタになっている．そのため，暗黙に型キャストが起こっており，(3) の規則に違反している．

ASTgrep を使って android_util_Binder.cpp から S-AST を生成すると，4,906,784 バイトのファイルが生成された．生成した S-AST から，もとのプログラムの行数や linkToDeath という関数を呼び出していることを使って絞り込むと，(3) の規則違反は，**図 9** に示す付近に存在す

```
1  static void android_os_BinderProxy_linkToDeath(JNIEnv* env, jobject obj,
2                                                  jobject recipient, jint flags) {
3    ...
4    status_t err = target->linkToDeath(jdr, recipient, flags);
5    ...
6  }
7
8  status_t BpBinder::linkToDeath(
9      const sp<DeathRecipient>& recipient, void* cookie, uint32_t flags) {
10   ...
11   ob.cookie = cookie;
12   ...
13 }
```

図 8　JNI 規則違反 (3) を含んだソースコード（文献 [16] に掲載）
Fig. 8　Source code containing JNI rule violation (3) (reported in Ref. [16]).

```
1  {:kind "Var" :name "err" ···
2   :loc-begin ["···util_Binder.cpp" 986 9]
3   :loc-end   ["···util_Binder.cpp" 986 65]
4   :init
5    {:kind "FuncCall" ···
6     :loc-begin ··· :loc-end ···
7     :Func [{··· :name "linkToDeath"}]
8     :parm [···
9      {:kind "VarRef" :name "recipient"
10      :type [
11       {:kind "Typedef-type"
12        :typedefname "jobject"
13        :displaytype "jobject"
14        ···}
15       {:kind "Pointer-type"
16        :Pointee {:kind "Void-type"}
17        :displaytype "void *"}]
18      ···}
19     ···]}}
```

図 9　図 8 を変換した S-AST
Fig. 9　S-AST generated from the program shown in Fig. 8.

ることが分かった．さらに，図 8 の 4 行目の第 2 引数を見れば，問題となっている変数が `recipient` であると分かる．それをたよりに，今度は図 9 で実引数リスト中の `recipient` を探すと，問題があるのは 9 行目から 18 行目であると分かる．この**:type** 属性中の**:displaytype** 属性を見ると，

`[{jobject 型}  {void *型}]`

となっており，5.2.4 項で述べたとおりの形式で，`jobject` 型から `void *`型へのキャストを表している．

(3) の規則違反を検出するためには，`jobject` 型の値が他の型へキャストされていることがわかればよい．これを検出するパタンは，**:type** 属性のベクタの先頭の要素が `jobject` 型を表しており，要素を 2 つ以上持っているパタンである．図 9 の 9 行目から 18 行目から必要な情報だけ残すと，パタンは

```
1  {:type [
2   {:typedefname "jobject"} _ ...]}
```

と書ける．ここで `_`（アンダースコア）は変数である．

このパタンを用いて android_util_Binder.cpp に対してパタンマッチを行った結果，ASTgrep では 19 カ所検出した．それぞれの箇所のソースコードを調査すると，問題がありそうな箇所は 7 カ所であり，残りは `jobject` 型から，Java の Object クラスのサブクラスである配列型への参照を表す `jarray` 型への明示的なキャストなど，JNI で許されているキャストであった．このパタンでは，キャスト先の型を指定していなかったために，JNI で許されているキャストまで検出する結果になった．

これらのケーススタディを通して，ASTgrep が生成した S-AST から容易にパタンを作成できることが示せた．ここで作成したパタンは，少なくとも文献 [16] にソースコード中の具体的な違反箇所が掲載された箇所を検出でき，規則違反 (3) については，文献 [16] にソースコード中の具体的な違反箇所が記載されていない規則違反も検出できた．

6.2　セキュアコーディング

ここでは，JPCERT で紹介されているセキュアコーディング規約のうち式（EXP）に関するルール[*8]を対象に，ASTgrepで検出できるか（パタンが書けるか）どうか検討した．結果を**表 1** に示す．ASTgrep が適していると考えられるものは○，検出できなくはないが ASTgrep が適していないと考えられるものは△，ASTgrep で検出することが難しいと考えられるものは×とする．EXP38-C は，日本語で書かれた規約の記述を定式化することが難しく，検討することができなかった．

ASTgrep が適していると考えられるルールについては，実際にパタンを作り，検出できることを確認した．作成したパタンは，ルールによっては偽陽性が出る可能性があるが，十分実用的であった．作成したパタンは付録に示す．

EXP32-C と EXP40-C は型のキャストに関する規則である．ASTgrep の S-AST は各ノードが型を持っており，これが応用範囲を広げる結果になった．型は，広範囲のプログラムを解析した結果を集約したものであるため，規則違反の検出に利用しやすいということは納得できる．型だけでなく，たとえば，各変数ノードに到達可能な定義を持たせれば，データフローに依存するような検出も可能になるのではないかと考える．

[*8] レコメンデーションは対象にしていない．

表 1　CERT のセキュアコーディング規約
Table 1　CERT secure coding standards.

規約番号	内容	結果
EXP30-C	副作用の評価順序に依存しない	△
EXP31-C	assert() の中では副作用は避ける	○
EXP32-C	非 volatile 参照により volatile オブジェクトにアクセスしない	○*
EXP33-C	初期化されていないメモリからの読み込みを行わない	×
EXP34-C	null ポインタを参照しない	×
EXP35-C	一時的な生存期間を持つオブジェクトを変更しない	○
EXP36-C	ポインタをより厳密にアラインされるポインタ型に変換しない	○
EXP37-C	正しい引数の数と型で関数を呼び出す	△
EXP38-C	ビットフィールドメンバや無効な型で offsetof() を呼び出さない	-
EXP39-C	適合しない型のポインタを使って変数にアクセスしない	△
EXP40-C	定数（const）オブジェクトを変更しない	○*
EXP44-C	sizeof 演算子のオペランドは副作用を持たせない	○
EXP45-C	選択文に対して代入を行わない	○

* パタンの記述に`:when` パタンを用いている．

EXP30-C，EXP31-C，EXP35-C，EXP44-C は副作用の判定をともなう．C/C++言語では，関数が副作用をともなう関数かどうかは型などからは分からない．そのため，保守的に検出するか，検出漏れをあきらめるしかない．

EXP30-C，EXP37-C，EXP39-C は単純なパタンでは記述できない．しかし，Clojure のプログラムを駆使すれば検出できなくはないと考えられる．たとえば，EXP39-C では，パタンマッチを使って型キャストが起こっているところをすべて探し，さらに Clojure のプログラムを使って型キャストが妥当かどうかを調べることで検出することができる．

EXP33-C や EXP34-C はデータフローに依存した検査が必要であるため，ASTgrep では検出することが難しい．

以上より，各ノードに型を持たせる S-AST の設計は有用であったが，データフロー解析や，関数が副作用を持つかどうかの解析など，構文と型だけでは判断できない規則違反は検出が難しいといえる．

6.2.1　パタンマッチの時間

ASTgrep が適していると考えられる規則のパタンファイルを大規模なソースコードに適用し，パタンマッチにかかる時間を計測した．パタンマッチ対象のソースコードには，Apache HTTP サーババージョン 2.2.31 の中で最大の行数のソースファイルである mod_rewrite.c を用いた．mod_rewrite.c は 4,998 行であり，パタンマッチの対象となるプリプロセッサ処理後のプログラムは 17,993 行だった．実験を行った環境は以下のとおりである．

表 2　パタンファイルの大きさとパタンマッチの時間
Table 2　Scales of pattern files and elapsed times for pattern matching.

規約番号	行数	ルール数	`#nest` 数	時間（秒）
EXP31-C	39	13	1	27.1
EXP32-C	12	2	0	7.6
EXP35-C	6	1	1	6.0
EXP36-C	426	48	0	99.0
EXP40-C	16	2	0	8.2
EXP44-C	117	26	1-2	53.5
EXP45-C	142	40	0	79.0

```
1  void f(int a, int b) {
2    while (a = b)
3      ;
4  }
```

図 10　EXP45-C の規則に違反する C++のソースコード
Fig. 10　C++ source code violating the rule EXP45-C.

CPU: Intel Core i7-6700K 4 GHz
OS: Ubuntu Linux 14.04.4 LTS
JVM: OpenJDK (java version 1.7.0_95)

ほとんどのパタンファイルは複数のルールで構成されている．**表 2** にパタンファイルの行数，ルール数，1 ルール中の`#nest` の数，パタンマッチにかかった時間を示す．

各ルールは 10 行に満たないが，ルールの数が多いパタンファイルでは行数が大きくなっている．そのようなパタンファイルではパタンマッチの時間が長くなっているが，それでも最大で 99 秒であり実用的と考えられる．

6.2.2　ASTMatchers との比較

パタンの作成が容易にできることを確認するために，ASTgrep と Clang コンパイラが提供する宣言的な記述で構文木にパタンマッチできるドメイン特化言語である AST-Matchers とで，EXP45-C の規則違反の中から，while 文の条件式が代入式になっているものを検出するパタンを作成した．そのために，まず**図 10** に示す検出したい違反を含む小さなプログラムを作った．

6.2.2.1　ASTMatchers

ASTMatchers はマッチャと呼ばれるマクロを組み合わせて構文木を探索するプログラムを構築する C++の言語内 DSL である．ASTMatchers のパタンは，ドキュメント[*9]に示されているパタン構築の戦略に従って，構文木の根に近い方から少しずつ構築した．その際，少し構築してはマッチするかどうかを確認するという手順を繰り返した．

構文木の形状は Clang コンパイラに`-ast-dump` オプションを渡すことで表示できる．表示された構文木から while 文に該当する箇所を抜き出したものを**図 11** に示す．各ノードにはソースコード上の行数と，ノードオブジェクト

[*9] 「How to create matcher」
http://clang.llvm.org/docs/LibASTMatchers.html

```
‘-WhileStmt
  |-<<<NULL>>>
  |-ImplicitCastExpr ’_Bool’ <IntegralToBoolean>
  | ‘-ImplicitCastExpr’int’ <LValueToRValue>
  |   ‘-BinaryOperator’int’ lvalue ’=’
  |     |-DeclRefExpr’int’ lvalue ParmVar ’a’ ’int’
  |     ‘-IntegerLiteral’int’ 0
  ‘-NullStmt
```

図 11　Clang コンパイラによる構文木のダンプ
Fig. 11　Dumped AST using Clang compiler.

```
whileStmt(
  hasCondition(expr(
    ignoringImpCasts(
      binaryOperator(hasOperatorName("="))))))
```

図 12　while 文の条件部で代入するパタン（ASTMatchers 用）
Fig. 12　Pattern of assignment in condition of while-statement for ASTMatchers.

```
1  {:kind "While" ···
2   :condition
3    {:kind "Binop" :op "=" ···
4     :type [{:kind "Int-type"}
5            {:kind "Bool-type"}]
6     :LHS {:kind "VarRef" :name "a" ···}
7     :RHS {:kind "VarRef" :name "b" ···}}}
```

図 13　ASTgrep で出力した S-AST
Fig. 13　S-AST produced by ASTgrep.

```
1  {:kind "While"
2   :condition
3    {:kind "Binop" :op "="}}
```

図 14　while 文の条件部で代入するパタン（ASTgrep 用）
Fig. 14　Pattern of assignment in condition of while-statement for ASTgrep.

のアドレスも表示されていたが，図 11 では見やすさのために省略している．

この構文木にマッチするように作成したパタンを**図 12** に示す．このパタンは，`whileStmt` マッチャで while 文のノードを探し，`hasCondition(expr(...))` で条件部が式である while 文に限定する．さらに，`binaryOperator(hasOperatorName("="))` で代入式に限定してマッチする．ただし，真偽値型への型キャストや，左辺値から右辺値への型キャストが暗黙に行われるため，それらを無視するために `ignoreImpCasts` マッチャを用いている．

6.2.2.2　ASTgrep

ASTgrep でも，まず構文木である S-AST を生成する．生成した S-AST の while に該当する箇所を**図 13** に示す．このうち，while 文の条件が代入式ということを表す箇所だけを取り出すことで，**図 14** に示すパタンを作成できる．

6.2.2.3　比較

ASTMatchers と ASTgrep を比較する．まず，ASTMatchers では構文木とパタンの形式が異なる．そのため，パタンを作成する際には，対応するマッチャを調べる必要があった．特に，`ignoringImpCasts` はたまたま発見できたのでパタンに用いたが，構文木に直接対応するノードがなく，探す手掛りもない状況だった．それに対して ASTgrep では，図 13 の S-AST の一部を削除するだけでパタンが作れており，手掛かりなくキーワードを探す必要はなかった．したがって，特に慣れていないユーザには ASTgrep の方が簡単にパタンを記述できると考えられる．一方で，`ignoringImpCasts` のようなアドホックなマッチャは，うまく使えば便利であり，ASTMatchers に精通していれば簡潔にパタンを記述できると考えられる．

7.　おわりに

本研究では，与えられた C 言語または C++言語で記述されたプログラムを S 式で表された抽象構文木に変換して，それを対象として bcmacro を用いて規則違反のパタンを検出するツール ASTgrep を作成した．JNI 規則違反を検出するケーススタディを通して，ASTgrep のパタンの作成が容易であることが示せた．また，セキュアコーディング規約の違反が検出できるか検討し，構文木の各ノードに型を持たせることが有効であることが分かった．

謝辞　本研究の一部は，JSPS 科研費 25330080 の助成を受けたものです．

参考文献

[1] Brunel, J., Doligez, D., Hansen, R.R., Lawall, J.L. and Muller, G.: A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking, *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL ’09*, pp.114–126 (2009).

[2] Furr, M. and Foster, J.S.: Polymorphic type inference for the JNI, *Proc. 15th European conference on Programming Languages and Systems*, pp.309–324 (2006).

[3] Gregor, D. and Schupp, S.: STLlint: lifting static checking from languages to libraries, *Software: practice and experience*, Vol.36, No.3, pp.225–256 (2006).

[4] Guntli, C.: Architecture of clang, available from ⟨http://wiki.ifs.hsr.ch/SemProgAnTr/files/Clang_Architecture_-_ChristopherGuntli.pdf⟩ (2011).

[5] Horváth, G. and Pataki, N.: Clang Matchers for Verified Usage of the C++ Standard Template Library, Eötvös Loránd University (2015).

[6] Kondoh, G. and Onodera, T.: Finding Bugs in Java Native Interface Programs, *Proc. 2008 International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pp.109–118 (online), DOI: 10.1145/1390630.1390645 (2008).

[7] Li, S. and Tan, G.: JET: exception checking in the Java native interface, *Proc. 2011 ACM international conference on Object oriented programming systems languages and applications*, pp.345–358 (2011).

[8] Liang, S.: *The Java Native Interface: Programmer’s Guide and Specification*, Addison-Wesley (1997).

[9] Nishiwaki, H., Ugawa, T., Umatani, S., Yasugi, M. and Yuasa, T.: SEAN: Support Tool for Detecting Rule Violations in JNI Coding, *Journal of Information Processing*, Vol.5, No.3, pp.23–28 (2012).
[10] Padioleau, Y., Lawall, J., Hansen, R.R. and Muller, G.: Documenting and Automating Collateral Evolutions in Linux Device Drivers, *Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, *Eurosys '08*, pp.247–260 (2008).
[11] Quinlan, D.J.: ROSE: Compiler Support for Object-Oriented Frameworks, *Parallel Processing Letters*, Vol.10, No.02n03, pp.215–226 (2000).
[12] Quinlan, D.J., Vuduc, R.W. and Misherghi, G.: Techniques for specifying bug patterns, *Proc. 2007 ACM workshop on Parallel and distributed systems* (2007).
[13] Seacord, R.C.: *The CERT(R) C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems*, Addison-Wesley (2014).
[14] Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R. and Kobayashi, H.: Xevolver: An XML-based code translation framework for supporting HPC application migration, *Proc. 21st International Conference on High Performance Computing*, pp.1–11 (2014).
[15] Umatani, S., Ugawa, T. and Yasugi, M.: Design and Implementation of a Java Bytecode Manipulation Library for Clojure, *Journal of Information Processing*, Vol.23, No.5, pp.716–729 (2015).
[16] 西脇春菜，鵜川始陽，馬谷誠二，八杉昌宏，湯淺太一：JNI 規則違反検出ツール SEAN を用いた Android のバグ修正，第 15 回プログラミングおよびプログラミング言語ワークショップ論文集 (PPL2013) (2013).
[17] 平石　拓，李　暁，八杉昌宏，馬谷誠二，湯淺太一：S 式ベース C 言語における変形規則による言語拡張機構，情報処理学会論文誌：プログラミング，Vol.46, No.SIG (PRO 24), pp.40–56 (2005).

付　　録

A.1 セキュアコーディング規約違反のパタンファイル

6.2 節で作成した CERT のセキュアコーディング規約違反を検査するパタンファイルを示す．ASTgrep には選択（choice）パタンがないため，たとえば副作用をともなうすべての演算子について同じような使い方にマッチするルールを作ろうとすると，複数のルールに分けて作る必要がある．

パタンファイルには Clojure のプログラムを記述できるので，Clojure のマクロを用いるとこれらのルールをまとめることができる．しかし，ここではマクロを用いずに個々のルールを書く前提で，その一部だけを示す．

A.1.1 EXP31-C

`assert` 文の引数の中に副作用をともなう式を書いてはならないという規則である．副作用をともなう関数呼び出しの検出は諦め，以下のようなルールを副作用をともなうすべての演算子に対して作成した．

```
1  (defrule exp31-inc
2    {:Func [{:name  "assert"}]
3     :parm [#nest @{:kind "Unop"
4                    :op "++"}]}
5    "EXP31-C violation")
```

A.1.2 EXP32-C

非 volatile 参照により volatile オブジェクトにアクセスしてはならないという規則である．volatile 修飾が外されるような型キャストを検出するように，以下の 2 つのルールを作成した．ASTgrep には否定のパタンがないため，`:when` パタンを使っている．厳密には，3 段以上の任意の段数のポインタにも対応する必要があるが，ASTgrep では表現できない．

```
1  (defrule exp32-ptr
2    @{:type
3      [... {:Pointee {:volatile "true"}}
4       ... {:Pointee
5            (t :when (not (t :volatile)))}
6       ...]}
7    "EXP32-C violation")
8  (defrule exp32-ptrptr
9    @{:type
10     [... {:Pointee {:Pointee
11           (t :when (not (t :volatile)))}}
12      ... {:Pointee {:Pointee
13           {:volatile "true"}}}
14      ... ]}
15   "EXP32-C violation")
```

A.1.3 EXP35-C

配列をメンバに含む構造体や共用体を返す関数を呼び出し，その結果を格納することなくメンバ配列を参照してはいけないという規則である．メンバが別の構造体や共用体の場合，再帰的にそれらの配列メンバへの参照も許されないため，`#nest` パタンを用いている．厳密には，`:struct` キーだけをたどって `FuncCall` ノードを見つける必要があるが，現在の ASTgrep では表現できない．

```
1  (defrule exp35
2    {:kind "Struct"
3     :struct #nest @{:kind "FuncCall"}
4     :structmember
5     {:kind "Field"
6      :type [... {:kind "Array-type"}]}}
7    "EXP35-C violation")
```

A.1.4 EXP36-C

アライメント境界に関する制約が厳しくなる方向へポインタ型をキャストしてはいけないという規則である．たとえば，次のルールは `char *`型から `int *`型へのキャストを検出する．厳密には，キャスト元の値が，キャスト先の型が要求するアラインメントに合致していればキャストしてもよいが，ASTgrep では値を使ったパタンマッチはできないため，値に関係なくキャスト式だけを探している．EXP36-C では，すべてのプリミティブ型に関するこのような型キャストのすべての組合せに対し，1 つずつルールを定義している．

```
(defrule exp36-cs
  @{:type
    [... {:Pointee {:kind "Char-type"}}
     ... {:Pointee {:kind "Short-type"}}
     ...]}
  "EXP36-C violation")
```

A.1.5　EXP40-C

定数オブジェクトを変更してはならないという規則である．const 修飾が外されるような型キャストを検出するように，EXP32-C に類似の以下の 2 つのルールを作成した．

```
(defrule exp40-ptr
  @{:type
    [... {:Pointee {:const "true"}}
     ... {:Pointee
          (t :when (= (t :const) nil))}
     ...]}
  "EXP40-C violation")
(defrule exp40-ptrptr
  @{:type
    [... {:Pointee {:Pointee
          (t :when (= (t :const) nil))}}
     ... {:Pointee {:Pointee
          {:const "true"}}} ...]}
  "EXP40-C violation")
```

A.1.6　EXP44-C

sizeof 演算子の引数中で副作用を起こしてはいけないという規則である．EXP31-C の assert の引数と同様に扱えばよい．ただし，引数が式だけでなく型の場合も対処する必要があり，それぞれ別のルールとして記述している．

```
(defrule exp-44-inc
  {:kind "SizeOf"
   :Argument
   #nest @{:kind "Unop" :op "++"}}
  "EXP44-C violation ")
(defrule exp-44-inc-t
  {:kind "SizeOf"
   :ArgumentType
   #nest {:ArraySizeExpression
          #nest @{:kind "Unop" :op "++"}}}
  "EXP44-C violation")
```

A.1.7　EXP45-C

if 文などの条件部で代入を行ってはならないという規則である．while や for など同種の構文や，&&や||演算子について以下のようなルールを作成した．

```
(defrule exp45-if
  {:kind "If"
   :condition @{:kind "Binop" :op "="}}
  "EXP45-C violation")
```

また，条件部にカンマ演算子や三項演算子 (?:) を書き，その中で代入を行う場合も同様に禁止する規則である．そのため，次のような規則も作成した．

```
(defrule exp45-if-comma
  {:kind "If"
    :condition
    @{:kind "Binop" :op ","
      :RHS {:kind "Binop" :op "="}}}
  "EXP45-C violation")
```


中村 真也　（正会員）

1994 年生．2016 年高知工科大学情報学群卒業．2016 年電気学会・電子情報通信学会・情報処理学会四国支部奨励賞受賞．

鵜川 始陽　（正会員）

1978 年生．2000 年京都大学工学部情報学科卒業．2000 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年同専攻博士後期課程修了．同年京都大学大学院情報学研究科特任助手．2008 年電気通信大学助教．2014 年より高知工科大学准教授．博士（情報学）．プログラミング言語とその処理系に関する研究に従事．情報処理学会 2012 年度山下記念研究賞受賞．

馬谷 誠二　（正会員）

1974 年生．1999 年京都大学工学部情報工学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了．同年同大学院情報学研究科産学官連携研究員．2005 年同研究科助手．2007 年より同研究科助教．博士（情報学）．プログラミング言語，コンパイラ，並列/分散処理に興味を持つ．日本ソフトウェア科学会，ACM 会員．