Vol.2018-SE-198 No.21
2018/3/9

イベント処理を考慮した正確かつ高速なデータフロー解析

高野 健太[1]　荒堀 喜貴[1]　権藤 克彦[1]

概要：JavaScript のデータフロー解析を難しくする要因に非同期に実行されるイベント処理が存在する。先行研究において非同期処理のモデル化は不正確であり、またイベントの発火順序を考慮したモデル上では解析自体がスケールしないという課題が存在する。本研究ではこのイベント処理の振る舞いを解析した結果を元に、精度を保ちつつ高速にデータフロー解析を行うための 2 つの簡約手法、イベント順序ベース簡約と類似操作ベース簡約を提案する。これらの簡約手法では、条件式の簡易な値解析によるイベント処理の枝刈りと、イベントの発火順序を考慮したデータフローの集約によって、データフロー解析をより正確かつ高速に行えるようにする。そして、これらの提案手法を実際にツールとして実装し、7 種類の小さいケースと 3 種類の実際のライブラリに対するデータフロー解析により評価実験を行い、提案手法を用いないベース手法との比較を行った。この評価の結果、小さいケースでは、7 種類の内の 4 種類のケースにおいてほぼ同様の解析時間でより良い精度の解析を行うことができた。また、ライブラリでは、3 種類の内の 1 種類のケースにおいて解析精度の向上が確認できた。

キーワード：JavaScript、静的解析、データフロー解析、非同期処理

1. 序論

近年、JavaScript はブラウザやサーバーサイド、デスクトップアプリケーションなどで広く使われるようになってきた。一方で、JavaScript のコードは XSS や SQL インジェクションといった脆弱性を抱えることが多く、このようなバグを検出するための解析ツールが必要とされている。

しかし、JavaScript は動的言語であり、プロパティの参照や代入、動的型付けといった特徴により解析が難しくなっている。特に静的解析においては、ファイル入出力処理や通信処理等、実行時のみ分かる情報が存在しないため、プログラムの実際の振る舞いを正確に解析するのは難しい。そして、実際に実行されるか判断できない制御フローを解析することで誤検出が多くなり、制御フローが膨大さで解析がスケールしなくなるという課題が存在する。

JavaScript の解析を難しくする要因の 1 つに非同期処理を実行するイベント処理がある。このイベント処理の特徴として run-to-completion があり、1 つの非同期処理中には他の非同期処理が割り込まず、1 つの非同期処理の実行が完了してから別の非同期処理が実行される。また、複数の非同期処理を実行する場合の実行順序は非決定的であるため、N 個の非同期処理が存在した場合は $N!$ 通りの実行順序が考えられる。そのため、この複数の非同期処理を愚直に解析する場合、実行経路の組合せ爆発によって解析時間が膨大になる。逆に、この非同期処理の実行順序を考慮せず、全て常に発火しうる完全非同期を仮定して解析した場合、実際に存在しない制御フローが多く含まれてしまい、多くの誤検出が発生する [1]。

このイベント処理を解析する既存手法に Madsen ら [2] によって提案された λ_ϵ が存在する。この手法では、従来の Call Graph をイベント処理の listen 文や emit 文を付加することで拡張した Event-based Call Graph を生成し、イベントの発火順序を考慮しながらイベント発火に関連するバグを検出する。しかし、イベントの発火順序を非決定的に解析しているため解析がスケールせず、関数のコンテキストを元に制御フローを集約しているため、データフロー解析において誤検出が発生するという課題が存在する。

本研究の目的は、イベント処理を考慮した上で正確かつ高速にデータフロー解析を行うことである。そして、イベント処理による膨大な実行経路の組み合わせを、イベント処理の振る舞いを解析した結果を用いて簡約することで、データフローをより正確かつ高速に解析できる手法、イベント順序ベース簡約と類似操作ベース簡約を提案する。これらの簡約手法では、イベント処理の振る舞いを解析した上で、その情報を元にイベント処理の制御フローやデータフローの簡約を行い、イベント処理を考慮した上で正確か

[1] 東京工業大学
Tokyo Institute of Technology

つ高速にデータフロー解析を行う。そして、この提案手法を実装したツールを用いて小さいケースや実際のライブラリのデータフロー解析を行った。その結果、解析精度が改善するケースが存在することを確認できた。

本研究の貢献は以下の通りである。

- JavaScript においてイベント処理を考慮した上でデータフロー解析を行う際に、解析が不正確になる、または解析がスケールしなくなる問題を明らかにした。
- 解析が不正確になる、または解析がスケールしなくなる問題を解決するために、イベント処理の振る舞いを解析した結果を用いて簡約を行うことで、データフローをより正確かつ高速に解析できる手法である、イベント順序ベース簡約と類似操作ベース簡約を提案した。
- これらの提案手法を利用できるツールを実装し、実際のプログラムを解析できるようにした。
- 実装したツールを用いて、小さいケースや実際のライブラリを解析し、提案手法を用いずに解析を行うベース手法と、解析精度やパフォーマンスの比較を行った。その結果、いくつかのケースにおいて、ベース手法に比べて精度が改善できることを示した。

1.1 本論文の構成

2 章では、本研究の関連研究について説明する。3 章では、例を用いて本研究の Motivating Example を説明し、4 章では提案手法の説明を行う。5 章では、提案手法を利用するためのツールの構成について説明し、6 章では具体的な実装方法について説明する。7 章では、実装したツールで実際にデータフロー解析を行い評価実験を行う。8 章では、本手法における議論を行う。

2. 関連研究

前章で説明した λ_ϵ [2] を除く、データフロー解析やイベント処理解析の関連研究について説明する。

JavaScript を静的にデータフロー解析する手法として WALA [3] や TAJS [4], SAFE [5] が存在する。しかし、これらの手法ではイベント処理を完全非同期に解析しており、発火していないイベントを含め、常に発火し得る状態で解析するため、解析が不正確になっている。そのため、実際に実行されないイベントの発火順序が多く存在することで解析がスケールせず、実際には存在しない制御フローに沿ったデータフローによる誤検出が多くなっている。

データフロー解析を高速に行う手法として Sparse Analysis [6, 7] が存在する。この手法は、従来の制御フローに沿ってデータフローの値を伝播させていくのではなく、変数の def-use の関係に沿ってデータフローの値を伝播させることにより、高速かつ少ない空間量でデータフローを計算する手法である。この手法では前段階で不正確なポインタ解析、メインで正確なポインタ解析の二段階に分けた解析を行う。しかし、JavaScript では前段階の解析として行う flow-insensitive な解析がスケールしないため、Sparse Analysis をそのまま適用できないという問題がある。

Ko ら [8] はこの Sparse Analysis を拡張し、JavaScript で解析を高速に行える Tuned Analysis を提案した。この手法は、前段階の解析として不正確だが高速にポインタ解析が可能な Field-based Analysis [9] を適用した上で、各データフロー変数の値の範囲を、前段階の解析で得られたデータフローの値のみに絞った上で、メインの flow-sensitive な解析を用いて解析をすることで高速に解析する。

イベント処理を解析する手法として、Loring ら [10] は JavaScript におけるイベント処理を Promise をベースにモデル化した λ_{async} を提案した。このモデルでは、実際の JavaScript のようにイベント処理に紐づくコールバックを複数のキューで管理し、イベント処理の優先順位に基いてコールバックが順番に処理される。また、イベント処理の優先順位は JavaScript を実行する環境に依存するため、優先順位の戦略を環境に応じて選択できるようにしており、実行環境に対するモデルの柔軟性を持たせたモデルとなっている。

ソースコード 1 条件式によるイベント処理の制御例

```
1  // コンストラクタ
2  Stream = function() {
3    this.writable = true;
4  }
5
6  // write関数
7  Stream.prototype.write = function(data) {
8    if(!this.writable) {
9      this.emit('error', err);
10     return false;
11   }
12   this.emit('data', data);
13   return true;
14 }
15
16 var stream = new Stream;
17 stream.write("data");
```

3. Motivating Example

例えば、ソースコード 1 を考える。このプログラムの write 関数では、条件式内の this.writable の値によって”error”イベントもしくは”data”イベントの片方のみが発火する。このイベント処理を Event-based Call Graph [2] を用いて解析する際、制御フロー内に含まれる分岐内のどのイベントが発火したかを判断できないため、全ての分岐に含まれるイベントが実行されると仮定して解析する。例え

ソースコード 2 複数のイベント処理が同時に発火する例

```
1  var x = {v: 1};
2  fs.readFile(..., (er, data) => { // A
3    console.log(x);
4    transporter.sendMail(..., (...)=> { // B
5      x = {v: 2};
6    });
7  });
8  fs.readFile(..., (er, data) => { // C
9    ...
10 });
```

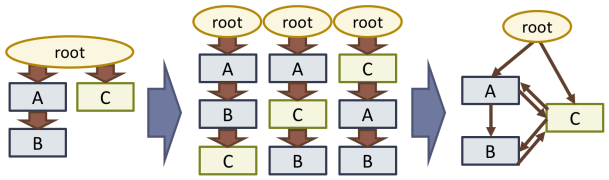

図 1 ソースコード 2 において発火するイベント処理の列 (左) とその発火順序を考慮した際の全ての制御フロー (中)、制御フローを関数単位でマージした際の制御フロー (右)

ば、ソースコード 1 の 17 行目で呼び出した write 関数内でイベントを発火しているが、”error”イベントと”data”イベントのいずれが発火するか判別できず、常に両方発火したと仮定して解析することになり、実際に実行されない非同期処理も含まれ不正確な解析になる。このように、実際のプログラムの実行中には存在しないイベントの発火順序が解析に含まれることで解析が不正確になり、同時に発火順序の組合せの増大により解析時間が膨大になる。そのため、イベント処理をより正確に解析するためには、プログラムの流れをより正確に解析し、実行されるイベント処理を絞り込む必要がある。

また、もう一つの例としてソースコード 2 を考える。この例では、図 1 の左図のように 2 つのファイル読み込み処理のイベントから始まり、外部の処理が完了した順番にそのイベントに紐づくコールバックが実行される。このプログラムをイベントの発火順序を考慮した上でデータフロー解析を行う場合、図 1 の中央の図のように 3 つのイベント A,B,C の発火順序を全て区別して解析すると、発火順序の組み合わせにより解析時間が膨大になるため、Event-based Call Graph のようにコンテキスト等で関数を集約した上で制御フローを集約し解析することになる。その場合、図 1 の右図のように制御フローが集約されるが、B→C→A のように実際のプログラム上では発生しない制御フローが含まれる。この制御フロー上で変数 x のデータフローを解析した場合、5 行目のイベント B のコールバック [*1] 内の変数 x の値が 3 行目の処理 A 内の変数 x に到達するという誤検出が発生する。そのため、正確にデータフロー解析を行う際には、このような発火順序の組合せから生じる不正確な制御フローを排除する必要がある。

[*1] 以降、イベント X のコールバックを「処理 X」と表記

4. 提案手法

4.1 概要

本研究では、JavaScript におけるイベント処理を考慮した上で正確かつ高速にデータフロー解析するための手法を提案する。

今回は、既存手法である Event-based Call Graph [2] をベースとしたデータフロー解析を行う。単純に解析を行う場合、イベントの発火順序を考慮した制御フローを構築した上で Event-based Call Graph を構築し、その上でデータフロー解析を行うことになる。しかし、この解析はイベントの発火順序の組合せが膨大になることでスケールせず、元の制御フローで残っていた正確なデータフローが Event-based Call Graph を構築した時点で失われ、不正確なデータフローになるという課題が存在する。そこで、本研究では解析中にイベント処理を解析して得た情報を元に簡約することで、データフロー解析を正確かつ高速に行うための簡約手法として、イベント順序ベース簡約と類似操作ベース簡約を提案する。

イベント順序ベース簡約は、イベント順序を考慮した制御フローを構築する際に、既存手法のようにイベントに関連する処理のみではなく、その処理の実行に影響する条件文を考慮してイベントの発火順序を解析する。この手法で実際に実行されない非同期処理を枝刈りすることにより、制御フローをより正確かつ高速に解析する。

類似操作ベース簡約は、イベント処理を考慮した制御フロー上でデータフロー解析を行うために簡約する際、既存手法のように制御フローをベースに簡約するのではなく、データフローをベースに簡約を行う手法である。この簡約により、イベント順序を考慮した制御フローにおけるデータフローの情報をより正確に残すことができ、それによりデータフローを正確かつ高速に解析する。

4.2 イベント順序ベース簡約

4.2.1 概要

本手法は、イベントを考慮した制御フローを構築する際に条件式の値解析を行い、分岐内に含まれるイベントに関係する処理の枝刈りを行う。この簡約により、実際に発火しないイベントを枝刈りして発火の候補を絞り込み、イベント処理の制御フローを正確かつ高速に解析する。

4.2.2 簡約の対象

今回は、対象を減らすことで高速に簡約を行うために条件式はイベントに関連する処理の直近の条件式のみに絞り込む。この条件式は手続き間も考慮して選択する。さらに条件式の解析の際も、null や undefined の定数または 2 値の値を持つ変数で構成され、演算子として等号 (==,===)、

不等号 (!=,!==)、AND(&&)、OR(||)、NOT(!) で構成される単純な式のみを考慮して解析する。

4.2.3 簡約の具体的手法

イベント処理の発火順序を考慮して制御フローを計算する際、各変数の定義箇所の追跡を行う。この def-use 情報を用いて条件式内の変数の値解析を行い、条件式が真もしくは偽のどちらか一意に決まる場合はその条件文の一方の節が実行されないと判断し枝刈りを行う。

今回の変数の値追跡は簡易に行うことで高速に解析を行う。まず、イベント順序を考慮した制御フロー上で正確に変数の値解析を行う場合、解析対象のデータフローが膨大になりスケールしないため、条件式で参照する変数の定義箇所は flow-sensitive に追跡した上で、その定義箇所の右辺値は flow-insensitive に解析する。また、複数の条件文の関係を考慮することで、より多くの条件文を枝刈りできるケースが存在するが、解析時間が膨大になるため、今回は各条件式を独立に解析し、他の条件文における節の枝刈りによって値の真偽が一意に決まるケースは考慮しない。

4.2.4 手法の適用例

この手法をソースコード 1 に適用した例を説明する。

まず、このプログラム内でイベント順序ベース簡約の対象となる条件式は、23 行目から呼び出された write 関数内の 8 行目の条件式である。この条件式の値解析として this.writable の定義箇所を追跡すると、3 行目の値”true” が定義されていることが分かるため、条件式の then 節を枝刈りできることから、’data’ イベントのみが発火することが分かる。この結果、従来の解析に比べてより正確かつ高速に解析できるようになる。

4.3 類似操作ベース簡約

4.3.1 概要

Event-based Call Graph 上で計算したデータフローを元にポインタ解析を行う場合、複数の制御フローが集約されることによって実際の実行経路には含まれないような制御フローが含まれてしまうため、データフローが不正確になってしまう。

データフロー解析が不正確になる原因として、イベントの発火順序を考慮した制御フローを集約した Event-based Call Graph 上でデータフローを解析することで、実際には含まれない def-use 関係を抽出することが挙げられる。

そこで本手法では、集約前のイベント順序を考慮した制御フローに含まれる、正確なデータフローの情報を用いて解析することを考える。しかし、この制御フロー上に含まれるデータフローは膨大になるため、解析がスケールしない。そこである変数において、非同期処理の実行順序が異なる場合でも類似した def-use 関係を持つ場合があることに着目し、関数のコンテキストに基づいて変数間の def-use 関係を集約し、解析対象を減らすことでスケールするデータフローを構築する。

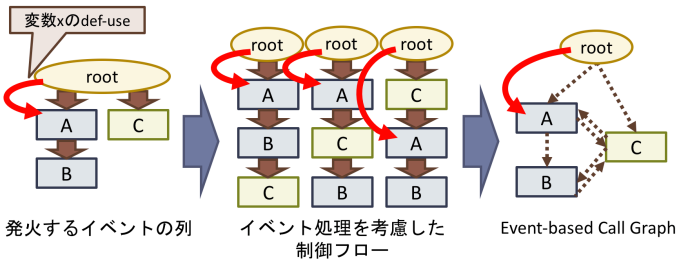

図 2 ソースコード 2 を類似操作ベース簡約を用いて解析する例
(右図の破線は本簡約で集約しない制御フロー)

4.3.2 簡約の具体的手法

まず、イベント処理を考慮した制御フローを構築する際に各変数の def-use 関係を追跡することで、この簡約に必要となる各変数の def-use 関係を計算する。次に、Event-based Call Graph を構築することにより、同じコンテキストを持つ関数を集約するが、従来手法のように制御フローを集約する代わりに、集約前に計算した各変数の def-use 関係を、集約した関数内の対応する変数間の def-use 関係と見なし、各変数の def-use 関係を集約する。

4.3.3 手法の適用例

この手法をソースコード 2 に適用する例を説明する。

まず、図 2 の左図の発火するイベント処理から中央図のイベント処理を考慮した制御フローを構築する際に、各変数間の def-use の関係を計算することで中央図の赤線の関係が得られる。そして、Event-based Call Graph として同じコンテキストの関数を集約し、計算した def-use 関係を、集約した関数内の対応する変数の間の def-use 関係として集約することで図 2 の右図のデータフローを構築する。この例の変数 x では、root と処理 A 内の同じ def-use 関係が 3 つあり、それらが集約された root と処理 A 内の同じ 1 つの def-use 関係として集約される。

この簡約の結果、従来手法で含まれていた制御フロー (B→C→A) に沿った変数 x に関する不正確なデータフローが除去される。これは、変数 x が関係しない処理 C が変数 x のデータフロー上では排除されているため、処理 C の集約 [*2] により生成される不正確な制御フロー (図 2 の右の B→C→A) に沿ったデータフローが生成されないためである。そのため、このデータフロー上で解析することで正確かつ高速にデータフローが解析できる。

5. 手法の構成

5.1 概要

今回の解析手法は以下の 4 つの段階の解析で構成される。

(1) 前段階のポインタ解析

[*2] 図 2 の中央図から右図への簡約において、イベント発火順序の制御フローに含まれる全ての処理 C を 1 つに集約し、それらの処理 C に繋がる制御フロー (B→C, C→A 等) を集約

(2) イベント処理を考慮した制御フローの構築
(3) Event-based Call Graph/データフローの構築
(4) メインのポインタ解析
この章では、4 つの各段階の解析について説明を行う。

5.2 前段階のポインタ解析

本ツールは始めに前段階のポインタ解析を行い、各変数のポインタ集合と従来の Call Graph を構築する。前段階のポインタ解析として flow-insensitive な解析を行う。この際に非同期処理の呼び出しも解析するが、この時の非同期処理の関数は全てイベントの発火元から同期的に呼び出される関数であると仮定して解析する。

5.3 イベント処理を考慮した制御フローの構築

前段階のポインタ解析の結果を元に、イベント処理の発火順序を考慮した制御フローを構築する。この構築は λ_ϵ [2] をベースに行い、非同期処理内や非同期処理間の制御フローを辿りながら解析する。

各非同期処理に対し、始めにイベント順序ベース簡約を適用し、イベントに関連する処理を含む分岐の枝刈りを行う。その後、非同期処理内のリスナの登録やイベント発火の処理を flow-sensitive に追跡し、この非同期処理内で発火するリスナや、新たに登録されるリスナの解析を行う。登録リスナは λ_ϵ と同様に集合で管理しながら解析する。この時、λ_ϵ と異なる点として、process.nextTick() や setTimeout() 等のイベントオブジェクトに紐付かないイベント発火処理については、ユニークオブジェクトに紐づくリスナ登録を行わずにイベント発火のみ行うと仮定し解析する。これは、後の Event-based Call Graph において、関数のコンテキストとして利用する登録リスナの種類を少なくすることで解析対象を少なくし高速にデータフロー解析を行うためである。

この制御フローを構築する際、イベント順序ベース簡約で行う値解析や類似操作ベース簡約に必要となる def-use 関係を計算しながら解析する。

また、この解析における制御フローを単純にするために例外処理を考慮せずに解析を行う。

5.4 Event-based Call Graph/データフローの構築

次に、構築した制御フローを元に Event-based Call Graph を構築し、その上でデータフローを構築する。

Event-based Call Graph の構築では、イベント処理を考慮した制御フロー内の関数を同じコンテキストごとにまとめる。本手法では、登録リスナの集合をコンテキストとして持つ listener-sensitive [2] を関数のコンテキストに利用するが、異なるスコープで発火される同じ非同期処理など、このコンテキストのみでは区別できないケースが存在する。そのため、本手法では listener-sensitive に加え、k-CFA [11] を組み合せることでリスナ間を区別する。

また、データフローの構築は類似操作ベース簡約を適用して行う。

このデータフローグラフを構築する際、今回は解析を単純にするため、関数間の呼び出し関係の解決には、前段階のポインタ解析で構築した Call Graph における呼び出し関係を利用する。

5.5 メインのポインタ解析

前の段階で構築したデータフローグラフを元にポインタ解析を行う。このポインタ解析は Sparse Analysis [6] のメイン解析の手法を用いて行い、データフローグラフ上を伝播させることで高速に解析する。

6. ツールの実装

本ツールは Java を用いて WALA [3] をベースに実装した。ツールの実装行数は約 9000 行である。ツールの構成は、ソースコードから Event-based Call Graph [2] と各変数の def-use 関係を計算するツールとそれらを元にデータフロー解析を行うツールに分かれている。

前者のツールでは WALA による前段階のポインタ解析とイベント順序を考慮した制御フローの構築、Event-based Call Graph の構築を行う。前段階のポインタ解析では 1-CFA をコンテキストにして解析を行い、さらに正確なポインタ解析を行うために Correlation Tracking [12] を適用する。イベント発火を考慮した制御フローの解析の際は listener-sensitive と 1-CFA を組み合せたコンテキストで関数を区別する。

後者のツールでは、前者のツールで計算した Event-based Call Graph と大域変数等の def-use 関係を元に、データフローを構築した上でメインのポインタ解析を行い、最終的なデータフロー解析の結果を得る。

今回実装したツールでは、Node.js の fs, stream 等の標準ライブラリをモデル化した上で解析する。これは、ソケット等の通信処理や外部プロセスの振る舞いを静的解析で考慮しながら解析するのは難しく、かつ標準ライブラリのプログラムが複雑であるため、解析がスケールしなくなるためである。

本ツールの開発実行環境は、2.9GHz Intel Core i5, 8GB メモリの Mac-OSX である。また、JVM で実行する際のメモリ量は 4GB(-Xms4096m -Xmx4096m) を設定している。

7. 評価実験

評価実験では、実装したツールを用いてイベント処理を含むプログラムを解析し、得られるデータフローの正確さや解析のパフォーマンスを評価することで、提案手法の有効性を示す。

本実験では、イベント順序ベース簡約と類似操作ベース

ライブラリ名	バージョン
mkdirp	0.5.1
resolve	1.5.0
stringstream	0.0.5

図 3 評価対象とする Node.js プログラム

簡約の２つの提案手法を用いた解析手法と提案手法を適用せずに解析を行ったベース手法の比較を行う。

7.1 評価対象

本実験では、Node.js のプログラムを対象とし、Node.js のバージョンは v5.12.0 とする。解析対象は、自作した小さいケースと、実際の Node.js のライブラリである。

小さいケースでは、process.nextTick 等の非同期処理呼び出しや EventEmitter によるリスナ登録、イベント発火を利用したプログラムを準備して評価を行う。また、ライブラリを評価する際はライブラリを利用する簡単なプログラム作成し、そのプログラムを解析し評価を行う。

実際の Node.js ライブラリは、簡単に利用したプログラムを記述した上で解析する。実際のライブラリとして対象とするプログラムは図 3 の通りである。

また、Object.keys() 等、WALA でサポートできず、実際の振る舞いや値を解析するのが難しいメソッドが存在するが、これらについては、近い振る舞い、もしくは不明な値を返す処理を定義して対応している。また、util.inherits() 等の prototype メソッドの継承処理は、単純な動的代入では解析が不正確になりスケールしないため、実際のプログラムの振る舞いにおいて継承される prototype メソッドのみを継承するようにコードを書き換えている。

7.2 評価方法

本実験では、WALA の中間表現において、ある変数が汚染されていると仮定した上で、その汚染が伝播する範囲の調査を行い、評価を行う。この時、汚染の伝播は ϕ 関数を含む代入処理や二項演算によって伝播すると仮定する。

評価指標として、解析精度と解析時間を調べる。解析精度は、実際の実行において到達し得ない変数に汚染が伝播されている場合を誤検出、実際の実行において到達し得るのに汚染が伝播されていない場合を検出漏れと見なし、それらの数によって評価を行う。ただし、検出漏れについてはライブラリで評価するのは難しいため、今回は小さいケースのみで評価を行う。解析時間は、前段階のポインタ解析 (前段階解析)、イベント処理を考慮した制御フローの構築 (イベント処理解析)、データフロー解析、Event-based Call Graph の構築 (EBCG の構築)、メインのポインタ解析の各段階に分け、5 回同じ解析を実行した際の平均時間によって評価を行う。

7.3 評価結果

7.3.1 小さいケースの解析

まず、小さいケースを解析結果を示す。図 4 は提案手法、図 5 はベース手法で解析を行った結果である。伝搬変数数における数は、listener-sensitive + 1-CFA のコンテキスト上で異なる関数ごとの、汚染オブジェクトが伝播したデータフロー変数の数の和を示しており、左から局所変数/大域変数/レキシカルスコープ変数/プロパティに対応するデータフロー変数の数を示す。この際の大域変数、レキシカルスコープ変数、プロパティは、代入箇所ごとに区別して数える。

まず、解析精度として誤検出数や検出漏れ数を評価した結果、両手法において検出漏れは確認されず、7 種類のケースの内、4 種類のケースにおいて精度の改善を確認した。小さいケースには、Motivating Example の２つの例に似たケースが含まれており、それぞれイベント順序ベース簡約と類似操作ベース簡約によって、誤検出を除去できた。

また、解析時間を評価した結果、提案手法の解析はベース手法の解析と、ほぼ同等の時間で解析できた。解析時間の中で、特に Event-based Call Graph の構築の時間が増大しているが、これは、類似操作ベース簡約に必要なデータフロー情報である、変数間の def-use 情報を関数のコンテキストを元に集約を行う際に集約対象となる def-use 情報が膨大となり、集約に時間がかかっているためであると考えられる。

小さいケースの評価結果より、提案手法の解析はベース手法の解析に比べ、ほぼ同等の解析時間で、より正確な解析ができていることが分かる。この評価結果は、イベント順序ベース簡約により実際に発火しないイベントが除去され、その上で類似操作ベース簡約によってイベント発火順序を考慮した制御フローから集約した、より正確なデータフローを用いることで誤検出を減らすことができた結果であり、かつ、イベント順序ベース簡約を行う際の条件式の値解析を高速に行うことで、解析のパフォーマンスを落とさずに解析できた結果であると考えられる。

7.3.2 ライブラリの解析

次に、実際のライブラリを各手法で解析した結果を示す。図 6 は提案手法、図 7 はベース手法で解析を行った結果である。

まず、解析精度として誤検出数を評価した結果、提案手法の解析はベース手法の解析に比べ、3 種類のケースの内、1 種類のケースにおいて精度の改善を確認した。

また、解析時間を評価した結果、ベース手法の解析に比べ、提案手法の解析は解析時間が増大しているが、最大で約 1.21 倍の時間内で解析できているため、この結果は許容範囲の結果であると考えられる。

今回精度を改善できた mkdirp のケースでは、ベース手法の解析では、異なる非同期処理間の制御フローが集約さ

	汚染された伝播変数数			5 回の平均解析時間 (単位: 秒)					
ソースコード名	検出数	誤検出数	検出漏れ	合計	前段階解析	イベント処理解析	データフロー解析	EBCG 構築	ポインタ解析
callback-simple.js	3/0/0/2	0/0/0/0	0/0/0/0	9.7096	2.9040	1.5936	4.2434	0.5320	0.4366
callback-series.js	4/0/1/2	0/0/0/0	0/0/0/0	11.3078	3.3656	2.4758	4.4846	0.4990	0.4828
callback-parallel.js	8/0/5/2	0/0/0/0	0/0/0/0	13.2610	3.0220	3.0570	5.5708	0.9888	0.6224
stream-simple.js	2/0/0/1	0/0/0/0	0/0/0/0	15.2218	3.5822	3.1228	6.8240	1.1048	0.5880
callback-multi.js	4/0/1/2	0/0/0/0	0/0/0/0	10.7782	2.9820	1.9924	4.6196	0.6720	0.5122
callback-multi2.js	10/0/2/4	6/0/0/2	0/0/0/0	13.0290	3.7714	2.3652	5.3780	0.8492	0.6652
callback-multi3.js	14/0/3/5	4/0/0/2	0/0/0/0	9.9924	2.9284	1.7454	4.3024	0.6046	0.4116

図 4 小さいケースを提案手法で解析した結果

	汚染された伝播変数数			5 回の平均解析時間 (単位: 秒)					
ソースコード名	検出数	誤検出数	検出漏れ	合計	前段階解析	イベント処理解析	データフロー解析	EBCG 構築	ポインタ解析
callback-simple.js	3/0/0/2	0/0/0/0	0/0/0/0	9.0468	2.8512	1.5076	4.0606	0.1454	0.4820
callback-series.js	4/0/1/2	0/0/0/0	0/0/0/0	10.9618	3.1134	2.0072	5.1914	0.1866	0.4632
callback-parallel.js	8/0/5/2	0/0/0/0	0/0/0/0	11.4202	3.3254	2.1024	5.1446	0.2572	0.5906
stream-simple.js	13/0/1/5	11/0/1/4	0/0/0/0	16.0584	4.4000	3.1464	7.5360	0.3494	0.6266
callback-multi.js	8/0/1/4	4/0/0/2	0/0/0/0	11.6842	3.3584	2.0960	5.4530	0.1522	0.6246
callback-multi2.js	10/0/2/4	6/0/0/2	0/0/0/0	11.7532	3.7390	2.0050	5.2892	0.2092	0.5108
callback-multi3.js	18/0/3/7	8/0/0/4	0/0/0/0	9.5740	2.7414	1.7058	4.4384	0.1866	0.5018

図 5 小さいケースをベース手法で解析した結果

れることで発生する誤検出が存在したが、提案手法の解析では、類似操作ベース簡約で変数間の def-use 関係を計算する際に非同期処理ごとに解析することで、異なる非同期処理間の制御フローの集約を回避できたため、この誤検出を除去できた。

ライブラリにおける評価の結果、最大約 1.21 倍の時間内で、3 ケースの内の 1 ケースについて精度を改善できた。しかし、評価したライブラリは 3 ケースのみであり、他のライブラリに対して提案手法の解析を適用した場合における効果は、現状では不明である。そのため、本ツールを用いてより多くのライブラリ等のプログラムを解析し、提案手法の効果について評価を行うのが今後の課題である。

8. 議論

本実験で解析対象としたライブラリは比較的小さく解析しやすいソースコードから構成されており、今回対象としなかった、より大きなライブラリについては解析のスケーラビリティの問題より、今回の評価対象にできなかった。本ツールにおけるスケーラビリティの改善を行った上で、より大規模かつ複雑なプログラムを解析できるようにし、より広いベンチケースにおいて本手法の有効性を詳細に調べることが今後の課題となる。

本ツールの解析がスケールしなかった原因として、以下の点が考えられる。

- 前段階の解析による flow-insensitive な解析やループ内の不正確な解析により、誤った呼び出し関係が大量に生成されることで、解析対象が膨大になる
- イベントの発火順序の組み合わせが膨大で、解析対象の制御フローが膨大になる

前段階の解析における改善案として、Field-based Analysis [9] や Tuned Analysis [8] の適用や、ループ解析を改善するためのループ展開や Loop-Sensitivity [13] の適用が考えられる。

イベントの発火順序を減らす改善案として、提案手法であるイベント順序ベース簡約の改善が挙げられる。また、λ_{async} [10] のようにイベント処理の優先順位を考慮することや、データフロー等で無関係な非同期処理同士を直列化して解析することで、発火順序の組み合わせを減らし、解析がよりスケールすると考えられる。

また、本ツールの解析が不正確になる原因として、以下の点が考えられる。

- 標準ライブラリのモデル化により、実行時とふるまいが異なる
- 例外処理を考慮しないことで制御フローが不正確になる
- メインのポインタ解析において、不正確である、前段階の flow-insensitive な呼び出し関係を元に関数呼出しを解決している
- 関数のコンテキストが不十分で、制御フローが集約され不正確になる
- 条件文の枝刈りが不十分で、実際に実行されない非同期処理が含まれる

不正確な解析の改善案として、標準ライブラリや例外処理、メインのポインタ解析の呼び出し関係を正確に解析す

	汚染された伝播変数数		5 回の平均解析時間 (単位: 秒)					
ソースコード名	検出数	誤検出数	合計	前段階解析	イベント処理解析	データフロー解析	EBCG 構築	ポインタ解析
resolve	155/0/10/35	0/0/0/0	93.8754	5.9710	22.8476	46.1696	16.8394	2.0478
mkdirp	39/0/8/20	0/0/0/0	43.6288	4.6548	9.5038	25.5854	2.7478	1.1370
stringstream	44/0/1/11	42/0/1/10	33.3762	4.8890	5.9486	19.2236	1.8692	1.4458

図 6 ライブラリを提案手法で解析した結果

	汚染された伝播変数数		5 回の平均解析時間 (単位: 秒)					
ソースコード名	検出数	誤検出数	合計	前段階解析	イベント処理解析	データフロー解析	EBCG 構築	ポインタ解析
resolve	155/0/10/35	0/0/0/0	77.4202	6.6292	25.6460	41.0656	1.7634	2.316
mkdirp	44/0/8/20	5/0/0/0	41.1266	4.7366	9.9422	24.0544	1.1138	1.2796
stringstream	44/0/1/11	42/0/1/10	32.3846	5.3762	8.3440	16.3464	0.9316	1.3864

図 7 ライブラリをベース手法で解析した結果

る、より詳細なコンテキストで関数を区別する、イベント順序ベース簡約を改善する、等が挙げられる。

9. 結論

JavaScript のデータフロー解析を難しくする要因に非同期に処理が行われるイベント処理が存在し、実際に実行されない制御フローが増えることによって解析が不正確になり、スケールしなくなる原因になっている。既存研究のデータフロー解析においては、このイベント処理を不正確に解析することで誤検出が多くなっており、また、イベントの発火順序を考慮したモデル上の解析手法は、非同期処理の実行順序の組み合わせにより解析がスケールしないという課題が存在することを明らかにした。

本研究では、JavaScript におけるイベント処理を考慮した上で高速かつ正確にデータフロー解析を行うための 2 つの簡約手法であるイベント順序ベース簡約と類似操作ベース簡約を提案した。そして、提案手法を用いたツールを実装し、7 種類の小さいケースや 3 種類の実際のライブラリを対象にデータフロー解析を行うことにより、実際に精度が改善するケースが存在することを示せた。

本研究の今後の課題は、本ツールの解析のスケーラビリティを改善することで、より複雑かつ大規模なプログラムに本手法を適用することである。そして、スケーラビリティを改善した上で、より多くのプログラムに本ツールの解析を適用し、提案手法による簡約が有効なケースをより広く見つけることである。

参考文献

[1] Park, J., Lim, I. and Ryu, S.: Battles with false positives in static analysis of JavaScript web applications in the wild, *Proceedings of the 38th International Conference on Software Engineering Companion*, ACM, pp. 61–70 (2016).

[2] Madsen, M., Tip, F. and Lhoták, O.: Static analysis of event-driven Node. js JavaScript applications, *ACM SIGPLAN Notices*, Vol. 50, No. 10, ACM, pp. 505–519 (2015).

[3] Research, I.: T.J. Watson Libraries for Analysis (WALA), http://wala.sf.net Online; accessed November 25, 2017.

[4] Jensen, S. H., Møller, A. and Thiemann, P.: Type Analysis for JavaScript., Springer.

[5] Lee, H., Won, S., Jin, J., Cho, J. and Ryu, S.: SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript, *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Vol. 10 (2012).

[6] Oh, H., Heo, K., Lee, W., Lee, W., Park, D., Kang, J. and Yi, K.: Global sparse analysis framework, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 36, No. 3, p. 8 (2014).

[7] Hardekopf, B. and Lin, C.: Flow-sensitive pointer analysis for millions of lines of code, *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, IEEE, pp. 289–298 (2011).

[8] Ko, Y., Lee, H., Dolby, J. and Ryu, S.: Practically tunable static analysis framework for large-scale JavaScript applications (T), *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, IEEE, pp. 541–551 (2015).

[9] Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J. and Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services, *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, pp. 752–761 (2013).

[10] Loring, M. C., Marron, M. and Leijen, D.: Semantics of asynchronous JavaScript, *ACM SIGPLAN Notices*, Vol. 52, No. 11, ACM, pp. 51–62 (2017).

[11] Sharir, M. and Pnueli, A.: Two approaches to interprocedural data flow analysis (1978).

[12] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M. and Tip, F.: Correlation tracking for points-to analysis of JavaScript, *ECOOP 2012–Object-Oriented Programming*, pp. 435–458 (2012).

[13] Park, C. and Ryu, S.: Scalable and precise static analysis of JavaScript applications via loop-sensitivity, *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015).