特別研究報告

題目

宇宙機搭載ソフトウェアの安全性検査を目的とした データフロー検査支援ツールの開発

指導教員
井上 克郎 教授

報告者
石田 直人

平成 30 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 29 年度 特別研究報告

宇宙機搭載ソフトウェアの安全性検査を目的としたデータフロー検査支援ツールの開発

石田 直人

内容梗概

ソフトウェアの欠陥の中には大きな問題を発生させるものがあり，特に宇宙機搭載ソフトウェアにおける欠陥は人命喪失，多額の経済的損失などの重大な問題を招く．

このような事態にならないために，ソフトウェアの信頼性を向上させる手順がソフトウェア開発作業の中に取り込まれている．その 1 つがコードレビューであり，コードレビューを適切に行うことで欠陥の検出を行うことができる．コードレビューはソフトウェア開発元が自ら行う場合と，ソフトウェア開発元と独立した第三者組織が行う場合がある．後者は IV&V(Independent Verification and Validation) と呼ばれ，コードレビューを客観的な立場で行うことができるという利点がある．

一方で，規模の大きいソフトウェアほど関数や変数の数が多くなり処理が複雑になることからコードレビューのコストも大きくなる．第三者組織がコードレビューを行う場合はさらに困難なものとなるため，コードレビューを行うための補助ツールが求められている．

本研究では，宇宙機搭載ソフトウェアの特徴を備えたソースコードに対して，モジュール間の変数のデータフロー図を生成するツールを開発した．本ツールにより，ソースコードの詳細を知らなくてもより短時間で正確なソフトウェアの安全性検査を可能とすることを目指す．

評価実験では，IV&V を実務とする者に，ツールが出力したモジュール間データフロー図を参照しながらコードレビューを行ってもらった．図がある場合とない場合でレビュー時間に直接的な効果は見られなかったものの，実務者にとってもコードレビューにおいて本ツールが有用であるという意見が多く報告された．

主な用語

静的解析
データフロー
コードレビュー

目 次

1　はじめに

ソフトウェアの欠陥の中には大きな問題を引き起こすものがある．特に自動車や宇宙機に搭載されるソフトウェアの欠陥は人命喪失，ミッション不達成，経済的損失などの重大な問題を発生させ得る．例えば 1996 年に起きたアリアン 5 型ロケットの打ち上げ失敗事故は，搭載されたソフトウェアの 64 ビット浮動小数点値から 16 ビット符号付き整数値に変換する過程で発生したエラーが原因であり [7]，多額の経済的損失をもたらした．

ソフトウェアの信頼性を向上させる手法の 1 つにコードレビューがある．コードレビューを行うことで，コーディング時に見落とされた欠陥を発見し，修正を加えることができる．コードレビューはソフトウェアの開発元が自ら行う場合と，ソフトウェア開発元と独立した第三者組織が行う場合があり，後者は IV&V(Independent Verification and Validation) と呼ばれる．第三者がコードレビューを行うことで，ソースコードの記述内容の検証を客観的な立場で行うことができる．しかし，規模の大きいソフトウェアのコードレビューは非常にコストが大きい．なぜなら規模が大きくなるほどソースコードの行数，関数や変数の数は多くなるため，複雑な処理内容を正確に把握し，欠陥の有無を検証することが難しくなるからである．第三者組織がコードレビューを行う場合はさらに困難なものとなる．

コードレビューの他にソフトウェアの実装や動作を検証するための手法が提案されており，形式手法 [4] は数学を基盤とした頑健な検証を行うことができる．しかし，人間による証明やコンピューターによる自動証明は時間的または経済的コストが非常に大きく，適用範囲は限られる．プログラムスライシング [11] はソースコード中のスライシング基準を指定することでデータ依存，制御依存のある場所を調べることができ，あらゆる実行経路を検証することができるが，適切なスライシング基準を選ぶにはソースコードの処理内容を把握しておく必要がある．IV&V の場合のように，コードレビュアーが未知のソースコードに対してプログラムスライシングを適用することは難しい．

本研究では信頼性が極めて重要である宇宙機搭載ソフトウェアに対する IV&V に焦点を当て，初見のソースコードのコードレビューを行う上で，大局的な変数のデータフローを素早く検査するための支援ツールを開発した．具体的には，C 言語で記述された宇宙機搭載ソフトウェアに対して，その特徴を考慮したデータフロー解析を行い，ソースコードにおけるモジュール間の変数のデータフローを図示する仕組みを実現した．ソースコードの詳細を知らなくても使用することができ，Reflexion Model[10] に則った形式でモジュール間データフローを図示するため視覚的にも分かりやすいものとなっている．本研究によるモジュール間データフロー図をコードレビューに取り入れることで，より短時間でコードレビューを完了したり，単位時間当たりのコードレビュー量の増加やデータフローの確認漏れが削減されることが期待できる．

評価実験として，実務として第三者コードレビューを行う 8 名の被験者に対し，宇宙機搭載ソフトウェアを模したプログラムを対象としたコード理解度テストを実施した．また，モジュール間データフロー図がコードレビューに与える影響や図の改善案などを調査するアンケートを実施した．モジュール間データフロー図を与える場合と与えない場合でコードレビュー時間が短縮するなどの直接的な効果は見られなかったが，アンケートではモジュール間データフロー図によりコードレビューする上で調査の基点が分かったり，思考の整理がしやすくなるなど実務者にとっても本ツールが有用であるという意見が報告された．

以降，2 章では本研究の背景について述べ，3 章では提案手法を説明する．4 章では評価実験とその結果に対する考察を行い，5 章ではまとめと今後の課題を述べる．

2　背景

宇宙機には従来からソフトウェアが搭載されているが，宇宙機搭載ソフトウェアの欠陥は人命喪失，ミッション不達成，経済的損失などの重大な問題を引き起こすため，高い信頼性が求められる．ソフトウェアを開発する上でコーディング時はもちろん，コードレビュー時にもプログラムが想定通りのふるまいをするかどうかを慎重に検証する必要がある．

しかし，Frederick Brooks の「銀の弾丸などない」[3] で触れられているように，プログラムの検証は非常にコストが大きい．大規模なプログラム全体の検証を行うことは現実的でなく，最重要部のみを検査することしかできない．このような状況で，ソースコードの安全性検査を支援するツールにより検査時間を短縮することができれば，最重要部だけでなくより広範囲の検証を行うことが可能になる．また，ツールによるソースコード解析は人間によるコードレビューで発生し得る検査漏れを防ぐことも期待できる．

2.1　コードレビューの種類

コードレビューには，開発者の知識レベルを向上させることを目指すなどソフトウェアの欠陥の発見を目的としないものと，欠陥の発見を目的とするものがあるが，本研究におけるコードレビューは後者の欠陥を発見することを目的とするものを指す．さらに，コードレビューはソフトウェアの開発元が自ら行う場合と，ソフトウェア開発元と独立した第三者組織が行う場合があり，後者を IV&V(Independent Verification and Validation) と呼ぶ．IV&V は，ソースコードの記述内容の検証を客観的な立場で行えるという利点がある反面，ソースコードの詳細を知らない状態でコードレビューを行うため，コードレビューの難易度は高くなる．

コードレビューによりソフトウェアの欠陥を見つけ出すには，例えば「比較演算子 == と代入演算子 = を混同していないか」や「変数の未初期化」などの発見が容易な項目だけでなく，プログラムのふるまいを精査して「あらゆる入力に対しても予期しない状態に陥ったりしないか」といった事項も確認する必要がある．前者はプログラムの意味を理解していなくても検査可能であるが，後者はプログラムのふるまいを詳細まで把握する必要があり，ツールによる自動修正が困難なため人間によるコードレビューが必須である．

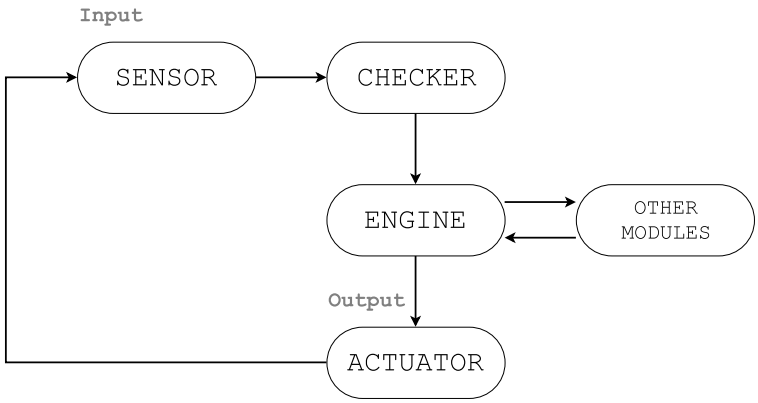

図 1: 宇宙機搭載ソフトウェアの構成の例

2.2 既存のコードレビュー手法

コードレビューはソースコードを目視して行うことが通例であるが，Coverity[1]，Understand[2]，CodeSonar[3] などの静的解析ツールを活用することがある．Coverity は静的解析によりソフトウェアの欠陥や問題を検出するツールの 1 つで，自動的に欠陥の検出を行う手段として用いられる．しかし，静的解析ツールは自動的に欠陥を検出することができるなど効果的である反面，開発ツールとの統合，設定が難しく導入コストが大きいという側面がある．また，狭いソースコード範囲に対して大量の警告が表示されたり，実際にはバグでないものがバグとして検出される誤検知も発生しやすい．プログラムの意味をよく理解した上で行う必要のあるコードレビューでは，やはり目視によるレビューが好まれることがある．

コードレビューを支える手法として Reflexion Model[10] がある．Reflexion Model は，ソースコードのクラスやファイルをより抽象的なモジュールに対応づけ，想定したモジュール間の依存関係と実際のソースコードのモジュール間の依存関係の差分を図示するものである．Reflexion Model を Excel のリエンジニアリングに使用したケーススタディ [9] では，Reflexion Model が大規模ソフトウェアにおけるモジュール間の依存関係を把握し，問題の発見と修正に役立つことが示されている．

2.3 宇宙機搭載ソフトウェアの特徴

宇宙機搭載ソフトウェアは主に C 言語で開発される．安全性を最優先にするスタイルで開発されるため，動的にメモリを確保する `malloc` 関数のように動的に行われる処理は予期せぬ障害の原因となり得るため好まれない．

[1] https://www.synopsys.com/software-integrity/resources/datasheets/coverity.html
[2] https://scitools.com/features/
[3] https://www.grammatech.com/products/codesonar

また，宇宙機搭載ソフトウェアは図 1 のようにセンサからの入力を受け取り，その値を元に複数の独立したモジュールを経てアクチュエータへの出力が決定されるような構成になっており，役割ごとに明確にモジュール分割されていることも特徴である．モジュールは一定の実行サイクルの中で繰り返し処理され，モジュール間のデータの受け渡しはグローバル変数を介して行われる．すなわち，モジュールは他のモジュールに渡したい値をグローバル変数にセットしておき，その値を他のモジュールが読み取るという方法が取られている．グローバル変数の数も多いためこれは非常に厄介な特徴であり，よって宇宙機搭載ソフトウェアに対するコードレビューにおいてモジュール間の変数のデータフローを知ることは重要な要素の 1 つである．

2.4　宇宙機搭載ソフトウェアにおけるコードレビュー

宇宙機搭載ソフトウェアには高い信頼性が求められるため，厳格なコードレビューを行う必要がある．しかし，特殊な特徴を備える宇宙機搭載ソフトウェアのコードレビューの難易度は高く，複雑な処理内容に対して既存の静的解析ツールを使用することは難しい．IV&V のように，ソースコードの詳細を知らないという状況ではさらに困難なものとなる．ここで，ツールによって短時間で正確にモジュール間の変数のデータフローを知ることができれば，宇宙機搭載ソフトウェアにおけるコードレビューの支援に繋がると考えられる．

3　提案手法

本研究では C 言語で記述された宇宙機搭載ソフトウェアの特徴を備えたプログラムに対して軽量なプログラムスライシングを適用し，調べたい変数のモジュール間におけるデータフローを図示するためのツール群を開発した．ツール群の一部のツールは既存のものをそのまま用いている．本ツール群を使用することで，ソースコードの詳細を知らない状態でもモジュール間の変数のデータフローを図示できる．

本章では，初めにツール群の構成を説明し，続いて各ツールの機能や役割，実装を詳しく説明する．

3.1　ツール群の構成

ツール群の構成を図 2 に示す．ツール群は入力として解析対象のソースコード (拡張子が `.c` や `.h` のファイル) と，ソースコード中の関数をどのモジュールに対応づけるかを示すモジュール情報を含む解析設定ファイルを与え，最終的に調査したい変数のモジュール間データフロー図を出力する．ツール群は以下の 3 つのツールから成る．

1. srcML
2. srcSlice (forked)
3. Visual Reflexion

1. は既存のオープンソースソフトウェアであり，ソースコードを抽象的な表現に変換する．2. は既存のオープンソースソフトウェアに対し本研究用に機能追加したもので，プログラムスライシングを行う．3. は新規開発したソフトウェアであり，モジュール間データフロー図を生成する．

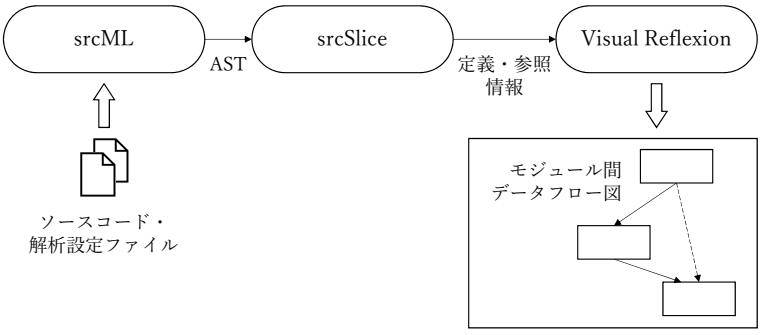

図 2: ツール群の構成

3.2 各ツールの機能

本節ではツール群の各ツールの詳しい機能や役割，実装を説明する．また，ツール群全体の入力としてソースコードに加え，ソースコードの関数とモジュールの対応づけを示すモジュール情報と，モジュール間のデータフローを調べたい変数を示すターゲット変数を与える必要がある．

3.2.1 srcML

srcML[8] はソースコードを XML 文書フォーマットに可逆変換し，様々な解析，操作を可能にする．変換によりソースコードはトークンに分割され，構文木が構築される．本手法では srcML をソースコードの字句解析，構文解析を行うツールとして使用している．

リスト 1，リスト 2 はソースコードの変換例である．リスト 2 において，XML 文書の要素名が抽象構文木におけるノード名に相当する．

リスト 1: 変換前のソースコード

```
v1 = v2;
```

リスト 2: 変換後のソースコード

```
<expr_stmt>
  <expr>
    <name>
      v1
    </name>
    <operator>
      =
    </operator>
    <name>
      v2
    </name>
  </expr>
  ;
</expr_stmt>
```

3.2.2 srcSlice

srcSlice[2] はオープンソースソフトウェアで，srcML が出力した XML 文書を入力として軽量なプログラムスライシングを行うことを目的として開発されたツールである．

プログラムスライシング [11] とは文 s と変数 v の組をスライシング基準と呼ぶとき，その基準から依存関係のある文の集合 (プログラムスライス) を算出することである．注目する変数の影響を受ける文の集合をフォワードスライス，注目する変数に影響を与える文の集合をバックワードスライスと言う．依存関係には主にデータ依存と制御依存の 2 種類がある．プログラム中の文 s1 で変数 v が定義され，v を参照している文 s2 に至る v が再定義されない実行経路が 1 つ以上存在しているとき，v は s1 から s2 へデータ依存していると言う．また，文 s1 が制御文であり，s1 により文 s2 が実行されるかどうかが決まるとき，s1 から s2 へ制御依存していると言う．

srcSlice による解析で得られる変数の情報は次の通りである．

- 定義 (def) 行番号

表 1: srcSlice の変数表の出力内容

アイテム	説明
id	変数に一意に割り当てられる ID
file	変数が宣言されているソースファイルのパス
func	変数が宣言されている関数．グローバル変数の場合は `__GLOBAL__`とする．
var	変数名
def	変数が定義された位置の集合
use	変数が使用された位置の集合
dvars	自身を使用して定義された変数の集合．例えば `y = x;` のとき，x の dvars に y が含まれる．
cfuncs	自身を引数として渡した関数

- 使用 (use) 行番号
- 自身を使用して定義された変数
- 自身がポインタ変数の場合，ポインタが指す変数
- 自身を引数として渡した関数

しかし，オリジナルの srcSlice では本研究のモジュール間データフロー図を出力するために必要な情報を十分に得られないため機能追加をした．機能追加により取得可能となった内容を次に示す．

- グローバル変数
- 構造体変数
- 関数を介して定義/使用される場合の伝播元の関数

srcSlice はカンマ区切りの CSV 形式でそれぞれの変数の定義行や使用行などを出力する．これを**変数表**と呼び，出力内容を表 1 に示す．なお，本研究ではポインタ解析を行わないためオリジナルの srcSlice に含まれるポインタ解析機能は削除している．よって，変数表はポインタに関する情報を含んでいない．

さらに新規に追加した機能として，宣言されている関数の一覧表と If ブロックの一覧表を出力するようにした．これをそれぞれ関数表，制御表と呼ぶ．制御表は簡易的な制御依存

表 2: srcSlice の関数表の出力内容

アイテム	説明
id	関数に一意に割り当てられる ID
func_name	関数名
file_path	関数が宣言されているソースファイルのパス
declare_range	関数が宣言されている範囲．行番号で表される．

表 3: srcSlice の制御表の出力内容

アイテム	説明
id	If ブロックに一意に割り当てられる ID
file	If ブロックのあるソースファイルのパス
control_range	If ブロックの範囲．行番号で表される．
control_vars	If 条件式の中で使用される変数の ID の集合

を踏まえた解析のために使用し，If ブロックに関する情報を含んでいる．関数表，制御表も変数表と同様にカンマ区切りの CSV 形式で出力する．出力内容は表 2，表 3 の通りである．

3.2.3 Visual Reflexion

Visual Reflexion は変数のデータフローを Reflexion Model[10] に倣って図示するツールであり，Java で新規に開発した．本研究の対象ソフトウェアは関数やライブラリの単位でモジュール分割が明確なため，Reflexion Model の考え方を適用することができ，モジュール間のデータフローを大局的に表すことに繋がる．

以降，Visual Reflexion の入力，出力，アルゴリズムを説明する．

入力

srcSlice の出力と解析設定ファイルを与える．解析設定ファイルには

- プロジェクト名
- エントリーポイントとなる関数名．通常は `main` 関数．
- ターゲット変数の集合
- 関数とモジュールの対応の集合

の 4 点を記述する．ターゲット変数にはデータフローを調べたい変数を指定する．また，関数とモジュールの対応づけを指定することで，データフロー図を関数間ではなくモジュール間に対して生成することができる．

出力

DOT 言語で記述されたモジュール間データフロー図 (Graphviz[5] 等で変換が必要) であり，図 3 はその例である．各頂点はモジュールを表し，モジュールからモジュールへの有向辺はデータフローを表す．有向辺に添えられたラベルはデータフローのある変数の名前である．実線の有向辺は以下のいずれかの場合を表し，これを依存タイプ 1 と呼ぶ．

- 辺の始点のモジュール内でターゲット変数の値が定義され，この値を使用して辺の終点のモジュール内で新たにターゲット変数の値が定義された場合
- 定義されたターゲット変数の値がプログラム実行サイクルの最後に実行されるモジュール内で使用された場合

点線の有向辺は，辺の始点のモジュール内でターゲット変数の値が定義されたが，この値が辺の終点のモジュール内で参照されなかった場合を表し，これを依存タイプ 2 と呼ぶ．また，ターゲット変数の値が未定義である場合や，値がのちのモジュール内で更新されるために値が到達し得ない場合にはモジュール間の有向辺は存在しない．

アルゴリズム

srcSlice の出力と解析設定ファイルから，モジュール間データフローグラフを構築する手順を説明する．

1. srcSlice の出力，すなわち変数の定義 (def) 行番号や使用 (use) 行番号などの情報の一覧である変数表，宣言されている関数の一覧である関数表，If ブロックの一覧である制御表と解析設定ファイルを読み込む．
2. モジュール内のデータ依存関係を解析する．エントリポイントとなる関数を基点にし，解析設定ファイル中のモジュール情報においてモジュールに対応づけられている関数に対し以下の操作を行う．
   (a) 関数内の命令を上から下へ順に 1 行ずつ解析し，変数の依存関係と参照される変数の情報を取得する．取得した変数の依存関係は，どの変数からどの変数へ値が渡るかを求めるために使用する．
   - その行で呼び出された関数に由来する定義 (def) または使用 (use) が発生している場合，その関数を再帰的に解析する．

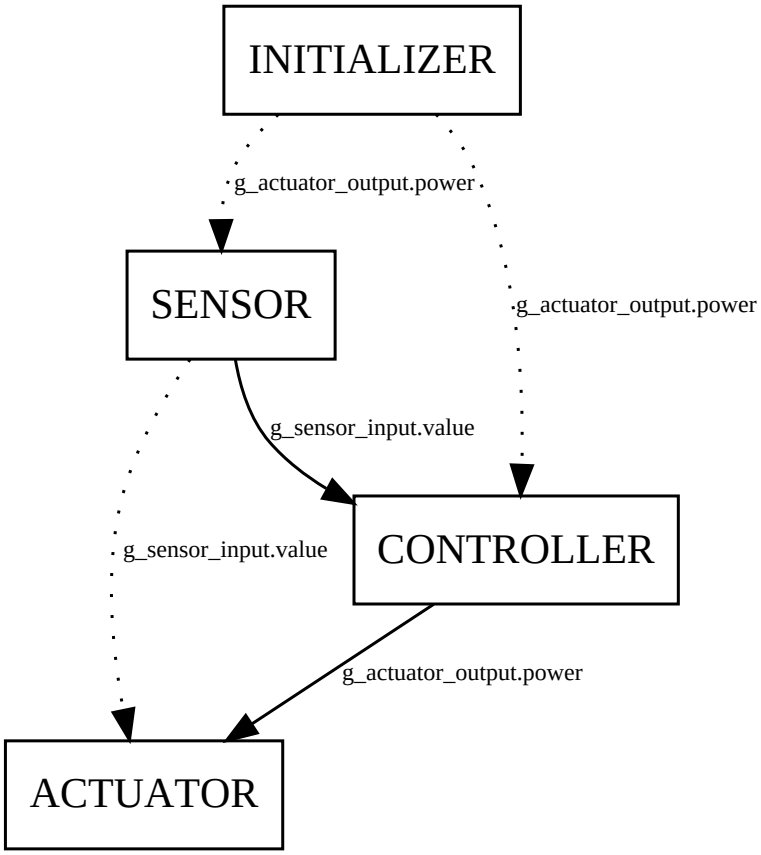

図 3: Visual Reflexion により生成されるモジュール間データフロー図の例

- 定義 (def) と使用 (use) が両方発生した場合は値の代入が起きたとみなして，使用された変数 (a と呼ぶ) から定義された変数 (b と呼ぶ) への依存関係「a → b」を得る．
- 定義 (def) のみが発生した場合は依存関係「X → b (X は定数など変数以外の要素を表す)」を得る．
- 使用 (use) のみが発生した場合は依存関係「a → NULL」を得る．

(b) 解析によって得られた依存関係と参照される変数の情報をモジュールごとにまとめ，リストにする．このリストをモジュールユニットリストと呼び，リストの要素であるモジュールユニットは 1 つのモジュール内における依存関係，参照される変数の情報，及びモジュール自身の情報を含んでいる．また，このときリスト中のモジュールユニットはプログラムの実行時にモジュールが処理される順と同じ並びになっている．

3. 2. (b) で得られたモジュールユニットリストを元にグラフを構成する．

(a) モジュール間の有向辺を格納する集合 E を作成する．

(b) 関数に対応づけられたモジュールのうち，最初に実行されるモジュールで定義される変数をそれぞれ<変数，モジュール名>の組 (変数定義データと呼ぶ) にして，リスト defList を作成する．

(c) 2 番目以降に実行される各モジュール ($module$ と呼ぶ) について以下の操作を行う．

i. $module$ 内でターゲット変数の値が定義されるような経路を列挙し，この経路の集合を paths とする．ここで，経路とはあるターゲット変数を開始点として，あるターゲット変数の値が定義されるまでに 2. (a) で取得した変数の依存関係において現れる変数の列のことである．例えば，x，z がターゲット変数であるとして,「x → y」,「y → z」の依存関係が得られているとき，経路「x → y → z」が求まる．

ii. リスト defList 中の各変数定義データ (<v, m>とおく) について以下の操作を行う．

A. モジュール間の有向辺を<辺の始点となるモジュール名 m，辺の終点となるモジュール名 $module$，データフローのある変数 v，依存の種類 $type$>の組で表すとして，有向辺<$m, module, v, type$>を集合 E に追加する．$type$ は，paths のうち始点となる変数が v と一致するか，プログラム実行サイクルの最後に実行されるモジュール内で v が使用される場合は依存タイプ 1，使用されない場合は依存タイプ 2 とする．

iii. *module* 内で定義される変数をそれぞれ<変数，モジュール名>の組 (変数定義データ) にして，リスト `additionalDefList` を作成する．

iv. `defList` 中の変数定義データのうち，`additionalDefList` 中の変数定義データと変数において同一のものを削除する．これは，古い定義が新しい定義によって上書きされることにより変数の値の寿命が切れることを意味している．条件分岐によっては定義が上書きされないことがある場合でも，本実装では必ず上書きされるとみなす．

v. `defList` と `additionalDefList` を結合する．

(d) モジュール間の依存辺の集合を DOT 言語に変換する．有向辺は依存の種類に応じて依存タイプ 1 ならば実線，依存タイプ 2 ならば点線とする．

4　評価実験

評価では第三者コード検査を実務として行っている者を被験者として，本ツールが生成するモジュール間データフロー図がコード理解に役立つかどうかを検証する．そのために宇宙機搭載ソフトウェアの特徴を備えたプログラムに対して理解度を測るためのテスト問題を作成し，一部の被験者には本ツールにより生成したモジュール間データフロー図を与えてモジュール間データフロー図を与えた場合と与えない場合で問題の回答時間に差が生じるかどうかを調べた．実際の宇宙機に搭載されるソフトウェアのソースコードは入手が困難であるため，教育版レゴ マインドストーム EV3[1] 上で動作するローバー (探査車) の動きを模した小規模なプログラムを評価実験用に用意した．また，本ツールが生成する図についての意見や改善案を問うためのアンケートも実施した．

4.1　実験概要

被験者

第三者コード検査を実務として行う 8 名．被験者の 8 名中，半数の 4 名をグループ A，もう一方の 4 名をグループ B として実験を実施した．グループ A とグループ B において，第三者コード検査の実務経験年数の平均がほぼ同一となるように調整されている．

実験手順

プログラムのソースコードは電子媒体で配布し，テスト問題は紙に印刷したものを配布して，回答は同紙に直接書き込む形にした．問題の回答時間は 60 分を目安とし，回答終了後は被験者全員にアンケートに回答させた．また，ソースコードはソースコードは日常的に使い慣れているエディタで閲覧し，エディタの検索機能は使用してよいものとした．

対象プログラム

教育版レゴ マインドストーム EV3 上で動作する，実際の探査車のように障害物を回避しながらゴール地点に進むプログラム．宇宙機搭載ソフトウェアを模したプログラム構造であり，グローバル変数が多用され，関数単位での明確なモジュール分割がなされている．C 言語で記述されソースファイル数は 11，コードサイズは全体で約 500 行であり，IV&V に従事する者が作成した．

対象プログラムの処理の流れを図 4 に示す．プログラム起動時に初期化処理を行い，その後は「センサからの入力を取得」→「センサの入力値を踏まえ状態遷移，出力値を決定」→「モーターを駆動」の処理サイクルを繰り返す．

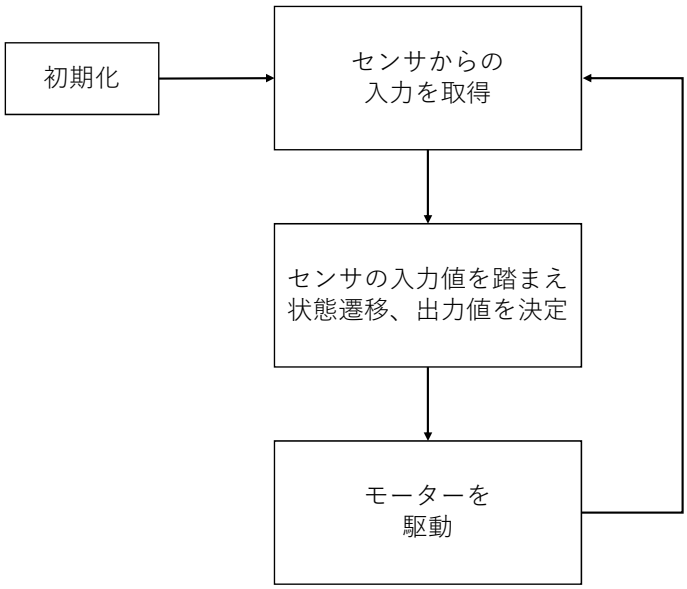

図 4: 対象プログラムの処理の流れ

表 4: 対象プログラムで使用する主要な変数

変数	説明
g_in	主にセンサで取得した値を格納
g_out	主にモーターの出力量を格納
g_status	プログラムの内部状態を格納

対象プログラムで使用する主要な変数を表 4 に示す．変数 g_in，g_out は構造体変数で，それぞれセンサで取得した値とモーターの出力量を格納する重要な変数である．g_status はプログラムの内部状態を格納する変数で，間接的に入出力に影響を受けたり与えたりする．g_status のように，間接的に処理内容に関わる変数はプログラムの理解をより難しいものにする．

テスト問題とその出題意図

テスト問題では大問を 2 問出題した．

問 1 は変数 g_in と g_out のプログラム中における役割を選択肢から選び，その選択肢を選んだ根拠を説明する問題である．

問 2 は問 2.1，問 2.2，問 2.3 の 3 問に分かれている．問 2.1 は構造体変数 g_in の各メンバの値は「SENSOR モジュール」内でのみセットされるべきだが，その通りの実装になっているかをレビューせよという問題である．問 2.2 は問 2.1 と同様に，構造

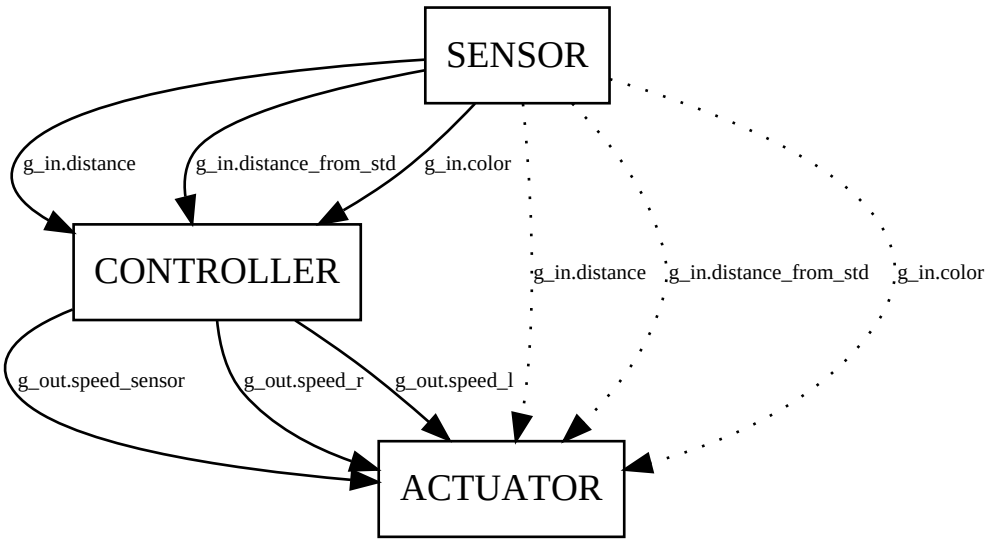

図 5: 対象プログラムのモジュール間データフロー図

体変数 g_out の各メンバの値は「CONTROLLER モジュール」内で変数 g_in の値を元に決定される，すなわちデータ依存か制御依存があるべきだが，その通りの実装になっているかをレビューせよという問題である．問 2.3 はグループ A にのみ出題し，本研究で開発したツール群が生成する図と同様の図を作図せよという問題である．

問 1 は対象プログラムに慣れるための問題である．問 2.1，問 2.2 については，グループ B にのみ本研究で開発したツール群が生成するモジュール間データフロー図 (図 5) とその生成方法の説明を与え，モジュール間データフロー図が与えられないグループ A との比較を行う．問 2.3 については，モジュール間データフロー図を手作業で作図した場合にかかる時間や図の正確さを調べるために出題した．

アンケート

アンケート内容を以下に示す．

- 第三者コード検査の実務経験年数
- C 言語の経験年数
- モジュール間データフロー図はプログラムの理解速度に良い影響を与えるか (1〜5 の 5 段階評価)，またその理由 (自由記述式)
- モジュール間データフロー図の意味は分かりやすいか (1〜5 の 5 段階評価)
- モジュール間データフロー図を改善するためにはどうすればよいか (自由記述式)

経験の違いを考慮して評価を行うために，第三者コード検査や C 言語の経験年数を

表 5: モジュール間データフロー図生成環境の性能

CPU	Intel Core i7-4770 3.40 GHz
OS	Windows 10 Pro 64-bit
メモリ	32.0 GB

聞き取った．また，モジュール間データフロー図が与えられないグループ A の被験者を含めた全ての被験者に対してモジュール間データフロー図の効果，明確さ，改善の方針の聞き取りを行った．

評価方法

テスト問題の問 2 の回答にかかった時間，アンケートの回答内容を踏まえてツールの有用性を評価する．テスト問題の問 1 は対象プログラムのソースコードに慣れるための問題であるため，評価対象から外した．

なお，対象プログラムのモジュール間データフロー図の生成にかかる時間は表 5 に示す環境下で約 300 ミリ秒である．

4.2 実験結果

本節では評価実験の結果を示す．

4.2.1 テスト問題の結果

テスト問題の回答時間の集計を表 6 に示す．表中の ID は被験者を識別するためのもので，ID が A から始まっているものはグループ A の被験者を表し，ID が B から始まっているものはグループ B の被験者を表す．問 1 のグループ A，グループ B の平均回答時間はそれぞれ 11 分，7 分，同様に問 2.1 はそれぞれ 11 分，24 分，問 2.2 はそれぞれ 17 分，25 分であった．グループ A のみに出題した問 2.3 の平均回答時間は 21 分であった．また，問 2.1，問 2,2，問 2.3 の回答時間の箱ひげ図を図 6，7，8 に示す．

4.2.2 アンケートの結果

アンケートの結果の集計を表 7 に示す．被験者 B-4 を除いて第三者コード検査の実務経験年数は 3 年から 18 年，C 言語の経験年数はは 5 年から 27 年であった．また，モジュール間データフロー図が理解度に良い影響を与えるかという質問の平均スコアは 4.125，モジュール間データフロー図の意味は分かりやすいかという質問の平均スコアは 3.875 であった．

表 6: テスト問題の回答時間の集計 (単位は 時:分)

ID	問 1	問 2.1	問 2.2	問 2.3	問 2 の総計
A-1	0:06	0:08	0:20	0:15	0:28
A-2	0:15	0:13	0:23	0:15	0:36
A-3	0:10	0:15	0:10	0:30	0:25
A-4	0:15	0:11	0:18	0:24	0:29
B-1	0:04	0:16	0:23	／	0:39
B-2	0:04	0:20	0:39	／	0:59
B-3	0:05	0:09	0:07	／	0:16
B-4	0:16	0:54	0:32	／	1:26
全体平均	0:09	0:18	0:21	0:21	0:39
A の平均	0:11	0:11	0:17	0:21	0:29
B の平均	0:07	0:24	0:25	／	0:50

また，モジュール間データフロー図がプログラムの理解速度に良い影響を与える理由として以下の意見があった．

- `g_status`(プログラムの状態を格納する変数) のような間接的な変数を無視して，`g_in`(入力を格納する変数) と `g_out`(出力を格納する変数) といった本質的なコントロールの流れを理解できる．
- 想定しているプログラムの挙動を抽象度の高いレベルで把握できる．
- モジュール間のデータ依存，制御依存関係が分かる．
- モジュール間 (関数間) でのデータの影響を視覚的に捉えることができ，思考する際に整理がしやすい．
- 着目する変数の値の更新がどこで行われるかが判別できる．使われ方やどのような条件でどんな値が設定されるかを調査する基点が分かる．

モジュール間データフロー図を改善するためにはどうすればよいかという質問には以下の意見があった．

- 有向辺において，実線と点線の意味合いが分かりにくい．

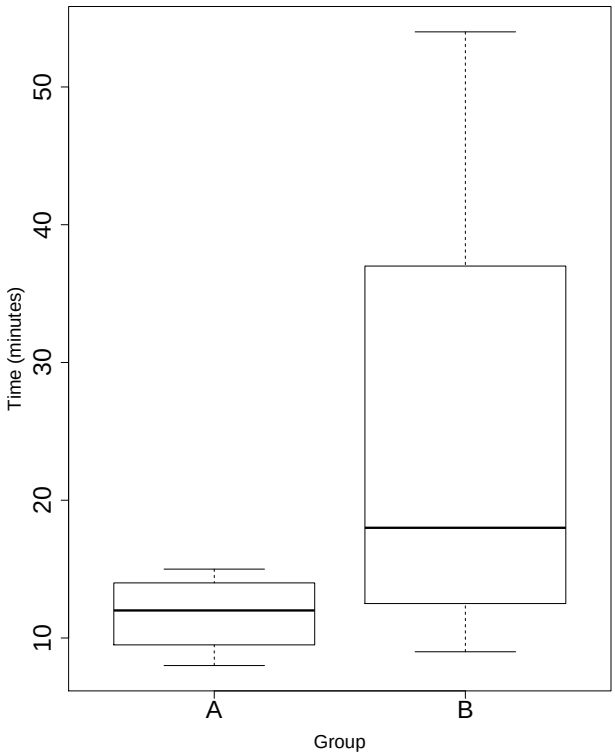

図 6: 問 2.1 の回答時間の箱ひげ図

- 実線，点線が同じ場合は構造体のメンバではなく構造体レベルで有向辺をまとめた方が理解しやすい．
- 点線の有向辺の用途が不明．図が大きくなる時には煩雑になるのではと思う．
- 現在の規模だと十分分かりやすく感じる．しかし，ソースコードの規模が大きくなると点線の有向辺で表現した部分が大きくなり，全体の可視性を下げることになる可能性がある．規模によっては点線の有向辺は無くてもいいかもしれない．

4.3 考察

本節では評価実験の結果についての考察を行う．

4.3.1 テスト問題の結果についての考察

全ての問題について，被験者の中で明らかな誤答を導いた者はいなかったため，問 2.1，問 2.2 についてはモジュール間データフロー図を与えられたグループ B がモジュール間データフロー図を与えられないグループ A に比べてより短時間で問題を解くことができたかを評価軸とする．問 2.3 については，モジュール間データフロー図の作図に要した時間に加えて，

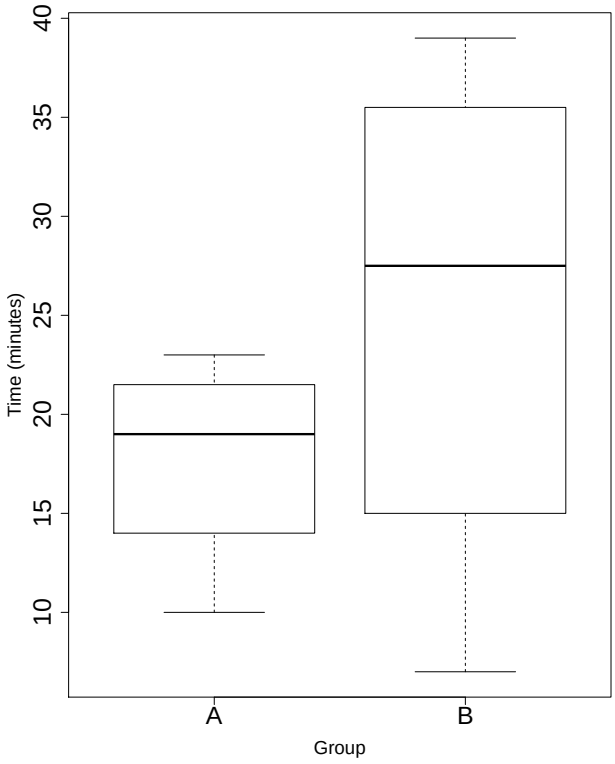

図 7: 問 2.2 の回答時間の箱ひげ図

過不足なくモジュール間のデータフローを書き出して本ツール群が生成するものと同等の図を作図できたかを評価する．なお，問 1 については前述の通り評価軸に含めない．

問 2.1，問 2.2 について，問題の平均回答時間はモジュール間データフロー図を与えられたグループ B の方が図を与えられないグループ A に比べて長かった．これは各被験者の能力差の影響や，モジュール間データフロー図を与えられたことでより詳細にソースコードを読み込めたために回答もより詳細になり，余分に時間がかかった可能性などが原因として考えられる．

問 2.3 の回答時間の平均は 21 分であった．被験者は問 2.3 に至る前に問 1，問 2.1，問 2.2 を解く過程で対象プログラムのソースコードを繰り返し読んでいると予想されるが，手作業によるモジュール間データフロー図の作図はある程度の時間的コストがかかることが分かった．評価実験対象プログラムの場合，本ツール群を使用すればモジュール間データフロー図を 1 秒未満で生成することができる．また，完全な正答を導いた被験者は 1 名で，他の 3 名にはモジュール間のデータフローを表す有向辺の記入漏れや有向辺の種類 (実線・点線) の誤りなどの間違いが見られた．これらのことから，手作業によるデータフローの抽出は時間的コストがかかり，しばしば確認漏れが発生することもあるが，本ツール群を使用することで短時間でかつ正確なモジュール間データフロー図を得られるという利点が確認された．

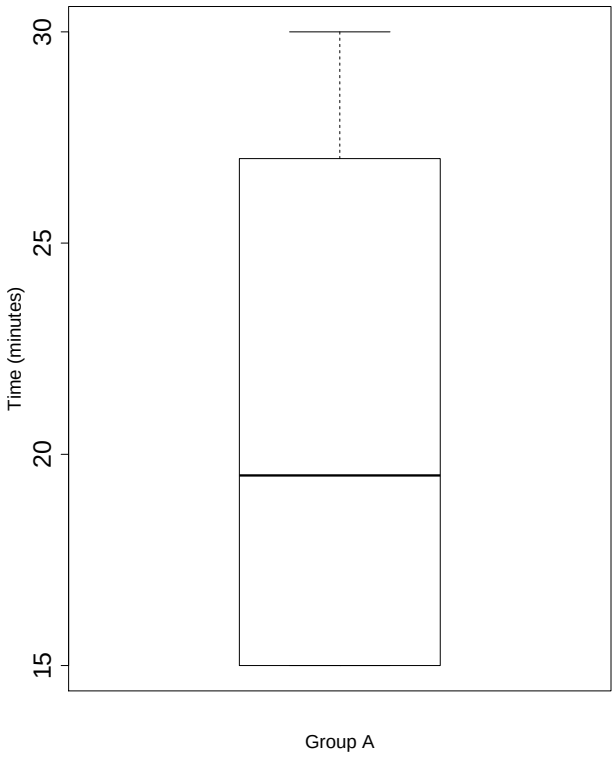

図 8: 問 2.3 の回答時間の箱ひげ図

4.3.2 アンケートの結果についての考察

モジュール間データフロー図がプログラムの理解速度に良い影響を与えると回答した理由については，重要な変数の本質的なデータフローを理解することができる，未知のソースコードでもコードレビューを行う基点のおおよその検討を立てることができる，モジュール間のデータフローを図で表現することで視覚的に変数のデータフローを把握することができるため思考の整理がしやすくなる，などの好意的な意見が多く得られた．このことから，実務で第三者コード検査を行う者にとって本ツール群によるモジュール間データフロー図がソースコードを理解する上で役に立つことが分かった．

モジュール間データフロー図の改善案については，データフローを表す有向辺の種類 (実線・点線) の違いが分かりにくいことや，点線の有向辺は特にプログラムの規模が大きくなった際に数が多くなり図が煩雑になるのではないかという懸念が報告された．点線の有向辺はデータフローが無いことを表すものであり，プログラムの規模が大きくなるほど大量の点線の有向辺が出現することが予想される．よって，点線の有向辺の表示を切り替えることのできるオプションを追加するなどの改善が必要であると考えられる．

表 7: アンケートの結果の集計

ID	第三者コード検査経験 (年)	C 言語経験 (年)	理解度に良い影響を与えるか？ (1〜5)	分かりやすいか？ (1〜5)
A-1	15	15	4	2
A-2	7	27	5	5
A-3	4	5	4	4
A-4	3	6	4	5
B-1	18	20	4	4
B-2	12	12	4	4
B-3	3	10	4	4
B-4	0.5	1	4	3
平均	7.8125	12	4.125	3.875
A の平均	7.25	13.25	4.25	4
B の平均	8.375	10.75	4	3.75

4.3.3 更なる評価実験の検討

本評価実験ではモジュール間データフロー図による直接的なコードレビュー時間の短縮が見られなかった．Kashima らによるデータフロー可視化の有用性を調査した実験 [6] では，可視化有りの場合の方が単位時間あたりのコード調査範囲が広がっているという結果が報告されている．そのため，より広範囲のコード調査が必要となる規模の大きいソフトウェアを対象として本評価実験と同内容の実験を行うことで，有意なコードレビュー時間の短縮が見られる可能性がある．また，本評価実験では対象ソフトウェアが 1 点であったことからモジュール間データフロー図の有無により同一の被験者のコードレビュー時間に変化が見られるかどうかを検証することができなかった．評価実験対象ソフトウェアを増やして実験を行うことも今後の追加実験の方針として挙げられる．

5　まとめと今後の課題

本研究では，宇宙機搭載ソフトウェアの安全性検査のためのデータフロー検査支援ツールの開発を行い，実務として第三者コード検査を行う者を被験者として評価実験を行った．評価実験により本ツール群が生成するモジュール間データフロー図によってコードレビュー時間が短縮されるような直接的な効果は確認されなかったが，アンケートでは重要な変数の本質的なデータフローを理解することができたり，データフローを視覚的に捉えることができ思考を整理しやすくなるなど，本ツール群が効果的であるという意見が多く報告された．

今後の課題として，srcSlice の機能の安定化，Visual Reflexion の機能強化などが挙げられる．特に Visual Reflexion はインタラクティブに解析対象の変数を変更したり，クエリで条件を指定することにより確認したいデータフローだけを表示するなど，多くの改善が考えられる．また，本研究では Reflexion Model に則ってデータフローを可視化したが，他のダイアグラム表現での可視化も視野に入れたい．評価実験で得られた意見を踏まえて本ツール群を発展させ，より大規模な実験を行って本研究の提案の有用性を定量的に示し，実現場で十分に活用できる水準を目指す．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎 教授には，研究において多くの意義ある御意見を賜りました. 心より感謝いたします.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，研究において多くの御力添えを賜りました．心より感謝いたします.

奈良先端科学技術大学院大学情報科学研究科ソフトウェア工学研究室 石尾隆 准教授には，研究において多くの適切な御指導及び御助言を賜りました．石尾隆 准教授のおかげで本論文を完成させることができました．心より深く感謝いたします.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 特任助教には，研究において多くの御助言を賜りました．心より感謝いたします.

有人宇宙システム株式会社 川口真司 氏，並びに同社安全開発保証部の皆様には，研究における御相談，及び評価実験のご協力を賜りました．心より感謝いたします.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊藤薫，嶋利一真，並びに大阪大学基礎工学部情報科学科 藤原裕士，小笠原康貴 諸氏には，研究に関する相談に乗っていただき，様々なご支援を頂きました．心より深く感謝いたします.

最後に，私を常に支えてくださった大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上研究室の皆様に心より感謝いたします.

参考文献

[1] 31313 mindstorms ev3. https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313.

[2] Hakam W Alomari, Michael L Collard, Jonathan I Maletic, Nouh Alhindawi, and Omar Meqdadi. srcslice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process*, Vol. 26, No. 11, pp. 931–961, 2014.

[3] F. P. J. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, Vol. 20, No. 4, pp. 10–19, April 1987.

[4] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, Vol. 28, No. 4, pp. 626–643, 1996.

[5] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pp. 483–484. Springer, 2001.

[6] Yu Kashima, Takashi Ishio, Shogo Etsuda, and Katsuro Inoue. Variable data-flow graph for lightweight program slicing and visualization. *IEICE TRANSACTIONS on Information and Systems*, Vol. E98-D, No. 6, pp. 1194–1205, 2015.

[7] Jacques-Louis Lions, et al. Ariane 5 flight 501 failure. Technical report, European Space Agency. http://www. esrin. esa. it/htdocs/tidc/Press/Press96/ariane5rep. html, 1996.

[8] Jonathan I Maletic, Michael L Collard, and Andrian Marcus. Source code files as structured documents. In *Program comprehension, 2002. proceedings. 10th international workshop on*, pp. 289–292. IEEE, 2002.

[9] Gail C. Murphy and David Notkin. Reengineering with reflexion models : A case study. *IEEE Computer*, Vol. 30, No. 8, pp. 29–36, 1997.

[10] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 4, pp. 18–28, 1995.

[11] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pp. 439–449. IEEE Press, 1981.