日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

拡張可能なJavaScript向けコーディング検査器

桑原 寛明　末次 亮　山本 晋一郎　阿草 清滋

本稿では，JavaScript プログラムを対象とする拡張可能なコーディング検査器を提案する．JavaScript は言語の特徴からエラーが紛れ込みやすいがインタプリタ型でありエラーが見つけにくく，コーディング検査器が有用である．提案するコーディング検査器は，検査項目を独自に実装して差し替える仕組み，検査結果の出力形式と出力先を差し替える仕組みを持つ．検査項目の自由な実装にはソースコードに関する様々な情報を利用できる必要があり，そのために JavaScript の細粒度ソフトウェアモデル JSX-model を定義する．既存の検査器が実現している構文的な検査に加えて API の利用方法など意味を考慮した検査も実現できることと，出力先を変更する仕組みを利用した CUI 版，GUI 版，Eclipse Plugin 版の検査器の実装について述べ，拡張可能性を示す．

1　はじめに

Web アプリケーションにおけるユーザインタフェースを実装するためのプログラミング言語として JavaScript が広く利用されている．Web ブラウザは HTML を用いて記述された Web ページの内容と構造を DOM ツリーとして保持しており，JavaScript プログラムを用いて DOM ツリーを操作することでページの内容や構造を変更できる．さらに，リンクのクリックとは無関係に JavaScript プログラム内でサーバと通信することが可能であり，これらの要素によりデスクトップアプリケーションに近い操作性が Web アプリケーションにおいても実現されている．これらの技術を総称して Ajax(Asynchronous JavaScript + XML) と呼ばれている．

リッチなユーザインタフェースを実現するために規模の大きな JavaScript プログラムが作成されているが，プログラムの品質を一定水準以上に保つのは容易ではない．言語の標準化が進んだこととライブラリを利用することで軽減されるとはいえ，処理系ごとの差異を意識してプログラムを作成する必要がある．また，同名の変数や関数を複数定義できる，変数の参照が宣言の前方に出現できる，暗黙の型変換が強力である，といった JavaScript の特徴は簡単な処理の簡潔な記述には有効であるが，堅牢で大規模なプログラムの作成には不向きである．JavaScript はインタプリタ型であるため，コンパイラ型言語であればコンパイラが発見するようなエラーについても実行時エラーとなるまで発見されない可能性がある．

本稿では，JavaScript プログラムを静的に解析して内容を検査し，望ましくない記述や実行時にエラーとなる箇所をプログラムの実行前に発見するコーディング検査器を提案する．提案するコーティング検査器は 3 つの特徴を持つ．1 つ目の特徴として，検査項目を実装する際に検査対象のプログラムに関するあらゆる情報を利用できるように細粒度なソフトウェアモデルを利用する．2 つ目の特徴として，個々の検査項目を部品として実装して差し替えられる仕組み

An Extensible Coding Checker for JavaScript Programs.

Hiroaki Kuwabara, 立命館大学情報理工学部, College of Information Science and Engineering, Ritsumeikan University.

Ryo Suetsugu, 株式会社ネットトレックス, NETREQS Co.,Ltd.

Shinichiro Yamamoto, 愛知県立大学情報科学部, Faculty of Information Science and Technology, Aichi Prefectural University.

Kiyoshi Agusa, 名古屋大学大学院情報科学研究科, Graduate School of Information Science, Nagoya University.

を備えており，検査項目を自由に追加して拡張することができる．3 つ目の特徴として，検査結果の出力形式および出力先も差し替え可能な部品として扱うことで，容易に他のツールと連携したり他のツールへの組み込むことができる．

必要とされる検査の内容は組織やプロジェクトによって異なっており，将来において新しい検査が必要になる可能性もある．また，コーディング検査器の利用方法は，統合開発環境への組み込みやビルドプロセスとの統合など一通りではない．そこで，検査項目や検査結果の出力先などを自由に変更できる柔軟で拡張可能なフレームワークを構成し，それを利用して JavaScript 向けのコーディング検査器を実現する．既存の JavaScript 向けコーディング検査器である JSLint [4] の検査項目と Ajax 技術を利用したアプリケーション（以下，Ajax アプリケーションと呼ぶ）における JavaScript プログラムを対象とする検査項目の実装，およびコーディング検査器の 3 種類のユーザインタフェースの実装を示す．

本稿の構成は以下の通りである．2 章で JavaScript について簡単に述べ，3 章で JavaScript を対象とする細粒度ソフトウェアモデル JSX-model について述べる．4 章でコーディング検査器フレームワークについて，5 章で実装について説明し，6 章でまとめる．

2 JavaScript

JavaScript はオブジェクト指向型のスクリプト言語であり，現在の主要な Web ブラウザ上で動作するため Web アプリケーションにおいてリッチなユーザインタフェースを実現するために広く利用されている．

JavaScript は Web ブラウザが提供する機能を利用して Web ページを操作するスクリプトを記述することが想定された言語である．従来，そのようなスクリプトは複雑にはならないと考えられていたため，JavaScript はプログラムを簡単に記述できて実行をできるだけ止めることなく先に進められるように以下に挙げるような特徴を持っている．

- インタプリタ型である
- 行末のセミコロンを省略できる
- 関数呼び出しにおいて実引数と仮引数の数が異なってもよい
- 同一スコープ内で同名の変数や関数を複数回宣言できる
- 参照と同一スコープ内に宣言がない場合はグローバル変数とみなす
- 強力な暗黙の型変換を行う
- オブジェクトのプロパティ（メンバ）を実行時に追加あるいは削除できる

しかし，このような特徴はエラーの発見と修正を難しくする．例えば，関数呼び出しの実引数の数が仮引数の数よりも少ない場合，対応する実引数がない仮引数には undefined 値が代入されて関数が実行される．関数の実行中に undefined 値が参照されることでエラーが発生する可能性がある．この時，修正すべき間違いが実引数の書き忘れであっても発生する現象は undefined 値の参照によるエラーであり，エラーの原因と発生箇所が遠く離れることになりやすい．インタプリタ型であるためこのようなエラーを実行前に発見することも難しい．

JavaScript 向けのコーディング規約として Mozilla や Google が公開しているスタイルガイド [9] [12] がある．空白の挿入や改行の位置などの記述スタイルや，使用を避けるべき構文要素などがまとめられている．JavaScript プログラムがよい習慣とスタイルに従って記述されているか検査するツールとして JSLint や Closure Compiler [7] がある．JSLint はエラーの原因になりやすい構文要素が使用されていないか調べる．Closure Compiler は入力されたプログラムにサイズ縮小と最適化を適用するコンパイラであるが，配列の範囲外アクセスなどの危険なコードの検出も行う．しかし，JSLint や Closure Compiler は Web ブラウザが提供するオブジェクトの利用方法を検査することはできない．また，新しい検査項目を簡単に追加する仕組みを持っていない．

3 JSX-model

JavaScript を対象とする細粒度なソフトウェアモデルとして JSX-model を定義する．JavaScript には様々な亜種が存在するが，JSX-model は標準化された

ECMAScript [5] を対象としている．JSX-model では JavaScript プログラムを 14 種類の構成要素（うち終端要素が 7 種類）および要素間の構成関連と宣言参照関連によってモデル化する．JSX-model の設計にあたっては細粒度リポジトリに基づく CASE ツール・プラットフォーム Sapid [6] における Java 言語モデル JX-model [13] や C 言語モデル CX-model [2] を参考にしている．

JSX-model は XML 文書による細粒度ソフトウェアリポジトリ XSDML [13] であり，JavaScript プログラムに対してタグ付けを行った XML 文書として扱われる．プログラムにタグ付けを行うため，XML のタグをすべて除去すれば元のプログラムを復元できる．構文要素は XML の要素として表現され，構成関連は XML 要素の親子関係によって表現される．同一ファイル内に存在する宣言参照関連は要素の属性として ID を付加することで表現される．XML を利用する利点として，要素の親子関係によって構成関連を自然に表現できること，プレーンテキストで表現されるため人間が読むことや既存のテキスト処理ツールの利用が容易であること，XML 関連技術が整備されており様々なプログラミング言語において実装されていること，などが挙げられる [13] [2] [8] [3].

JSX-model の要素を表 1 に示す．表の上側が非終端要素，下側が終端要素である．非終端要素はプログラムの構文情報を表しており，他の要素の組み合わせを子ノードとして持つ．終端要素はプログラムの字句情報を表しており，テキストノードのみを子ノードとして持つ．JSX-model は汎用的なソフトウェアリポジトリを指向しており，プログラムの意味に影響しないコメントや空白も含めてすべての字句情報を内部に持っている．そのため，例えば「演算子の前後に空白が存在するか」のような空白などに着目する検査も行われるコーディング検査にも適している．

一部の要素ではより細かい種別を sort 属性に記述する．例えば，if 文と for 文はいずれも Stmt 要素でタグ付けされるが，sort 属性の値をそれぞれ If および For とすることで区別する．式を表す Expr 要素においても sort 属性の値は演算によって異なる．要素と属性を組み合わせることで構成される木構造が過度に複雑にならず，どのような構文要素か判定するために字句要素を調べる手間も回避している．

表 1　JSX-model の要素

要素名	JavaScript の構成要素
File	プログラムファイル全体
VarDec	変数の宣言（var 文）
FunDec	関数の定義（function 文）
Stmt	var および function 以外の文
Expr	式
Param	仮引数
ObjProp	オブジェクトリテラルのプロパティ
ident	識別子
literal	リテラル
kw	予約語
op	演算子
comment	コメント
sp	空白文字
nl	改行文字

```
function f() {
  var a;
  a = 1;
}
```

図 1　JavaScript プログラムの例

宣言参照関連を表現するために，識別子を表す ident 要素はその宣言へのリンクを表す属性を持つ．宣言に対応する要素は要素を一意に特定するための ID を属性として持っており，その値をリンク先として利用する．JavaScript では，ある参照について対応する宣言が存在しない場合や複数存在する場合があるため，対応する宣言すべてを列挙して宣言参照関連とする．

JSX-model の例として，図 1 の JavaScript プログラムから得られる JSX-model の XML を図 2 に示す．図 2 の XML は読みやすくするために改行やインデントを挿入して整形している．プログラムの構成要素の種類がタグ名と sort 属性の値，構成関連が要素の親子関係，宣言参照関連が id 属性，refid 属性に反映されていることがわかる．

JSX-model は XML 文書であり DOM などの XML 関連技術を用いて情報を取り出すことができるため，

```
<File id="0">
 <FunDec id="2" sort="Named">
  <kw>function</kw>
  <sp> </sp>
  <ident id="1">f</ident>
  <op>(</op>
  <op>)</op>
  <sp> </sp>
  <Stmt id="4" sort="Block">
   <op>{</op>
   <nl line="1" coffset="15" offset="15">
</nl>
   <sp>    </sp>
   <VarDec id="5">
    <kw>var</kw>
    <sp> </sp>
    <ident id="3" refid="3">a</ident>
    <op>;</op>
   </VarDec>
   <nl line="2" coffset="11" offset="11">
</nl>
   <sp>    </sp>
   <Stmt id="6" sort="Expr">
    <Expr id="7" sort="Assign">
     <Expr id="8" sort="VarRef">
      <ident id="9" refid="3">a</ident>
     </Expr>
     <sp> </sp>
     <op>=</op>
     <sp> </sp>
     <Expr id="10" sort="Literal">
      <literal sort="decimal">1</literal>
     </Expr>
    </Expr>
    <op>;</op>
   </Stmt>
   <nl line="3" coffset="11" offset="11">
</nl>
   <op>}</op>
  </Stmt>
 </FunDec>
 <nl line="4" coffset="2" offset="2">
</nl>
</File>
```

図 2 図 1 の JSX-model

JSX-model を利用するツールの実装には様々なプログラミング言語が利用できる．しかし，JSX-model に存在する要素，各要素間に存在する構成関連，各要素が持つ属性は決まっているため，構成関連をたどる API や属性ごとに値を取得する API があると便利である．そこで，XML 文書を操作する API をスキーマ定義から自動生成する Relaxer [1] と，構成関連をたどる API およびオフセットや長さ，開始行などの位置情報を取り出す API を Relaxer が生成した API に対して追加するツール [10] を利用して，JSX-model にアクセスする API を Java 言語を用いて提供する．API を提供するプログラミング言語の選択は難しい問題だが，今回は既存のツールを利用して自動生成可能であり低コストで提供できることから Java 言語を選択した．

4 コーディング検査器フレームワーク

コーディング検査において必要な検査は，組織やプロジェクト，採用したコーディング規約などによって異なる．そのため，必要な検査を過不足なく行うには既存のコーディング検査器を利用するだけでは不十分であり，不足分を補うための検査器を用意しなければならないことが多い．コーディング検査器は，検査対象の JavaScript プログラムの字句，構文，意味を解析して検査を行い結果を出力する．大半の検査器では字句，構文，意味の解析と検査ルーチンおよびユーザインタフェースが強く結合しているため，不足する検査を追加することは難しい．そこで，字句，構文，意味の解析といった基本的な解析と検査ルーチンおよびユーザインタフェースを分離し，検査ルーチンとユーザインタフェースを差し替えられるコーディング検査器フレームワークを提案する．基本的な解析には JSX-model を利用する．

4.1 構成

提案するコーディング検査器フレームワークの概略を図 3 に示す．図中の検査項目は検査ルーチンを実現した部品を表し，出力系はユーザインタフェースを表す．フレームワークは複数の検査対象それぞれを複数の検査項目について検査して結果を内部に蓄積する．そして，検査結果を検査結果整形器によって特定の出力形式に変換し，変換結果を出力系によって特定の出力先に出力する．検査対象群，検査項目群，検査結果整形器および出力系はすべて外部から設定する．このように検査，整形，出力を分離して差し替え可能な部品とみなす構造とすることで柔軟で拡張可能なコーディング検査器を実現する．

検査対象の JavaScript プログラムは JSX-model

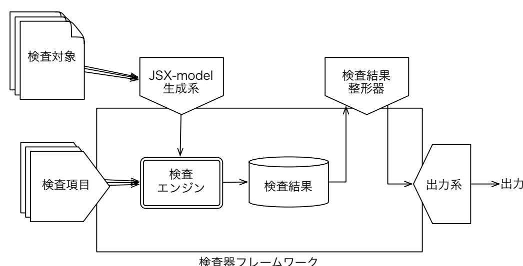

図 3　コーディング検査器フレームワークの概略

によって表現される．JSX-model への変換を行うタイミングは実装によって自由に決定できる．例えば，図 3 のように検査を行う際にフレームワークが JSX-model を取り出すタイミングで変換することも，事前にすべての検査対象を変換しておいてその結果を利用することも可能である．統合開発環境と連携する場合は前者が適しているが，一度生成された JSX-model はファイルに出力しておけば再利用できるためバッチ処理する場合は後者が適している．

各検査項目における検査の手続きは JSX-model を操作するプログラムとして記述される．検査対象の JSX-model は検査エンジンが検査項目を呼び出す時に渡される．検査によって発見された違反は検査エンジンが回収して蓄積する．検査を行うプログラムを作成しフレームワークに差し込めば検査項目を追加できる．プログラムさえ作成できればどのような検査でも行うことが可能である．

発見された違反は何らかの表現形式に整形されてから指定された出力先に出力される．出力先として標準出力やファイル，各種 GUI 部品が考えられるが，自由に設定できる．また，それぞれの出力先に対して検査結果を適切な形式で出力できるように検査結果の整形器が指定できる．必要とされる出力形式や出力先も組織やプロジェクトによって様々なので自由に変更できる．これにより，コーディング検査器を単独のアプリケーションとして利用することや，Eclipse などの統合開発環境に組み込んで利用することが可能となる．

本フレームワークは Java 言語で実装されており，検査項目や出力系などフレームワークに差し込む部品を除く規模は約 2,000 行である．検査項目や出力系の実装にも Java 言語を利用する．

4.2　検査項目の実装方法

検査項目は Java 言語のクラスとして実装される．実装の例を以下に示す．

```
public class SomeRule implements Rule {
  public Violation[] check(JsxFile root) {
    ...
  }
}
```

Rule インタフェースが検査項目を表すインタフェースであり，検査の実行時に呼び出される check メソッドを実装する．check メソッドの引数には JsxFile クラスのオブジェクトが渡される．JsxFile クラスはスキーマ定義から自動生成された API であり，JSX-model のルート要素である File 要素に対応しているため，check メソッドには検査対象の JavaScript プログラム全体が渡されることになる．引数に渡されたオブジェクトを起点に自動生成された API を利用して検査アルゴリズムを実現する．check メソッドは違反を表す Violation クラスの配列を返す．複数の違反を検出する可能性があるため返り値は配列になっている．Violation クラスには検出された違反の位置情報が格納される．

4.3　出力形式と出力先の設定

検査の結果として，検査対象および検査項目と検出された違反の列の組が蓄積される．この検査結果は特定の形式に整形されてから出力されるが，そのための整形器と出力先も Java 言語のクラスとして実装される．

整形器のクラスは以下に示すように整形器を表す ReportGenerator インタフェースを実装する．

```
public class SomeReportGenerator
  implements ReportGenerator {
  public Report generate(Result[] results) {
    ...
  }
}
```

整形時には generate メソッドが呼び出される．generate メソッドの引数は検査対象，検査項目，検出された違反の組を持つ Result クラスの配列であり，配

列にはすべての検査対象に対してすべての検査項目を適用した結果が格納されている．Result クラスから検査対象のファイル名や検査項目の名前，違反の内容，違反の位置などを取り出して任意の形式に整形し，その結果を Report クラスのオブジェクトに格納して返す．

出力先を表すクラスは ReportViewer インタフェースを実装する．

```
public class SomeReporter
  implements ReportViewer {
  public void output(Report report) {
    ...
  }
}
```

出力する際に呼び出される output メソッドには整形済みの検査結果を格納した Report クラスのオブジェクトが渡されるので，検査結果を取り出して標準出力やファイルなど任意の出力先に出力する．

5 検査項目とユーザインタフェースの実装

5.1 検査項目

JSX-model は JavaScript プログラムに記述されている情報のすべてを持っているため，プログラムに関する様々な内容が検査できる．このことを確認するために，初めに JSLint の検査項目が実現できるか確認した．さらに，Ajax アプリケーションにおける JavaScript プログラムを対象とする独自の検査項目を実装した．

5.1.1 JSLint の検査項目

JSLint の 21 個の検査項目のうち 17 個を実装した．これらの検査はプログラムを局所的に調べれば違反かどうか判定できるものであり，構成関連を 1 度から 2 度たどれば十分であった．また，JSX-model の sort 属性の値が有用であり，字句情報にアクセスする必要はほとんどなかった．

実装しなかった 4 個の検査項目のうち 1 個は検査の詳細が不明であった．残りの 3 個は制御フローに関する情報が必要であり，個々に実装するのではなく制御フローに関する汎用的なライブラリを用意してそれを用いるべきであると判断して実装していない．JavaScript プログラムの制御フロー解析については現在研究を進めている．

5.1.2 独自の検査項目

Ajax アプリケーションにおいて Web ブラウザ上で実行される JavaScript プログラムを対象とする検査項目を 28 個実装した．これらの検査項目は任意の JavaScript プログラムに共通な検査ではなく，Web ブラウザ上で実行されることを考慮した検査を行う．XMLHttpRequest に関する検査項目，DOM やスタイルの操作に関する検査項目，オブジェクトプロパティの互換性に関する検査項目などが含まれる．より具体的には，XMLHttpRequest オブジェクトの open メソッドの呼び出しにおいて引数が 2 つ以上渡されているか，open メソッドと send メソッドの引数が整合しているか，DOM 関連のオブジェクトに対して標準化されていないプロパティにアクセスしていないか，主要な Web ブラウザで実装されていないプロパティを参照していないか，といった検査を行う．

5.2 ユーザインタフェース

コーディング検査器フレームワークが持つ出力形式と出力先を自由に設定できる仕組みを利用することで，ユーザインタフェースに関わる部分の実装のみで CUI，GUI，Eclipse Plugin といったユーザインタフェースの異なるコーディング検査器を実現できる．

出力形式として，標準出力に出力して利用者に提示することを想定したプレーンテキスト形式と，他のツールで利用することを想定した XML 形式を定義して整形器を実装した．出力先は，CUI 版では標準出力とファイル，GUI 版では Swing のツリーウィジェット，Eclipse Plugin 版ではマーカーを選択して実装した．プレーンテキスト形式や XML 形式で渡される検査結果を標準出力やファイルに出力したり，ウィジェットや Eclipse の作法に従って検査結果を表示する．

実装したコーディング検査器の GUI 版を図 4 に，Eclipse Plugin 版を図 5 に示す．GUI 版と Eclipse Plugin 版では検査対象の選択方法，検査項目の選択方法，検査結果の表示方法のいずれも異なっているが，内部では同じコーディング検査器フレームワークを利用している．

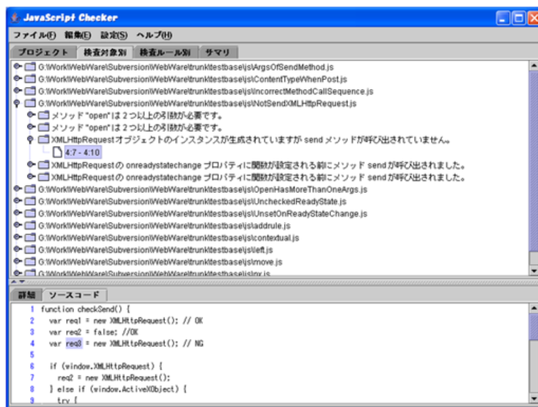

図 4　GUI 版のコーディング検査器

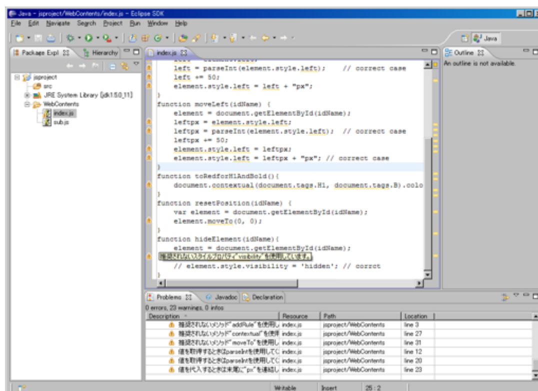

図 5　Eclipse Plugin 版のコーディング検査器

5.3　考察

JSLint の検査項目と Ajax アプリケーションにおける JavaScript プログラム向けの検査項目を実装し，フレームワークに差し込んでコーディング検査ができることを確認した．実装した検査項目は，特定の構文や演算子の使用を検出するもの，関数呼び出しの引数の数や値を検査するもの，オブジェクトのプロパティ名を検査するものなど多岐に渡っている．また，フレームワークを利用することで，検査対象と検査項目の選択および検査結果の表示を行うユーザインタフェースを実装すればコーディング検査器を実現でき，実際に CUI 版，GUI 版，Eclipse Plugin 版の 3 種類の検査器を実装した．これらのことから，提案するコーディング検査器フレームワークは検査項目の実装とユーザインタフェースの実装に関して柔軟で拡張性を持っているといえる．

しかし，特に検査項目の実装の容易さに関して課題が残る．JSX-model は直観的には抽象構文木と記号表の組み合わせであり，簡単に取り出せるのはプログラムに関する基本的な情報のみに限られる．検査の実現に制御フローやデータフロー，変数や関数の型といった情報が必要な場合はそれを計算しなければならずコストがかかる．フローや型の解析は構文解析や意味解析と同じ基本的な解析であるとみなし，制御フロー，データフロー，型解析などのライブラリを提供する必要がある．

今回の実装では検査項目をプログラムとして記述しているが，検査項目の記述に XPath などの XML 関連技術を積極的に活用できる仕組みが必要である．XPath を利用するコーディング検査器としては C 言語プログラムを対象とする CX-Checker [11] がある．組み込みソフトウェア向けのコーディング規約 MISRA-C の約 4 割の検査項目が XPath を用いて記述できることが示されており，コーディング検査において XPath の利用は有効であると考えられる．XPath による記述の具体例として，空白文字としてのタブ文字の使用を検出する XPath 記述は

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

であり，プログラムによる実装に比べて非常に簡潔である．

6　おわりに

本稿では，JavaScript プログラムを対象とする拡張可能なコーディング検査器を提案した．検査項目の追加や変更，検査結果の出力先の変更などを自由に行うことができるフレームワークを構成し，そのフレームワークに複数の検査項目とユーザインタフェースを追加することでコーディング検査器を実現した．

新しい検査項目を作成して追加できることが本コーディング検査器の特徴の一つであるが，検査の内容によって検査対象のプログラムに関する様々な情報が必要とされる．そこで，JavaScript を対象とする細粒度ソフトウェアモデルである JSX-model を定義した．JSX-model を利用することで JavaScript プログ

ラムの構造，識別子，参照などの情報を容易に取り出すことができる．既存の JavaScript 向けコーディング検査器である JSLint の検査項目と，Ajax アプリケーションにおける JavaScript プログラム向けの検査項目を実際に実装し，既存のコーディング検査器が提供する検査項目を含む様々な種類の検査項目が実装できることを確認した．

JSX-model に関する今後の課題として，フローや型に関する情報を提供するライブラリを用意することが挙げられる．未到達コードの検出や，変数の宣言が参照に先行することの検査などには制御フローの情報が必要である．また，JavaScript は型を記述しないプログラミング言語であるが，変数の型，つまり変数に代入される可能性のある値の型の情報は精度のよい検査を行うために必要である．

コーディング検査器に関する今後の課題として，さらに多くの検査項目を実装して検査項目を記述する能力と記述性を評価すること，XPath による検査項目の記述を可能にすることが挙げられる．また，Ajax アプリケーションでは複数のファイルに分けて記述された JavaScript プログラムが一体となって動作するため，複数のファイルを横断的に検査することが必要である．

コーディング検査器フレームワークは検査対象のプログラミング言語に独立な構成であるため，JavaScript 以外のプログラミング言語向けの検査器をフレームワーク上に実現することも今後の課題である．Web アプリケーションを初めとして複数のプログラミング言語を用いて開発されるソフトウェアも珍しくないため，複数のプログラミング言語を同時に扱うことができるコーディング検査器も考えられる．

本稿で述べた JavaScript プログラムから JSX-model への変換系は Sapid の Web サイト†1，コーディング検査器は文部科学省リーディングプロジェクト e-Society 基盤ソフトウェアの統合開発「高信頼 WebWare 生成技術」の Web サイト†2 にて公開している．

謝辞 本ツールの大部分は文部科学省リーディングプロジェクト e-Society 基盤ソフトウェアの統合開発「高信頼 WebWare 生成技術」において開発された．開発にあたり様々なご意見を頂いたプロジェクト関係者の皆様に深く感謝の意を表す．

参考文献

[1] 浅海智晴: Relaxer, http://relaxer.jp/.

[2] 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, Vol. 2005-SE-149, 2005, pp. 57–64.

[3] Badros, G. J.: JavaML: A Markup Language for Java Source Code, *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, Elsevier, 2000, pp. 159–177.

[4] Crockford, D.: JSLint, http://www.jslint.com/.

[5] Ecma International: ECMAScript Language Specification, http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[6] 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, Vol. 39, No. 6(1998), pp. 1990–1998.

[7] Google Inc: Closure Compiler, http://code.google.com/p/closure-compiler/.

[8] 川島勇人, 権藤克彦: XML を用いた ANSI C のための CASE ツールプラットフォーム, コンピュータソフトウェア, Vol. 19, No. 6(2002), pp. 21–34.

[9] Mozilla: JavaScript Style Guide, https://developer.mozilla.org/ja/JavaScript_style_guide.

[10] 新美健一, 山本晋一郎, 阿草清滋: 抽象ソフトウェアエレメントによる CASE ツール開発のためのフレームワーク, 信学技報 *(SS)*, Vol. 103, No. 582(2004), pp. 19–24.

[11] 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*, 近代科学社, 2009, pp. 119–126.

[12] Whyte, A., Jervis, B., Pupius, D., Arvidsson, E., Schneider, F., and Walker, R.: Google JavaScript Style Guide, http://code.google.com/p/google-styleguide/.

[13] 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(2003), pp. 1509–1516.

†1 http://www.sapid.org/index-ja.html

†2 http://www.agusa.i.is.nagoya-u.ac.jp/research/webware/