ソフトウェア・シンポジウム 2018 in 札幌

論文集

ソフトウェア技術者協会

募集案内

ソフトウェア・シンポジウムは，ソフトウェア技術に関わるさまざまな人びと，技術者，研究者，教育者，学生などが一堂に集い，発表や議論を通じて互いの経験や成果を共有することを目的に，毎年全国各地で開催しています．

第 38 回目を迎える 2018 年のソフトウェア・シンポジウムでは，この数年間で試みてきた新しい取り組み（チュートリアルや Future Presentation など）をさらに発展させたものにしたいと考えています．このほか，SS2017 に引き続き，論文発表や事例報告と，ワーキンググループで議論を行います．

開催概要

■ 日程 ： 2018 年 6 月 6 日（水曜日）～ 8 日（金曜日）

■ 場所 ： かでる 2・7

■ 主催 ： ソフトウェア技術者協会

■ 後援 ： 情報処理推進機構

■ 協賛 ： LOCAL，北海道 IT 推進協会，札幌市 IoT イノベーション推進コンソーシアム／札幌市，UNISON 札幌市IT振興普及推進協議会，ソフトウェアテスト技術振興協会，アジャイルプロセス協議会、オープンソースソフトウェア協会，情報サービス産業協会，情報処理学会，ソフトウェア・メインテナンス研究会，電子情報通信学会，日本ソフトウェア科学会，組込みシステム技術協会，日本 SPI コンソーシアム，日本ファンクションポイントユーザ会，派生開発推進協議会，日本科学技術連盟，組込みソフトウェア管理者・技術者育成研究会，TOPPERS プロジェクト，PMI 日本支部，アドバンスト・ビジネス創造協会

スタッフ一覧

実行委員会

実行委員長

中野 秀男（帝塚山学院大学）
本多 慶匡（東京エレクトロン）

実行委員

伊藤 昌夫（ニルソフトウェア）
小笠原 秀人（千葉工業大学）
小楠 聡美（HBA）
岸田 孝一（SRA）
栗田 太郎（ソニー）
小松 久美子（帝塚山学院大学）
杉田 義明（福善上海）
鈴木 裕信（usp lab.）
萩原 美穂（アートシステム）
奈良 隆正（NARA コンサルティング）
野村 行憲（フリー）
松田 英克（東京エレクトロン）
宮田 一平（SHIFT）
三輪 東（SCSK）

プログラム委員会

プログラム委員長

安達 賢二（HBA）
落水 浩一郎（University of Information Technology, Myanmar）

プログラム委員

秋山 浩一（富士ゼロックス）
天嵜 聡介（岡山県立大学）
荒木 啓二郎（九州大学）
伊藤 昌夫（ニルソフトウェア）
臼杵 誠（富士通）
大平 雅雄（和歌山大学）
小笠原 秀人(千葉工業大学)
小田 朋宏（SRA）
片山 徹郎（宮崎大学）
神谷 年洋（島根大学）
北須賀 輝明（広島大学）
日下部 茂（長崎県立大学）
楠本 真二（大阪大学）
栗田 太郎（ソニー）
河野 哲也（ディー・エヌ・エー）
小林 修（SRA）
小林 展英（デンソークリエイト）
後藤 徳彦（NEC ソリューションイノベータ）
古畑 慶次（デンソー技研センター）
阪井 誠（SRA）
酒匂 寛（デザイナーズデン）
菅原 広行（ソニー）
鈴木 裕信（鈴木裕信事務所）
鈴木 正人（北陸先端科学技術大学院大学）
高木 智彦（香川大学）
田中 康（奈良先端科学技術大学院大学）
張 漢明（南山大学）
角田 雅照（近畿大学）

土肥 正（広島大学）
富松 篤典（電盛社）
中谷 多哉子（放送大学）
中森 博晃（パナソニック スマートファクトリーソリューションズ）
西 康晴（電気通信大学）
根本 紀之（東京エレクトロン）
野村 行憲（フリー）
野呂 昌満（南山大学）
端山 毅（NTT データ）
松尾谷 徹（デバッグ工学研究所）
松本 健一（奈良先端科学技術大学院大学）
水野 修（京都工芸繊維大学）
宗平 順己（Kyoto ビジネスデザインラボ合同会社）
森崎 修司（名古屋大学）
諸岡 隆司（中電シーティーアイ）
八木 将計（日立製作所）
山本 修一郎（名古屋大学）
米島 博司（パフォーマンス・インプルーブメント・アソシエイツ）
劉 少英（法政大学）

ソフトウェア・シンポジウム 2018 in 札幌 目次

■論文・報告 形式手法

■論文・報告 品質管理

■論文・報告 チーム力向上

■論文・報告 開発管理

■論文・報告 個と組織の成長

■論文・報告 要求工学

アジリティのある探索的形式仕様記述のための テストフレームワーク

小田 朋宏
株式会社 SRA
tomohiro@sra.co.jp

荒木 啓二郎
熊本高等専門学校
araki@kyudai.jp

要旨

VDM-SL は実行可能なサブセットを持つ形式仕様記述言語である．我々は，形式仕様記述工程の初期段階における探索的な記述プロセスに注目し，それを支援するための仕様記述環境として *ViennaTalk* を開発してきた．探索的な仕様記述では仕様には頻繁に誤りが発生するため，仕様記述の生産性と品質向上のためには，誤りの早期発見と効率的な除去が求められる．本稿では，探索的な試行錯誤の中で *VDM-SL* 仕様の記述誤りを早期に発見するためのテストフレームワーク *ViennaUnit* について，*ViennaTalk* での実装を説明し，ツール機能としての応用と評価を示す．

1. はじめに

ソフトウェアシステムの仕様を形式的に記述することで 仕様記述の問題点をより早期に発見し修正する開発手法として, 形式手法が注目されている [1, 2]．VDM-SL [3] は実行可能なサブセットを持ち，複数のインタプリタ実装および自動コード生成器が提供されている形式的仕様記述言語である [4]．実行可能な VDM 仕様は vdmUnit 等のテストフレームワークを利用して単体テストを継続的に行うことで，高い品質の仕様記述が可能であることが事例から知られている [5].

筆者らはこれまで仕様記述工程の初期段階に着目し，探索的に形式仕様記述を遂行することを支援する環境 ViennaTalk[6] を開発してきた．探索的な仕様記述においては試行錯誤が多く行われ，問題に対する新たな知見に基づく仕様記述の変更が頻繁に行われる．そのため，高いアジリティが求められる．ViennaTalk は，Smalltalk 処理系の 1 つである Pharo[7] 環境上に構築されたメタ IDE であり，VDM-SL の外部インタプリタを利用するためのブリッジ，VDM-SL 仕様から Smalltalk ライブラリソースを出力する自動コード生成器，パーサ，構文ハイライトや自動整形機能付きのエディタ部品など，VDM-SL 向けの様々なツールを開発するためのコアとなるコンポーネントを提供している．現在 ViennaTalk 上で利用可能なツールとして，WebIDE サーバ VDMPad，VDM-SL 仕様の編集およびライブ実行をおこなう VDMBrowser，仕様の妥当性を確認するための UI プロトタイピング環境 Lively Walk-Through, Web API プロトタイプ環境 Webly Walk-Through がある．本稿では，ViennaTalk が提供するツールのうち VDMBrowser を取り上げて，探索的仕様記述に適したテストフレームワーク ViennaUnit と VDMBrowser での実現を説明する．そして，VDMBrowser での実現の評価として，仕様の誤りの早期発見につながるかを分析し議論する．

2. VDM-SL および ViennaTalk の概要

VDM-SL は ISO により標準化されたモデル規範型の形式的仕様記述言語で，記述対象を抽象するための型，定数，関数を定義し，状態空間の定義と状態を変更する操作を定義することで，システムの入出力や状態変化を記述する．VDM-SL にはモジュール機能があり，各モジュールは imports 宣言 および exports 宣言により外部インターフェイスを宣言する．VDM-SL で提供されている型には，自然数型，非 0 自然数型，整数型，小数型，文字型，ブール型，引用型，トークン型などの基本型と，複合型，列型，集合型，直積型，合併型，オプション型，

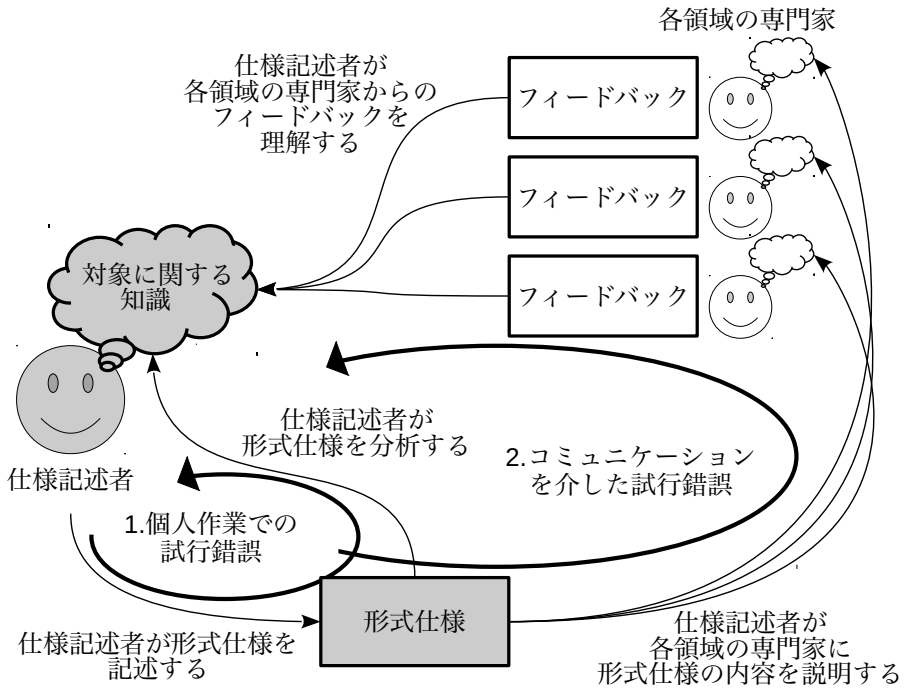


図 1. 探索的仕様記述における試行錯誤のプロセス

関数型などの合成型がある．文字列のための型はなく，文字列は文字型の列 `seq of char` として扱う．VDM-SL での関数とは参照透明な関数であり，操作とは状態への参照や破壊的代入を伴う手続きである．VDM-SL ではそれらの関数や操作に対して事前条件および事後条件を表明することで，それに対応する実装が満たすべき性質を定義する．また，型および状態に対して不変条件を表明することができる．VDM-SL は実行可能なサブセットを持つことから，プログラミング言語と同様にインタプリタ実行もしくはコード自動生成器を通してプログラムとして実行することも可能であるが，不変条件，事前条件および事後条件からなる表明が形式仕様としての重要な役割を果たす．

ViennaTalk は，Smalltalk 処理系の 1 つである Pharo[7] 環境上に構築された探索的仕様記述 [6] 環境である．探索的仕様記述とは形式仕様記述工程の初期段階に見られる試行錯誤を伴う工程であり，形式仕様が顧客の要求および対象ドメインに適合し，機能項目の記述漏れがないことを目標とする．探索的仕様記述に続いて精緻化により，機能定義について未定義部分や矛盾がない厳密な仕様を記述する．

図 1 に探索的仕様記述のプロセスを示す．探索的仕様記述では，個人作業における試行錯誤と，コミュニケーションを介した試行錯誤の，2 種類の試行錯誤が繰り返し行われる [6]．個人作業における試行錯誤は，仕様記述者が持つ記述対象に関する知識に基づいて，どのような機能項目をどのように定義するかについて，適切な記述を模索しながら進める作業である．コミュニケーションを介した試行錯誤は，作業中の仕様を各領域の専門家に説明し，フィードバックを得ることによって，より適切な記述を模索する作業である．これら 2 つの試行錯誤を通して，仕様記述者が対象に関する知識を深めていくことで，顧客の要求および対象ドメインに適合した仕様を作成する．ViennaTalk は探索的仕様記述に特化した仕様記述環境であり，形式仕様の初期段階を形成するための試行錯誤を支援する記述環境や，顧客および対象ドメインの専門家などからのフィードバックを得るためのプロトタイピング環境をツールとして提供している．

本節では特に，本稿のテーマであるテストフレームワークを利用するためのツールである VDMBrowser を説明する．VDMBrowser は探索的仕様記述を支援する仕様記述環境であり，大きな特徴として，仕様を記述す

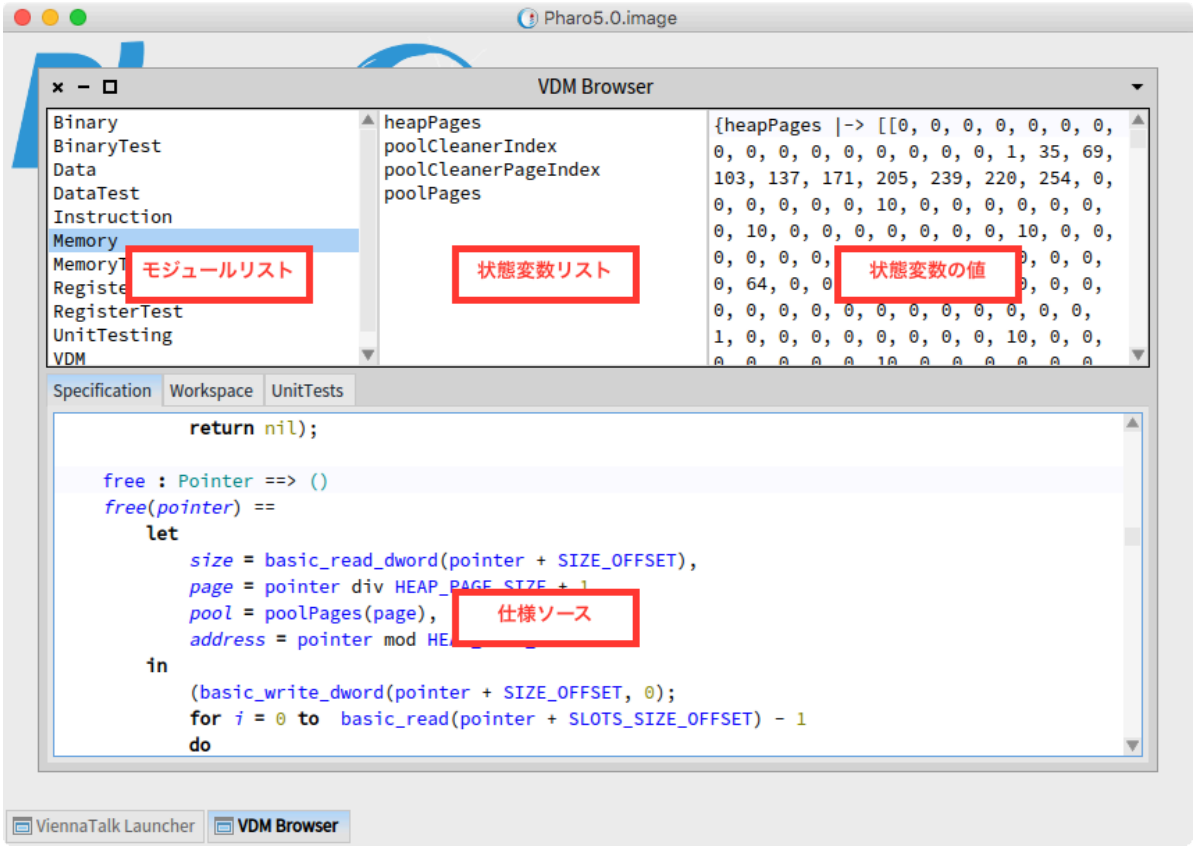


図 2. ViennaTalk 環境の画面例

るだけでなく，記述中の仕様のアニメーションが常に実行中であることが挙げられる．アニメーションとは，実行可能な仕様をインタプリタもしくは自動コード生成によって，当該仕様を実装したシステムの動作の模擬実行を指す．すなわち，一般的な開発環境でのテキストエディタの多くがテキストファイルを編集操作の対象としているのに対して，VDMBrowser が編集操作をする対象は VDM-SL 仕様を実装したものとして模擬実行している仮説的なシステムであり，仕様の編集とは模擬実行中のシステムのモデルを変更することを意味する．仕様記述者は VDMBrowser 上で模擬実行中のシステムの状態を問い合わせたり，状態を編集したり，特定の操作を模擬実行することができる．そして，システムの動作の具体例を見ながら，新たな機能項目を定義したり，既存の機能の定義を変更することができる．

図 2 に VDMBrowser の画面例を示す．VDMBrowser の画面は大きく上段と下段に分けられる．上段には左からモジュールリスト，状態変数リストおよび状態変数の値が表示される．VDM-SL の仕様はモジュール化されており，模擬実行中のシステムのうちどのモジュールに対してモデリングするかをモジュールリストによって選択する．上段中央の状態変数リストでは，システムの状態を表現した変数の一覧が表示され，ここで選択された変数の値が上段右側のエディタ上に表示される．上段右側の状態変数の値を編集し保存することで，模擬実行中のシステムの状態を変更することができる．これは VDM の他の統合開発環境である VDMTools[8] や Overture tool[9] にはない機能である．

下段には Specification，Workspace および UnitTests の 3 つのタブがあり，切り替えて使用する．Specification はモジュールリストで選択されたモジュールのソースを編集するエディタで，構文ハイライト，ソースの自動整形，表現式の評価実行などを行うことができる．Workspace はモジュールとは独立したテキストエディタで，探索的テストを行なったり，システムの利用シナリオを記述するために利用する．Workspace 上には任意のテキストを入力することができ，その一部を選択して VDM-SL 表現式として評価実行することができる．UnitTests は ViennaUnit によるテストの結果を一覧表示する．ViennaUnit については，第 3 節で説明する．

3. ViennaUnit

仕様の誤りは 2 つの試行錯誤のいずれにおいても発生し，いずれにおいても発見され得る．コミュニケーションを介した試行錯誤では，仕様の誤りがあまりにも多いと，他の開発関係者からの有効なフィードバックを得ることが困難になる．したがって，個人作業での試行錯誤の中で，仕様中の誤りを発見し修正することが求められる．

プログラミングにおいては，xUnit[10] をはじめとする自動テストによる誤りの発見が有効であり，実践されている．xUnit によるテスト駆動開発では，テストプログラムを一種の仕様と見做して，テストに合格するプログラムを記述することを直近の目標とする．ただし，xUnit でのテストプログラムはプログラム実行の例示であり，VDM-SL による系統的に抽象された仕様とは異なる．実行可能なサブセットを持つ VDM-SL でも，xUnit と同種の自動テストのためのツールが複数提供されている．VDM-SL 仕様にはプログラムの機能に関する表明が系統的に記述されることから，プログラミング言語での xUnit 用に記述するテストコードに記述される表明の多くは，既に仕様中に記述されていることが多い．VDM-SL 仕様がプログラムが持つべき性質を抽象し系統的に記述したものであるのに対して，VDM-SL 仕様に対するテストはそれらの性質に対する具体例を示して一貫性を確認する作業である．テストで確認すべき具体例は，仕様を模索する探索的仕様記述の段階では顧客やドメイン専門家の理解に基づいた特徴を表すことが求められ，高品質な仕様を定義する精緻化では確認の漏れがないよう機能定義に対して網羅的であることが求められる．

xUnit と同種のテストフレームワークに vdmUnit がある．vdmUnit は実際には，2 つのテストフレームワークが存在する．1 つは VDMTools[8] 上で動作するテストフレームワーク [11] であるが，VDM のオブジェクト指向拡張である VDM++が対象であり，VDM-SL はサポートされていない．もう 1 つは Overture tool[9] に付属するライブラリ [12] で，Java コード自動生成器と連携して JUnit 用のソースコードを出力する．Java コードの自動生成と Java プログラムのビルドおよび JUnit 実行というステップを踏まなければならず，頻繁な自動テスト実行による早期の誤り発見には向かない．

VDM で利用可能なテスト技法に，組み合わせテストがある．大量の組み合わせを効率よく実行するため，仕

様の各機能項目について，未定義な組み合わせがないか，条件に漏れがないかを確認することができる．しかし，VDM での組み合わせテストの目的は機能項目での未定義な組み合わせや条件漏れの発見が有効であるのは探索的仕様記述の後工程である精緻化であり，探索的仕様記述の段階ではまだ機能項目の定義における網羅性は必須ではない．また，組み合わせテストでは大量の組み合わせについてアニメーション実行を行うため，大きな計算リソースが求められる．

本稿で提案する ViennaUnit は，VDM-SL 向けの xUnit の簡略な実装である．ViennaUnit は，探索的仕様記述での利用を想定し，頻繁な仕様記述の変更に対して高いアジリティで追従して早期の誤り発見を可能にすることを目的としている．通常のプログラミング言語での xUnit では多くの表明が記述される．VDM-SL では仕様に不変条件，事前条件および事後条件などの多くの表明を持つ．ViennaUnit で記述する表明は主に仕様が要求されたシナリオを遂行可能であるかの確認であり，シナリオで約束した特性を満たすことの確認事項である．ViennaUnit で記述される表明の多くは仕様中に記述された表明と重複して，シナリオの粒度での意図の記述であることから，ViennaUnit での自動テストは表明のダブルチェックと見なすことができる．

ViennaUnit の実行メカニズムは非常に単純に作られている．ViennaUnit は構文的には通常の VDM-SL 仕様と同じ文法によって記述される．仕様が定義するモジュールのうち，モジュール名が `Test` で終わるものをテストモジュールとみなし，テストモジュールが定義する操作のうち，操作名が `test` で始まるものをテスト操作とする．

ViennaUnit を実行すると，対象となる仕様から全てのテスト操作を列挙し，それぞれアニメーション実行する．テスト操作を実行中に `AssertFailure` 型 または `AssertEqualsFailure` 型の例外が発生したらテスト結果は「失敗」となる，テスト操作を実行中に VDM-SL の処理系によって実行時エラーが発生したらテスト結果は「エラー」となる．テスト操作が正常終了した場合には「成功」と判定される．

ViennaTalk では，テスト操作を記述するための補助となるモジュールとして，`UnitTesting` モジュールを提供している．図 3 に `UnitTesting` モジュールの VDM-SL ソースを示す．`UnitTesting` モジュールは `assert` および `assertEquals` の 2 つの操作を提

```
module UnitTesting
exports all
definitions
types
     AssertFailure :: msg : seq of char;
     AssertEqualsFailure ::
         actual : ?
         expected : ?
         msg : seq of char;

operations
    assert : bool * seq of char ==> ()
    assert(b, msg) ==
        if not b
        then exit mk_AssertFailure(msg);

    assertEquals : ? * ? * seq of char ==> ()
    assertEquals(actual, expected, msg) ==
        if
            actual <> expected
        then
            exit mk_AssertEqualsFailure(
                actual,
                expected,
                msg);

end UnitTesting
```

図 3. UnitTesting モジュールのソース

供する．`assert` 操作は引数として真偽値およびメッセージを取り，渡された真偽値が `false` であった場合に `AssertFailure` 型の例外を発生させる．メッセージは `AssertFailure` 型の値の中に格納される．`assertEquals` 操作は引数として 2 つの値とメッセージを取る．2 つの値は任意の型で良く，等しくない場合には `AssertEqualsFailure` 型の例外を発生させる．2 つの値およびメッセージが `AssertEqualsFailure` 型の値の中に格納される．`assertEquals(x, y, msg)` は `assert(x = y, msg` と記述することも可能だが，`assertEquals` 操作を用いることで失敗時に `x` と `y` の具体的な値を例外に格納してデバッグのための情報として利用することができる．

図 4 にテストモジュールの例を示す．仮想機械の開発のために記述された VDM-SL 仕様から抜粋した．図 4 の `MemoryTest` モジュールは，`UnitTesting` モジュールを使って，`Memory` モジュールに対する 2 つのテスト操作を定義している．`Memory` モジュールは仮想機械のメモリモデルを定義するモジュールであり，メモリアライメントやデータの読み書きを規定する．`test_align` 操作では，メモリアライメントに対するテストとして，サ

```
module MemoryTest
imports
    from Memory all,
    from UnitTesting
        operations
            assertEquals renamed assertEquals;
            assert renamed assert;
exports all
definitions
operations
    test_align : () ==> ()
    test_align() ==
        (for i = 1 to  Memory`ALIGNMENT
        do assertEquals(
            Memory`align(i),
            Memory`ALIGNMENT,
            "0");
        for i = Memory`ALIGNMENT + 1
        to  Memory`ALIGNMENT  * 2
        do assertEquals(
            Memory`align(i),
            Memory`ALIGNMENT  * 2,
            "1"));

    test_basic_read : () ==> ()
    test_basic_read() ==
        (Memory`addPage();
        Memory`basic_write(
            0x12,
            0xfedcba9876543210);
        assertEquals(
            Memory`basic_read(0x12),
            0xfedcba9876543210,
            "basic_read 64bits immediate");
        Memory`basic_write_byte(0x10, 0x01);
        Memory`basic_write_byte(0x11, 0x23);
        Memory`basic_write_byte(0x12, 0x45);
        Memory`basic_write_byte(0x13, 0x67);
        Memory`basic_write_byte(0x14, 0x89);
        Memory`basic_write_byte(0x15, 0xab);
        Memory`basic_write_byte(0x16, 0xcd);
        Memory`basic_write_byte(0x17, 0xef);
        assertEquals(
            Memory`basic_read(0x10),
            0xefcdab8967452301,
            "basic_read little endian"));

end MemoryTest
```

図 4. ViennaUnit で記述したテストモジュールの例

イズ 1 からアライメントサイズまでは，アライメントサイズまで拡張され，アライメントサイズより大きくアライメントサイズの 2 倍未満まではアライメントサイズの 2 倍に調整されることを確認する．test_basic_read 操作では，メモリページ上への 64 ビット整数の読み書きの結果が整合することを確認し，バイト単位で書き込んで 64 ビット整数として読むとリトルエンディアンとして解釈されることを確認する．

VDM-SL 仕様に対するユニットテストでは，test_align 操作のようにどのような結果をもたらす定義がされるべきか，あるいは，test_basic_read 操作のようにどのようなシナリオを受け付けてどのような結果をもたらす操作群が定義されるべきかをテストコードという形で規定する．例えば，test_basic_read 操作は Memory モジュールの basic_read 操作がメモリページ中の連続した 8 バイトをリトルエンディアンで 64 ビット符号なし整数として解釈することを確認する．Memory モジュールの basic_test 操作の定義においてもリトルエンディアンであることは規定されているが，アセンブリ言語プログラマの要請として，1 バイト単位で書き込む basic_write_byte 操作との組み合わせによってエンディアンを明確に示し，basic_read 操作の定義がシナリオの中でそれに適合していることを確認することが test_basic_read 操作の目的である．

VDM-SL 仕様に対するユニットテストは，仕様記述者やドメイン専門家の理解と機能モデルの間の整合性を確認する作業であると言える．すなわち，ユニットテストの失敗を早期に検出することが，仕様記述者の理解と仕様記述内容の乖離を早期に発見し，正しい理解および正しい機能定義に導くことにつながる．

4. VDMBrowser における ViennaUnit の実行とフィードバック

ViennaTalk の VDMBrowser は探索的仕様記述での試行錯誤を伴う仕様記述を支援する．したがって，VDM-Browser が扱う VDM-SL 仕様には誤りが頻繁に混入する．これは試行錯誤による編集の頻度の高さに加え，仕様自体がまだ未成熟であること，仕様記述者の対象への理解が正確ではないことに起因する．仕様の誤りを早期に発見して修正するためには，高い頻度でのユニットテストの実行が有効であると考えられる．

VDMBrowser では，仕様の修正があるたびに自動的に

バックグラウンドで ViennaUnit を起動し，全てのテストモジュールの全てのテスト操作を実行する．ただし，自動実行するかどうか，および，全てのテストモジュールを対象とするかまたは修正に直接関係するテストモジュールのみ実行するかは，設定により選択することができる．また，モジュールリストのメニューからユーザーの指示によって，全てのテストモジュールまたはモジュールリストで選択されたモジュールに対応するテストモジュールを実行する．本稿では，全てのテストモジュールを自動実行した場合について議論する．

ViennaUnit の実行はバックグラウンドで行われる．また，テスト実行の状態空間は，VDMBrowser 上のアニメーションの状態空間とは独立している．テスト実行中も VDMBrowser 上で仕様を編集したり，あるいは，VDMBrowser 上のアニメーションの実行を行うことができる．

ViennaUnit によるテスト結果は VDMBrowser 上の UnitTests タブに切り替えることで確認することができる．図 5 に ViennaUnit の結果の画面例を示す．リストの各行に，それぞれ 1 つのテスト操作の結果が表示される．成功したテスト操作は，緑色の丸アイコンに続いてテストモジュール名，テスト操作名および OK というメッセージで示される．表明に失敗したテスト操作は，黄色の丸アイコンに続いてテストモジュール名，テスト操作名およびエラーメッセージで示される．`assertEquals` に失敗した場合には，具体的に一致しなかった値のペアが示される．エラー終了したテスト操作は，赤色の丸アイコンに続いてテストモジュール名，テスト操作名およびエラーメッセージで示される．不変条件，事前条件，事後条件などのテスト対象の仕様中での表明失敗もエラーとして扱われる．

ViennaUnit は，失敗またはエラーとなるテスト操作を発見すると，ViennaTalk 環境の周辺領域に，仕様記述タスクの邪魔をせずにテストの失敗を知らせる通知を表示する．図 6 に通知の例を示す．この通知はウィンドウシステムのフォーカスを奪わないため，ユーザは VDMBrowser などでの編集作業を継続することができる．また，通知は半透明で表示され，数秒で透明度が高くなり消滅することで，ユーザへの干渉を小さく抑えている．

5. 評価

VDM-SL を用いた仕様記述を行っている開発作業において，ユニットテストの結果をログに記録するよう改造した VDMBrowser を 4 時間使用して，ViennaUnit の自動実行の有効性を評価した．改造した VDMBrowser は，仕様が保存されるたびに，自動的にバックグラウンドで ViennaUnit を実行し，結果をユーザには表示せずにログに記録した．ユーザは，ユニットテストの実行が必要だと判断した時に明示的に ViennaUnit を実行し，明示的な ViennaUnit の実行であることをログに記録した上で，そのテスト結果をログに記録した．

表 1 に，自動的なテスト実行による誤り発見から明示的なテスト実行による誤り発見までの差を示す．テストの失敗は，まず自動的なテスト実行により記録され，その後，ユーザによる明示的な実行によりユーザに失敗が示される．自動的なテスト実行によるテスト失敗の記録からユーザによる明示的なテスト実行によるテスト失敗までの編集回数を，自動的なテスト実行による早期発見の効果として測定する．自動的なテスト実行による同一のテスト失敗が n 回繰り返された後でユーザによる明示的なテスト実行によりそのテスト失敗がユーザに示された場合，ユーザは誤りの存在を知らないまま $n-1$ 回の編集作業を行なったことになる．これは，自動的なテスト実行はユーザによる明示的なテスト実行より $n-1$ 回の編集作業分，早期に誤りを発見できることを意味する．

テスト失敗には，仕様の誤りによるものと，テストの誤りによるものがある．表 1 では，誤りの所在として示した．自動的なテスト実行によって発見された仕様の誤りが 5 つのうち，3 つの仕様の誤りについて，ユーザーによるテスト実行による発見が遅れた．特に誤り 12 については，自動的なテストの実行から 9 回分，ユーザーによるテスト実行による発見が遅れた．この時ユーザーは編集対象のモジュールのテストを毎回実行していたが，そのモジュールを利用している別のモジュールのユニットテストでエラーが発生していた．自動的に全テストモジュールを実行していれば，実際よりも修正 9 回分だけ早期にそのエラーを認識することができた可能性がある．

誤り 9 は，ユーザーの操作ミスによって，意図しないモジュールに対して意図しない編集が保存されてしまったために発生した．そのためユーザーがそのモジュールに対するテスト実行の必要性を認識せず，一通りの作業が終了した後に念のために実行した全テストモジュール

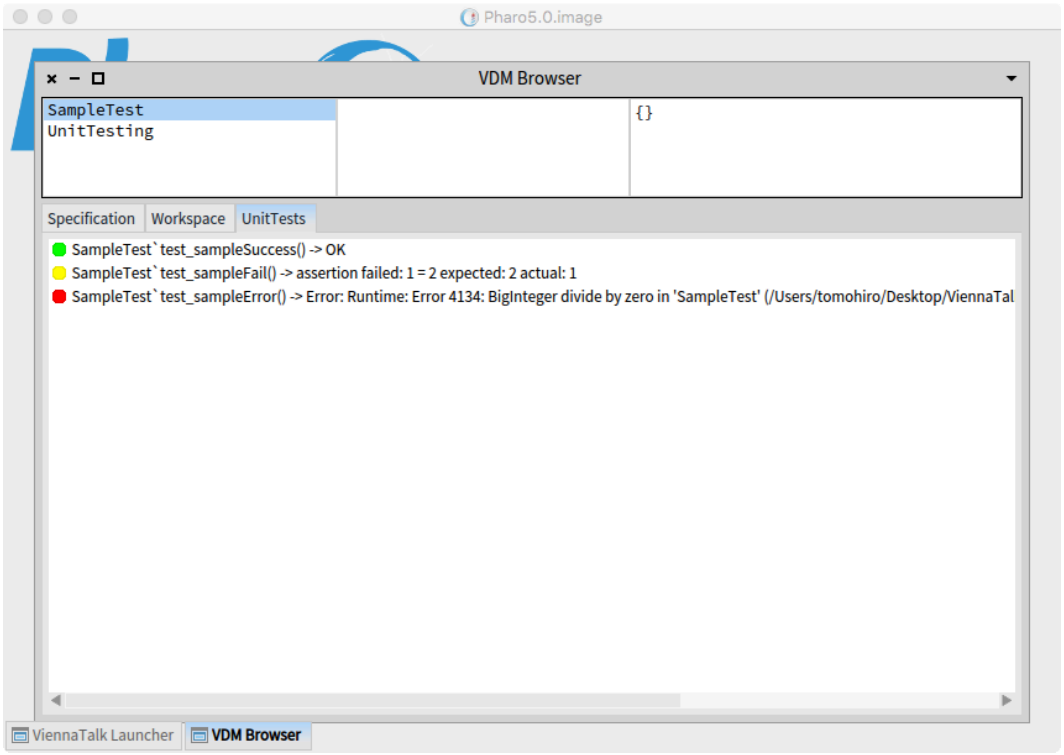


図 5. VDMBrowser でのテスト結果一覧の表示例

表 1. 自動的なテスト実行と明示的なテスト実行による誤り発見時期の差

誤り	失敗またはエラー	誤りの所在	発見時期の差	誤りの概要
誤り 1	失敗	テストの誤り	0	テストデータの誤り
誤り 2	エラー	テストの誤り	0	テストデータの誤り
誤り 3	失敗	仕様の誤り	0	操作中の処理の欠落
誤り 4	エラー	テストの誤り	0	テスト操作中での処理の欠落
誤り 5	失敗	テストの誤り	0	表明の参照先の誤り
誤り 6	エラー	テストの誤り	0	テストデータの誤り
誤り 7	エラー	仕様の誤り	0	場合分けの不足
誤り 8	エラー	仕様の誤り	0	場合分けの不足
誤り 9	エラー	仕様の誤り	4	場合分けの不足
誤り 10	エラー	テストの誤り	1	import 宣言の誤り
誤り 11	エラー	テストの誤り	0	テスト操作中での処理の欠落
誤り 12	エラー	仕様の誤り	9	場合分けの不足
誤り 13	エラー	テストの誤り	0	テスト操作中での処理の欠落

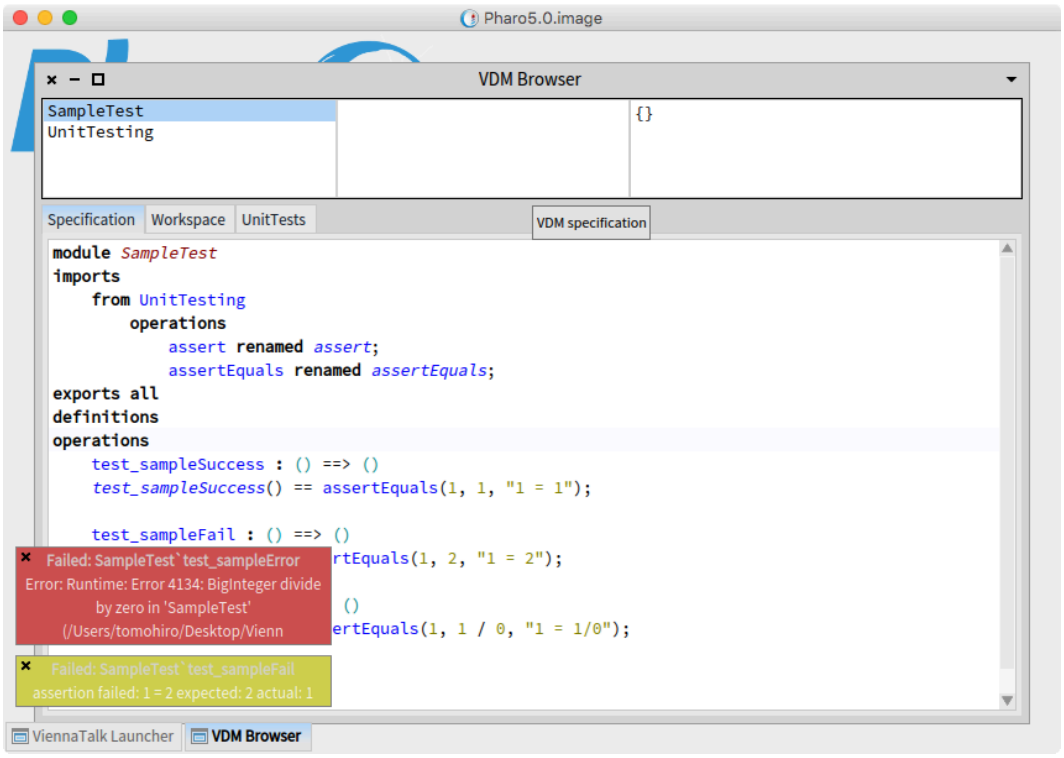


図 6. ViennaTalk におけるテスト失敗通知の表示例

実行によって誤りを発見した．誤り 10 は，テスト操作を 2 つ連続して記述する作業で，1 つ目のテスト操作を保存し，それに誤りがあったところを，そのまま 2 つ目のテスト操作の記述を開始したことにより，誤りの発見が 1 回分遅くなった．しかし，2 つ目のテスト操作を保存した後でそのテストモジュールを実行したため，1 回分の遅れのみで比較的早期に発見することができた．

以上より，ViennaUnit の自動的な実行は，仕様の誤りの早期発見に有効であると考えられる．一方，テストの誤りに関しては，必ずしも自動的な実行をしなくても，開発環境からユニットテストを明示的に実行することで早期発見された．これはテストがモジュール間の独立性が高いことが原因と考えられる．仕様を編集した場合には，編集の影響がモジュールをまたがって及ぶため，編集したモジュールのみをテストしても誤りを見逃すことがある．テストは仕様と比較してテストモジュール間の独立性が高いために，編集したテストのみを即座に実行することで，自動的なテスト実行にほとんど遅れることなく発見されたと考えられる．

6. 議論

ViennaTalk は探索的仕様記述を支援するための環境であり，本稿で紹介したテストフレームワークも探索的仕様記述での試行錯誤の中でより早期に誤りを発見し修正することを目的としている．テストフレームワーク ViennaUnit は xUnit から派生したテストフレームワークの簡易な実現であり，共通の機能が多い．形式仕様に対するユニットテストとプログラムコードに対するユニットテストには技術的には多くが共通しているが，目的は大きく異なっている．テスト駆動開発では，テストコードはプログラムコードの仕様として扱われる．一方で，形式仕様に対するユニットテストはテスト対象の仕様ではない．仕様がテスト対象であり，テストは仕様が定義する機能に関するシナリオを通して仕様記述者やドメイン専門家の理解と機能モデルの間の整合性を確認するための記述である．ViennaUnit が自動的なテスト実行により早期に誤りを発見することで，仕様記述者は記述内容と理解の不整合を認識することができる．

VDMBrowser によるテストの自動実行は，ユーザーが

仕様記述に集中することを助ける点でも効果的である．テスト実行の処理時間はテスト操作の定義に依存するが，第 5 節で評価した仕様およびテストでは，52 個のテスト操作の実行に約 30 秒かかった．仕様のモジュールまたはテストモジュールを編集するたびにメニューからテスト実行を選択して 30 秒待つことは，ユーザーである仕様記述者の集中を損ねる．テストの自動実行および邪魔にならないテスト失敗通知により，編集するたびに 30 秒待つことなく，仕様記述を継続することができる．

VDMBrowser でのテスト実行の課題としては，後工程との連携が挙げられる．仕様に対するテストの目標は，仕様テストの失敗やエラーが発生しなくなることではない．仕様として記述した内容が仕様記述者やドメイン専門家の理解と齟齬がないことを確認し，適切な仕様を記述することで後工程での精緻化により厳密な仕様を作成し，品質の高いソフトウエアの実装を出荷することが目標である．そのためには，実装に対するテストやデバッグにおいて，仕様に対するユニットテストのテストモジュールの定義やテスト結果が後工程で有効に利用可能であることが望ましい．

7. まとめ

探索的仕様記述を支援するための仕様記述環境 ViennaTalk でのテストフレームワーク ViennaUnit を説明し，評価した．ViennaTalk は探索的仕様記述を前提とした記述環境であるが，最終的な目的は実装されたソフトウェアの生産性や品質の向上である．そのためには，探索的仕様記述の後工程である精緻化や実装での開発手法やそのためのツールとの連携が求められる．今後は，精緻化や実装との連携を中心に，ViennaTalk に求められる機能やツール連携を分析し，開発を継続する．

参考文献

[1] J. Woodcock, P. G. Larsen, J. Bicarregui and J.. Fitzgerald, “Formal Methods: Practice and Experience”, *ACM Computing Surveys*, Vol. 41, No. 4, pp. 1–36, 2009.

[2] N. Ubayashi, S. Nakajima and M. Hirayama “Context-dependent product line engineering with lightweight formal approaches”, *Science of Computer Programming*, Vol. 78, Issue 12, pp. 2331–2346, 2012.

[3] J. Fitzgerald and P. G. Larsen “Modelling Systems – Practical Tools and Techniques in Software Development”, Cambridge University Press, 1998.

[4] K. Lausdahl, P. G. Larsen and N. Battle “A Deterministic Interpreter Simulating A Distributed real time system using VDM”, *Proceedings of the 13th international conference on Formal methods and software engineering*, pp. 179–194, 2011.

[5] T. Kurita and Y. Nakatsugawa “The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip”, *Intl. Journal of Software and Informatics*, Vol. 3, No. 2-3, pp. 343–355, 2009.

[6] 小田朋宏, 荒木啓二郎 “形式仕様工程の初期段階に着目した統合仕様記述環境 ViennaTalk”, コンピュータソフトウェア, 34 巻, 4 号, ppp. 4_129-4_143, 日本ソフトウェア科学会, 2017.

[7] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, Damien and M. Denker “Pharo by Example”, Square Bracket Associates, 2009.

[8] J. Fitzgerald, P. G. Larsen and S. Sahara “VDMTools: advances in support for formal modeling in VDM”, *ACM Sigplan Notices*, Vol. 43, No. 2, pp. 3–11, 2008.

[9] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl and M. Verhoef “The Overture Initiative – Integrating Tools for VDM”, *SIGSOFT Softw. Eng. Notes*, Vol. 32, No. 1, pp. 1–6, 2010.

[10] P. Hamill “Unit Test Frameworks”, O’Reilly, 2004.

[11] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat and M. Verhoef “Validated Designs For Object-oriented Systems”, Springer, 2005.

[12] P. G. Larsen, K. Lausdahl, P. W. V. Tran-Jørgensen, A. Ribeiro, S. Wolff and N. Battle “Overture VDM-10 Tool Support: User Guide”, The Overture Initiative, TR-2010-02, 2010.

SOFL 形式仕様に基づくC#プログラムのテストツール

網谷　拓海
法政大学情報科学研究科
takumi.amitani.2c@stu.hosei.ac.jp

劉　少英
法政大学情報科学研究科
sliu@hosei.ac.jp

要旨

形式仕様をプログラムの実装だけでなくテストにも利用できれば，ソフトウェアの開発コストを削減でき，形式仕様の有用性も高まる．具体的には，形式仕様に記述された入力に関する条件をテストケースの生成に，出力に関する条件はテスト結果の評価に利用することができる．先行研究ではこの特性を生かし，SOFL 形式仕様を使ってテストケースの生成とテスト結果を評価するプログラムが作成された．しかし，生成したテストケースをテスト対象プログラムに入力するプログラムの作成と，テストの実行，テスト結果を評価プログラムに入力することはすべて手動で行い，別のツールを使用する必要があった．そこで本研究では，SOFL 形式仕様に基いて実装された C#プログラムのテストケース生成，テスト用プログラムの生成と実行，テスト結果の評価までの一連の作業をすべて行えるツールを考案，実装し，実際の事例に適用可能なことを確認した．

1. はじめに

ソフトウェア開発プロジェクトにおいてテストの占める割合は新規開発の中で結合テストと総合テストを合わせると全行程の約30%を占める[1]．よってテストの割合は大きく，テスト時間の削減はプロジェクト全体のコスト削減につながる．

プログラムのテストを行う場合，通常はテストケースの生成と，テストの実行，テスト結果の評価は手動で行う必要がある．しかしプログラムに対する要求を形式仕様であらかじめ記述することにより，要求が数学的，論理的に表現され，テストケースの生成とテスト結果の評価の一部を自動化することができる．先行研究[2]ではこれを生かしてSOFL 形式仕様[3]を用いてテストケースの生成とテスト結果の評価を行うプログラムが作成された．しかしこのプログラムでは生成されたテストケースをテスト対象プログラムに入力してテストを行うことと，テスト結果を評価プログラムに入力することは手動で行い，別のツールを用意する必要がある．

そこで本研究ではテストケース生成とテスト結果評価を先行研究のプログラム[2]を利用して行い，加えて，テストケースをテスト対象プログラムに入力するプログラムであるテストスクリプトを生成して実行することと，テスト結果を評価プログラムに入力して評価を行うという，テストに必要な一連の作業を一つのツールで行えるようにした．

2 章では SOFL 形式仕様の説明，3 章から 7 章では，ツールの説明，8 章では実際の事例にツールを適用できるか確認し，9 章ではまとめと今後の課題を記す．

2. SOFL 形式仕様について

SOFL 形式仕様では一つの仕様を C#や Java のクラスと対応する，モジュールの中に記述する．モジュール内には型定義，データストア変数の定義 C#や Java のメソッドと対応するプロセス定義が記述される．プログラムに対する条件式はプロセスの事前条件と事後条件として記述される．この事前条件と事後条件にはメソッドの入力と出力が満たすべき条件が数学的，論理的に記述されるため，テストケース生成とテスト結果評価に利用できる．

SOFL は先行研究[4]で編集支援ツールが開発されており，そのツールで記述されたモジュールは fModule ファイルとして保存される．本研究で実装するツールではこの fModule ファイルを読み込むことで仕様の情報を取得する．

3. ツールの目的と概要

従来 SOFL 形式仕様を用いたテストを行う場合，テストケース生成からテスト結果評価までの流れを別々の場（ソフトウェアなど）で行う必要があった．そこでそれらの工程を同一ソフトウェア上で行えるようにし，テストの効率を上げて時間を削減することがツールの目的である．

本ツールは SOFL 形式仕様から実装された C#プログラムのテストをサポートする．主な機能は，テストケース生成，テストスクリプト生成，テストスクリプト実行，テスト結果

評価の4つである．ツールはWindowsフォームアプリケーションを C#で実装した．ツールの実行イメージは以下の図の通りである．

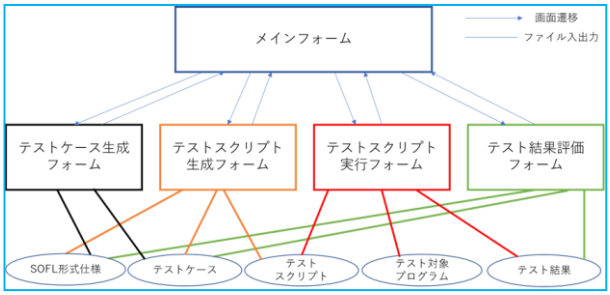


図 1：ツールの概要

各機能を持ったフォームへはメインフォームを経由して遷移することができる．ユーザーはテストケース生成フォームから順に進めていくことでテストを完了することができるようになっている．このツールの特徴として，各フォームでの実行結果を外部ファイルに保存することで，各機能ごとに作業を中断したり，再開することができる．例えばテストケース生成フォームでテストケースを生成した後にツールを閉じてしまっても，生成したテストケースはファイルに保存しているので，またツールを起動してテストスクリプト生成から始めることができる．メインフォームは以下の図のとおりである．

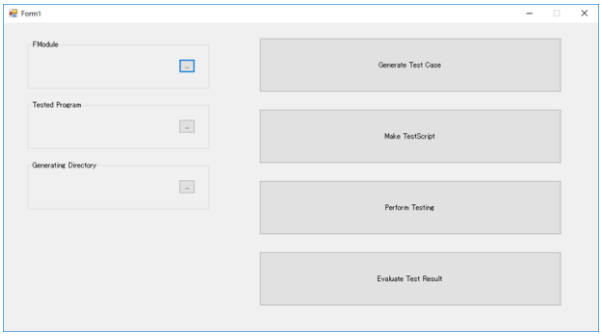


図 2：メインフォーム

まず各機能を使う前に，ファイルパスを指定する必要がある．左側の一番上の「FModule」欄のボタンを押すとフォルダブラウザが表示されるので，そこで SOFL 形式仕様のモジュールファイルである fModule ファイルを指定する．同様にして「Tested Program」欄ではテスト対象プログラムをまとめた dll ファイル（C#プログラムをライブラリ形式でまとめたファイル）を指定する．「Generating Directory」欄ではツールによって生成されるテストケース，テストスクリプト，テスト結果を保存するフォルダを指定する．

4. テストケース生成

テストケース生成フォームを使用する前に，SOFL 形式仕様のモジュールである fModule ファイルを指定する必要がある．ツールは指定された fModule ファイルを解析し，モジュールに含まれる各プロセスに対してテストケースを生成する．テストケース生成後のフォームの様子は以下の図の通りである．

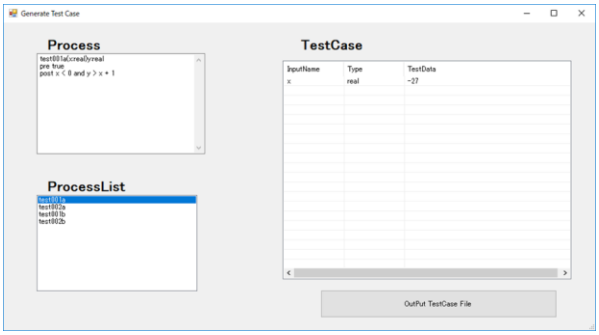


図 3：テストケース生成フォーム

テストケース生成後，まずモジュールに含まれる全プロセスが ProcessList ボックスの中にリストアップされる．ユーザーがリストからプロセス名をクリックすると Process ボックス内にプロセスの記述が，TestCase ボックスにはそのプロセスに対して生成されたテストケースが表示される．これによってユーザーは，各プロセスがどのようなプロセスであったか，さらにそのプロセスに対してどのようなテストケースが生成されたのかを確認することができる．もしテストケースを自分で指定したい場合は TestCase ボックスの TestData の項目をクリックすることで，入力することができる．そしてすべてのプロセスに対して正しいテストケースが準備できたと思ったら右下の「OutPut TestCase File」というボタンをクリックすることでテストケースをテキストファイルとして保存することができる．

テストケース生成に使用しているプログラムは先行研究の[2]のものである．このプログラムではプロセスに記述された，プロセスの入力に関する条件式をシナリオごとに分けて，さらに各シナリオの条件式を逆ポーランド記法に変換してテストケースを生成する．例えば以下のようなプロセスを考える．

```
Process Test (x : int) y : int
pre x > 0
post x > 10 and y = 5 or
x <= 10 and y = 20
end_process;
```

このプロセスの入力は int 型の x である. x に対する条件は, 事前条件の x > 0, そして事後条件の x > 10 and y = 5 or x <= 10 and y = 20 である. 事後条件は x の値によって二つのシナリオに分けられる. 一つは x > 10 and y = 5, もう一つは x < = 10 and y = 20 である. ここから出力の条件を省き, 事前条件はどちらのシナリオも満たさなければならないので加えると, x > 0 and x > 10 と x > 0 and x <= 10 という入力に関する二つの条件を得ることができ, テストデータは各条件に対して一つずつ生成されて, それらをまとめてテストケースとする. なおここで評価することは, 事前条件を満たすテストケースを入力したときの出力の値であるため, 事前条件を満たさない場合のテストケースは生成しない.

5. テストスクリプト生成

テストスクリプトはテスト対象プログラムをテストするためのプログラムで, テスト対象プログラムと同じプログラミング言語で記述される. 本ツールは C#のプログラムをテストすることを想定しているので, テストスクリプトも C#プログラムとして生成する.

テストスクリプトはテストメソッド, クラススクリプト, テスト詳細メソッド, テスト結果書き出しメソッド, Main メソッドで構成される. 各メソッドの説明を以下に記す.

5.1. テストメソッド

テストメソッドは各プロセスに対して生成され, 各プロセスに対応したテスト対象メソッドをテストする. テストメソッド内には, 各テストシナリオに対してテストケースを変数に格納する文, テスト詳細メソッド実行文が記述される. あるプロセスに二つのテストシナリオがあった場合, そのプロセスのテストメソッドには, 一つ目のテストシナリオのテストケースを格納するための変数宣言文, その変数にテストケースを格納する文, その変数をテスト詳細メソッドに入力して実行する文があり, 二つ目のテストシナリオに対しても同じように生成される. このとき変数に格納するテストケースは, テストケース生成フォームで保存したテストケースファイルから取得する. ちなみにテストメソッド内ではテストケースを格納する変数の型は, プロセスの入力の SOFL 型と一意に対応している. 各 SOFL 型と C#の型の対応は以下の表のとおりである.

表 1 : SOFL 型と C#型の対応

SOFL 型	C#型
nat, nat0, int	int
real	double
char	char
bool	bool
string, enum	string
Map	Dictionary
Set	HashSet
Sequence	List
Composite, Product	SOFL 型と同名のクラス

上の表のうち, Composite と Product 型は複数の要素を持つ混合型で, モジュールの型定義内で新たな型として定義される. そのため, C#には対応する型が存在しないので, テストケースを格納するために対応するクラスをテストスクリプト内に生成する. それをここではクラススクリプトと呼ぶ.

5.2. クラススクリプト

SOFL モジュール内に Composite 型か Product 型の新たな型が定義されていた場合, 新たなクラスを用意する. クラス名は定義された型の名前である. クラスのフィールドには Composite もしくは Product の要素に対応したものが作られる. さらにクラススクリプトはコンストラクタを持ち, 各フィールドにテストケースを代入できる.

5.3. テスト詳細メソッド

テストメソッドではテストケースを変数に格納する際に, SOFL 型と C#の型の対応を表 1 のように決めたが, 実際にはテスト対象プログラムの入力が別の型で実装されている可能性がある. そのため, テスト詳細メソッドはテストケースが収められた表 1 通りの C#型の変数を入力として受け取り, それをテスト対象プログラムの入力の実際の型に変換するプログラムをユーザーに記述してもらう. さらにテスト対象プログラムの実行文とテスト結果の取得もユーザーが記述する. これによって, どのような実装方法であっても, テストを行うことができる.

よってテスト詳細メソッドの構成要素は, テストケースをテスト対象プログラムと正しく対応した型に変換する文, テスト対象プログラムを実行する文, テスト結果を回収す

る文である．なおテスト結果を回収する際には，SOFL 形式仕様と比較しやすくするために，プロセスの出力に対してテスト結果を格納する変数の型は表 1 通りにする．テスト結果を変数に格納したら，テスト詳細メソッドの最後にテスト結果書き出しメソッドを呼び出す．

5.4. テスト結果書き出しメソッド

テスト結果を入力として受け取り，それを外部ファイルに書き出すための文字列変数に追加する

5.5. Main メソッド

Main メソッドではまずテスト結果を保存するための文字列変数を作る．そして各プロセスのテストメソッドを呼び出してテストを行う．呼び出すごとにテスト結果用文字列にはテスト結果が追加されていく為，すべてのテストメソッドを呼び出した後，その文字列を外部テキストファイルに書きだす．

以上の要素で構成されたテストスクリプトの動作を以下の図にまとめた．

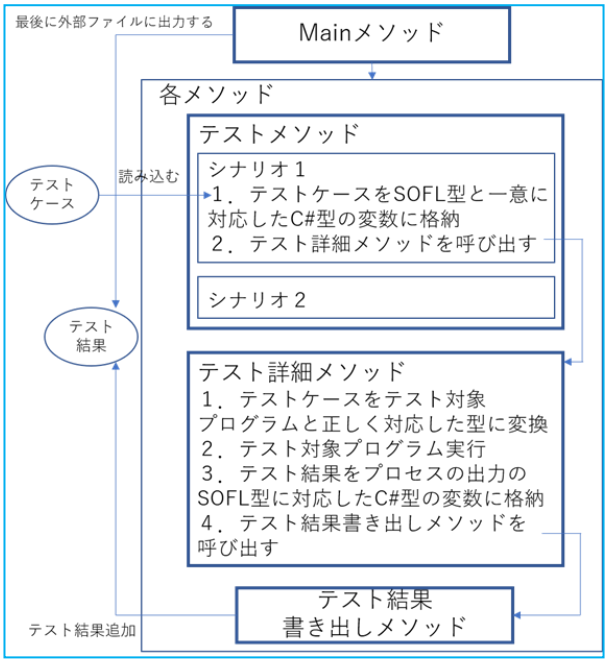


図 4：テストスクリプトの動作

テストスクリプト生成フォームでは，フォーム表示前にテストケースと SOFL 形式仕様を読み込んで，テスト詳細メソッド以外を自動生成する．フォーム出現後，フォーム内では各プロセスのテスト詳細メソッドを編集できる．その様子が以下の図である．

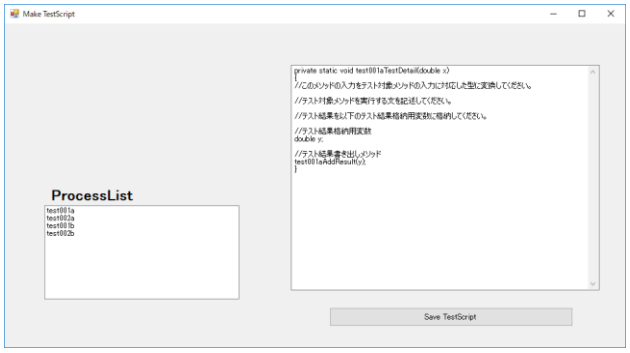


図 5：テストスクリプト生成フォーム

ProcessList ボックスにはモジュール内のプロセスがリストアップされる．そこからプロセスを選択すると，そのプロセスのテスト詳細メソッドが表示される．図のテスト詳細メソッドは自動生成直後のもので一切編集をしていない．テスト詳細メソッドにはユーザーに対する要望のコメントが記述されている．ユーザーはこれに従ってテスト詳細メソッドを完成させる．すべてのメソッドを完成させたら，右下の「Save TestScript」ボタンをクリックするとテストスクリプトを保存することができる．なおテスト詳細メソッドはテストスクリプトとは別のファイルにも保存しているので一度ツールを閉じてしまっても再度編集することができる．

6. テストスクリプト実行

このフォームを起動すると，テストスクリプト生成フォームで保存したテストスクリプトが読み込まれる．このとき左のテキストボックスにテストスクリプトの内容が表示されるが，実行前に編集可能である．以下の図がテストスクリプトを読み込んだ直後のフォームである．

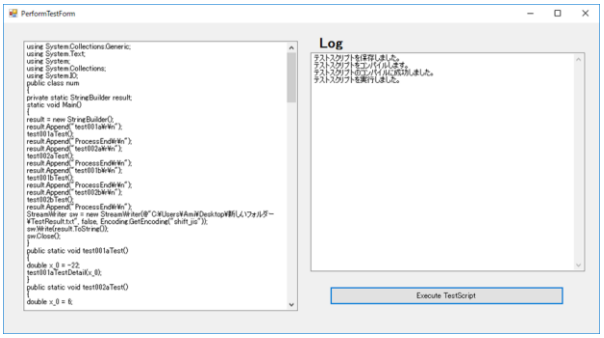


図 6：テストスクリプト実行フォーム

右下の「Execute TestScript」ボタンを押すとテストスクリプトが実行される．まずテストスクリプトが保存され，コンパ

イルが始まる．文法が正しければ「テストスクリプトのコンパイルに成功しました．」というメッセージが出るが，誤りがあれば，エラーメッセージと失敗メッセージが表示されて実行は中断される．この場合，エラーメッセージを参考にしてテストスクリプトを修正し，再度実行するとよい．コンパイルが成功すると実行可能ファイルが実行される．それが成功すると「テストスクリプトを実行しました．」という，メッセージとともにテストの実行が終了する．

テストスクリプト内にはテスト結果を外部ファイルに書き出す記述が含まれている．そのためこのフォームでテストスクリプトの実行が終了した際にテスト結果が記述されたテキストファイルが生成される．このファイルは次の章でSOFL 形式仕様の出力に関する条件と比較して評価する．

7. テスト結果評価

このフォームではテスト結果を形式仕様と比較して評価する．フォームは以下の図のとおりである．

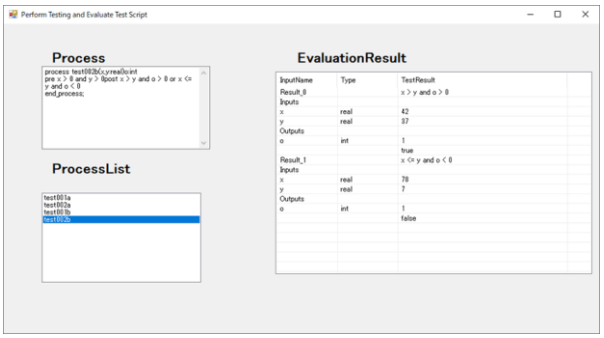


図 7：テスト結果評価フォーム

ProcessList にはモジュール内のプロセスがリストアップされ，プロセスを選択することで，評価結果を確認することができる．図のプロセスは二つのシナリオがあるので評価結果が二つある．一つ目のシナリオ Result_0 の条件は x > y and o > 0 だが，これを評価するためにまずはテストケースを条件に代入する．このテストケースはテストケース生成フォームで保存したファイルから取得する．この時条件は 42 > 37 and o > 0 となり，さらに出力 o が 1 なのでテスト結果は true となる．同様に Result_1 を確認すると 78 <= 7 and 1 < 0 となるので結果は false となる．

テスト結果の評価には[2]の先行研究のプログラムを利用している．このプログラムはまず SOFL 形式仕様を解析して，各プロセスの条件をテストシナリオごとに分ける．そして，条件とプロセスの入力（テストケース）と出力（テスト結果）を評価用メソッドに入力することでその条件の真偽値を得ることができる．よってこのフォームを使用する際は，各プロセスの条件を SOFL 形式仕様から，プロセスの入力をテストケースファイルから，出力をテスト結果ファイルから読み込む．

8. 実際の事例

今回は SOFL 形式仕様のモジュールを用意し，それを基に実装した C#プログラムをツールによってテストした．仕様とプログラムは以下の通りである．

```
module TestModule;
type
PayingCard = Composed of
name : string
buffer : nat0
end;
process Pay(c: PayingCard, price: int) payout: int
pre c.buffer > 100000 and price > 0
post payout = c.buffer - price
end_process;
end_module
```

```
namespace TestLibrary{
    public class TestModule{
        public static int Pay(Card card, int price){
            return card.GetBuffer() - price + 1;
        }
    }
    public class Card{
        private string name;
        private int buffer;
        public Card(string name, int buffer){
            this.name = name;
            this.buffer = buffer;
        }
        public string GetName(){
            return name;
        }
        public int GetBuffer(){
            return buffer;
        }
    }
}
```

モジュールには複数の要素を持つ複合型である PayingCard 型と，PayingCard 型の c と int 型の price を入力として受け取り，int 型の payout に c の buffer から price を引いた値を出力するという Pay プロセスが定義されている．

プログラムにはTestModuleクラスとCardクラスが実装されている．Cardクラスは PayingCard 型を実装したもので，string 型の name と int 型の buffer を持つ．また TestModule クラスは仕様通りに Pay メソッドを持つが，このメソッドが出力する値は意図的に入力 card の buffer から price を引いた値に 1 を足したものにした．つまりこのプログラムは誤りが含まれる．

次にツールに仕様とテスト対象プログラムを指定してテストを行った．まずは生成されたテストケースは以下の図の通りである．

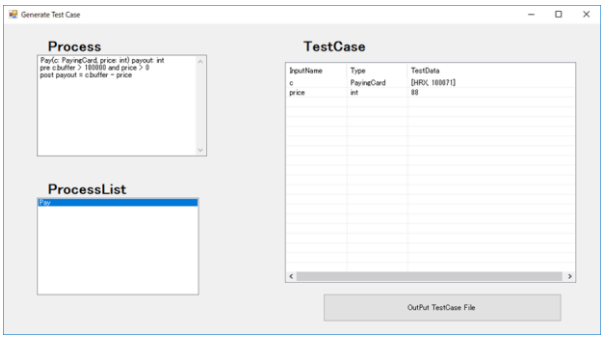


図 8：テストケース生成後

Pay プロセスの入力 c には[HRX, 100071]が，price には 88 がテストデータとして生成された．Pay プロセス内の条件で入力に関する条件は

c.buffer > 100000 and price > 0 となっているが，テストケースはこれを満たしている．

次に生成したテストケースと仕様を用いてテストスクリプトを生成した．テスト詳細メソッドを編集した後のテストスクリプトは以下の通りである．

```
using System.Collections.Generic;  using System.Text;
using System;  using System.Collections;  using System.IO;
public class TestScript{
    private static StringBuilder result;
    static void Main(){
        result = new StringBuilder();
        result.Append("Pay¥r¥n");
        PayTest();
        result.Append("ProcessEnd¥r¥n");
        StreamWriter sw =
 new StreamWriter(@"C:¥Users¥Desktop¥TestResult.txt",
            false, Encoding.GetEncoding("shift_jis"));
        sw.Write(result.ToString());
        sw.Close();
    }
    public class PayingCard
    {
        public string name;
        public int buffer;
        public PayingCard(string name, int buffer)
        {
            this.name = name;
            this.buffer = buffer;
        }
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("[");
            sb.Append(ConvertResult(name));
            sb.Append("," + ConvertResult(buffer));
            sb.Append("]");
            return sb.ToString();
        }
    }
    public static void PayTest()
    {
        string c_0_name = "HRX";
        int c_0_buffer = 100071;
        PayingCard c_0 =
            new PayingCard(c_0_name, c_0_buffer);
        int price_0 = 88;
        PayTestDetail(c_0, price_0);
    }
    private static void PayTestDetail(PayingCard c, int price)
    {
        TestLibrary.Card card =
            new TestLibrary.Card(c.name, c.buffer);
        int payout = TestLibrary.TestModule.Pay(card, price);
        PayAddResult(payout);
    }
    private static void PayAddResult(int payout)
    {
        result.Append("payout¥r¥n");
        result.Append(ConvertResult(payout) + "¥r¥n");
        result.Append("TestResultEnd¥r¥n");
    }
}
```

まず Main メソッドではテスト結果を格納する変数 result を生成している．result.Append はテスト結果についての情報を result に追加する文である．

次に Main メソッドではテストメソッドの PayTest メソッドが呼ばれる．このメソッドは生成されたテストケースを表1の対応に従った型の変数に格納し，それをテスト詳細メソッドである PayTestDetail メソッドに渡す．PayTestDetail メソッドは本ツール上で筆者が編集した．このメソッドでは，受け取った PayingCard 型の入力 c が，テスト対象メソッドである Pay メソッドの Card 型の入力 card に直接入力できないために，新たに Card 型の変数を用意して Pay メソッド

に渡すという処理と，Pay メソッドの出力を受け取りテスト結果書き出しメソッドである PayAddResult メソッドに入力するという処理が記述されている．

最後に Main メソッドでは result に貯まったテスト結果を外部ファイルに書き出してプログラムの処理は終了する．

テストスクリプトの実行は成功し，その後にテスト結果を評価した．その様子が以下の図である．

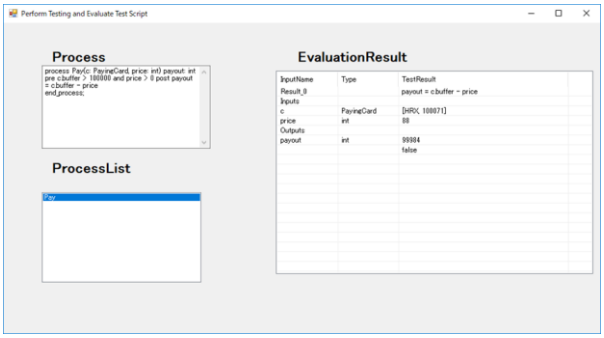


図 9：テスト結果評価後

入力 c の要素 buffer のテストデータは 100071 で入力 price のテストデータは 88 なので，出力 payout が満たすべき条件は payout = 99983 である．しかし，テストした Pay メソッドが出力した値は 99984 なので評価結果は false と出ている．これは期待していた通りの結果である．

以上のように SOFL 形式仕様に基づくプログラムに対して，テストケース生成，テストスクリプト生成と実行，テスト結果評価というテストの一連の流れをすべて本ツール上で行うことができた．

9. まとめと今後の課題

本研究では先行研究[2]のテストケース生成とテスト結果評価のプログラムに加えて，テストスクリプト生成と実行，テスト結果の入力を同じツール上で行えるようにした．これによって SOFL 形式仕様に基づくプログラムのテストをスムーズに行うことができ，テスト時間を削減することが期待できる．

今後の課題の一つ目は対応できるテスト対象プログラムの言語を増やすことである．現在は C#のプログラムのテストのみ行うことができるが，SOFL 形式仕様からは Java や C++言語のようなほかのプログラミング言語で実装が行われる可能性があるため，対応言語を増やす必要がある．この場合言語によってツールの変更すべき点はテストスクリプトの生成方法である．各言語ごとにテストスクリプトを生成するプログラムをパッケージ化し，それをスクリプト生成時に選択するような形を取れば拡張性が高まる．

二つ目はツールの自動化範囲を広げることである．現在はテストスクリプトの一部をユーザーが手動で記述する必要がある．それは主に，テストケースをテスト対象プログラムに変換する部分と，テスト結果を SOFL 型と一意に対応した型に変換させる部分である．これらを自動化するためにはテスト対象プログラムの入力と出力の型を調べて柔軟に対応できるようにしなければならない．

10. 謝辞

本研究は JSPS 科研費 26240008 の助成を受けたものです．

参考文献

[1] 独立行政法人情報処理促進機構 技術本部 ソフトウェア高信頼化センター,ソフトウェア開発データ白書 2016-2017, 2016

[2] 池田逸人，劉少英，形式仕様に基づくテストケースの自動生成とテスト結果の自動評価，第 79 回全国大会講演論文集，情報処理学会，2017

[3] Shaoying Liu , Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, 2004

[4] 劉少英，実用性が高い形式工学手法と支援ツールの研究開発，https://www.ipa.go.jp/files/000027372.pdf，2012

ソースコードからCDFDへの変換によるSOFL仕様記述の支援ツールの提案

新城 汐里
法政大学情報科学研究科
shiori.araki.9n@stu.hosei.ac.jp

劉 少英
法政大学情報科学研究科
sliu@hosei.ac.jp

要旨

ソフトウェアを開発する際に機能を定義する仕様の記述が疎かになる場合がある．これはレガシーソフトウェアにも該当することであり，仕様記述が疎かであるどころか存在しない場合もある．この問題を解決するために*Java*ソースコードから*CDFD*を生成し，*SOFL*形式仕様の記述を支援するツールを提案する．これにより仕様記述にかかるコストを軽減し，ソースコード中のバグの検出を助ける．

1. はじめに

ソフトウェア開発において機能を定義する仕様の記述が疎かになる場合がある．これは現在開発されているソフトウェアに限らずレガシーソフトウェアにも当てはまる．加えて仕様の紛失や，仕様が初めから記述されていないというような場合もある．また，ソースコード中のバグを検出するためにレビューという手段が存在するが，レビューは行う人物に依存するという問題がある．

これらの問題を解決するためにリバースエンジニアリングという手段があり，研究がされてきた．例えばBenedusiらはPascalで書かれたソースコードからDFD(データフロー図)を生成するための方法とDFDがソフトウェアの理解と保守に対して持つ有用性についての考察について述べており[1]，A. B. O'HareらはRE-Analyzerというツールの開発によるC言語で実装されたソフトウェアに対するリバースエンジニアリングの支援を行っている[2]．また井垣らはDFDを用いてレガシーソフトウェアの依存解析を行い，サービス指向アーキテクチャのサービス候補を抽出する手法を提案している[3]．

本研究ではJavaソースコードからCDFD(条件データフロー図)およびSOFL形式仕様を半自動的に生成するための変換規則を述べ，その規則を用いたSOFL形式仕様を記述するための支援ツールの研究開発を行う．SOFL(Structured Object-Oriented Formal Language)[4, 5]は構造化手法とオブジェクト指向言語を融合した仕様記述言語を提供する形式工学手法であり，Javaから形式仕様への変換も容易であるため本研究にとって有用である．ツールの目的は過去に作成されたソースコードを構文解析することでCDFDの自動的な生成やSOFL形式仕様の半自動的な生成を行い，ソースコードとCDFDの両方を見ながら生成されなかった部分の仕様をユーザが手動で記述することによって仕様の不備を発見できるよう支援することである．

以降，2節ではSOFLの概要およびその内容を簡潔に述べる．3節ではJavaソースコードからCDFDへの変換規則を定義する．4節ではJavaソースコードからSOFL形式仕様へ変換する際の方針を述べる．5節では定義した変換規則を基に開発した支援ツールについて述べる．6節では開発した支援ツールを用いて簡単なソースコードによる事例研究を行う．7節では事例研究で得られた結果と今後の課題をまとめる．

2. SOFLの概要

本節ではSOFLの概要について述べる．SOFLはDFD[6]，ペトリネット[7]，VDM-SL[8, 9]を統合した形式工学手法[10]である．要件と仕様の設計に対して形式仕様記述言語を提供することによりソフトウェアシステムを開発するための実用的な方法を提供している．この手法を用いた仕様はJavaやC++のようなオブジェクト指向言語への変換に適している．以降，単純な自動販売機のシステムを事例としてCDFDとSOFLのモジュールの構造について述べる．

2.1. CDFD の概要および事例

CDFD は 2 節で述べた 3 つの手法を統合し形式化された DFD を指す．SOFL を構成するものの 1 つであり，SOFL のモジュールと関連付けられた図である．システムの構造を描画したものでもあり，あらかじめ定義された機能要件と機能間の依存関係を表現する．また，CDFD は階層構造である．1 つのプロセスを分解することで低レベルのより詳細な複数のプロセスを表現できる．数字のない長方形はプロセスを意味し，入出力を持った 1 つの操作を表す．名前の付いた矢印はデータフローを意味する．データフローにつけられた名前はそのデータの性質を，矢印はデータフローの方向を表す．数字の ID がつけられた長方形はデータストアを意味し，ファイルやデータベースのようなデータを与える役割を持つ．

DFD との主な相違点は 3 点である．1 点目は破線で表した制御データフローを用いている点である．このデータフローはプロセスにデータを渡すことなく実行の指示を出す場合に用いる．2 点目はプロセスの表現が詳細である点である．プロセスはプロセス名 (中央部)，入力ポート (左部)，出力ポート (右部)，事前条件 (上部)，事後条件 (下部) で構成される．また，入力ポートと出力ポートは複数存在する場合がある．これは入力あるいは出力が複数通り存在することを表す．3 点目は入力あるいは出力を持たずともプロセスとして表現できる点である．そのため入力はないが出力のあるプロセスや，入力はあるが出力のないプロセスを表現することもできる．

事例として図 1 に電子マネーで商品を購入できる自動販売機の CDFD を示す．ただし，この CDFD はあくまで図の流れを見るために簡易的に記述されたのものであり現実に使用されている自動販売機とは細部が異なる．この図 1 における動きは次の通りである．まず自動販売機で購入する商品を選択したことを表すデータフロー *button* はプロセス *Pressed* で処理を行う．このプロセスはデータストア *stock_of_drink* に在庫があるかどうかを確認する．在庫がないなら在庫がないメッセージを表すデータフロー *out_of_stock_message* を生成し，在庫があるなら在庫が存在することを伝えるデータフロー *confirmation* を生成してプロセス *Selling* に渡す．そしてデータフロー *confirmation* を受け取ったプロセス *Selling* は投入金額を表すデータフロー *money* も受け取る．このプロセスではデータストア *stock_of_drink* の在庫から商品を取り出す．投入金額がデータストア *stock_of_drink* に記録された商品の値段未満であればエラーメッセージを表すデータフロー *error_message* と返金を表すデータフロー *repayment* を生成し，投入金額が商品の値段以上であればデータストア *stock_of_drink* に該当する在庫を減らすような変更を加えてから商品を表すデータフロー *drink* とおつりを表すデータフロー *change* を生成する．

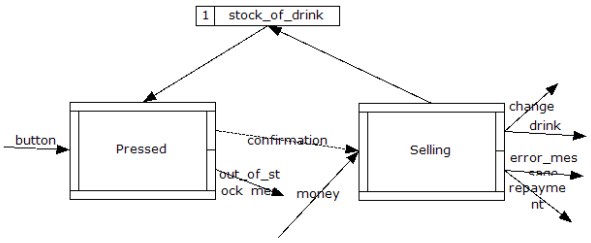


図 1. 自動販売機の CDFD

2.2. モジュールの構造

SOFL のモジュールは CDFD で表現したプロセス，データフロー，データストアを正確に定義するものである．事例として図 1 をモジュールとして定義したものを表 1 に示す．これらも例示のために自然言語を用いて簡単に記述したものである．表 1 において，モジュール名は「自動販売機」とする．このモジュール内で宣言する型は商品を表す Drink と選択した商品を表す Button であり，ストア変数は商品の在庫と値段を表す *stock_of_drink* である．このモジュールの振る舞いを表す CDFD は behav に番号が記述される．プロセス *Init* は変数の初期化を行うプロセスである．プロセス *Pressed* および *Selling* は 2.1 節の説明と同様である．

また，プロセス *Selling* を定義したものを表 2 に示す．表 2 において，プロセス名は *Selling* である．このプロセスに対する入力は *money* と *confirmation* であり，このプロセスからの出力は *drink* と *change* もしくは *error_message* と *repayment* のどちらかである．操作を実行するにあたってデータストア *stock_of_drink* に対し読み込みおよび書き込みを行う．事前条件として *money* が 0 より大きく，*confirmation* を受け取っていることが定義されている．事後条件として，2.1 節で説明したようにプロセスが実行すべき操作が定義されている．

表 1. SOFL モジュールの構造

module 自動販売機;
type Drink=Drink の型の定義; Button=Button の型の定義;
var stock_of_drink : stock_of_drink の型;
behav CDFD_1;
process Init;
process Pressed;
process Selling;
end_module

表 2. プロセス仕様の構造

process Selling (money: int, confirmation: sign) drink: Drink, change: int | error_message: string, repayment: int
ext wr stock_of_drink
pre money>0 and confirmation を受け取っている
post if money が商品の値段未満である then error_message の生成 and repayment=money else stock_of_drink から商品の取り出し and drink を生成 and change=money− 商品の値段
end_process

3. CDFD への変換規則

本節ではソースコード中の構造から CDFD への変換規則を定義し，その原理を述べる．ただし，定義する変換規則は構文解析を用いて機械的に変換することを前提とする．また，規則を用いた変換の事例を紹介する．

3.1. 順序構造の変換

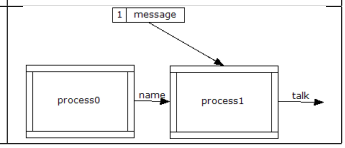
本節では順序構造における CDFD への変換規則の定義について述べる．これは本節で述べる変換規則の基礎となる定義である．順序構造の変換規則を以下に定義し，その理由を述べる．

- 1 つのステートメントを 1 つのプロセスとする．1 つのステートメントがそれ以上分解するのが難しい 1 つのまとまった処理だからである．
- 基本データ型もしくは見た目の挙動が基本データ型と同じように見える型を持つ 1 つのローカル変数を 1 つのデータフローとする．基本データ型の変数をメソッドに渡し，それに変更が加えられても元の変数には影響がなくフローとして表現しやすいためである．
- 参照型である 1 つのローカル変数，もしくは型に依らない 1 つのフィールド変数を 1 つのデータストアとする．参照型の変数をメソッドに渡し，それに変更が加えられた場合に元の変数にも影響がありフローとして表現しづらいためである．
- データフロー名およびデータストア名には変数名を用いる．
- データフローの方向やデータストアへの接続の方向はステートメントもしくは式の中における変数が使用された位置から決定する．例えば変数が宣言された場合はその変数を出力として扱い，式の中で計算に使用されたが値の変更がないような場合は入力として扱う．
- 同じデータストアへ接続するプロセス同士は実行順に制御データフローの始点と終点となる．データストアへ接続する順番を明確にする必要があるからである．

以上の定義を用いて Java ソースコードから CDFD に変換した場合の事例を表 3 に示す．

表 3. 順序構造の変換例

ソースコード	CDFD への変換例

`public static String message;`
`public static void function(){`
`  String name="I";`
`  String talk=name+" say "+message;`
`}`
	

3.2. 選択構造の変換

本節では選択構造における CDFD への変換規則の定義について述べる．選択構造のステートメントは条件と処理の 2 種類から構成される．ここで述べる選択構造は if 文についてであり，switch 文は本稿では言及しない．

CDFD への変換が変則的かつ正しくは選択構造に分類されないからである．以上を踏まえて選択構造の変換規則を以下に定義し，その理由を述べる．

- 1 つのプロセスはそのプロセスを親プロセスとした複数の子プロセスを持つ場合がある．1 つの選択構造が複数の処理を持つことが可能だからである．この規則を適用することによって CDFD の階層構造を表現することが可能となる．
- 条件は子プロセスとしてではなく，親プロセスの一部と見なして変換する．条件はステートメントではなく式として扱われるからである．
- 子プロセスに相当するステートメントは該当する変換規則に沿って変換を行う．
- 選択された処理を表すプロセスに制御データフローを接続する．選択しなかった処理を実行しないようにするためである．
- 選択に関係なくプロセスにつながる可能性のある全てのデータフローを表現する．静的なコード解析のみでは選択の特定がほぼ不可能なためである．
- 子プロセスを持つ親プロセスは子プロセスへ接続データフローも親プロセスに接続しているものとして表現する．子プロセスは親プロセスを構成する一部だからである．

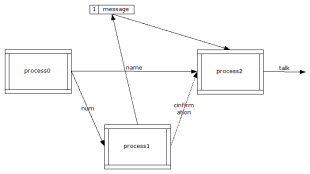
以上の定義を用いて if 文を用いた Java ソースコードから CDFD に変換した場合の事例を表 4 に示す．この時，連続した変数宣言は結合して 1 つのプロセスとして表現し，制御データフローは変数として表されていないため適当な名前を用いている．また，if 文の処理をプロセスに変換する際に記述した制御データフロー *then*，*else* および if 文全体をプロセスに変換する場合に記述した制御データフロー *confirmation* は親プロセスである if 文全体が生成したものとして見なす．そのため条件を表すプロセスは処理を表すプロセスより上の階層に存在するとして考える．なお，本稿における選択構造の表現は本来の SOFL における表現とは異なる．本来の SOFL では選択構造はひし形で分岐を表現している．

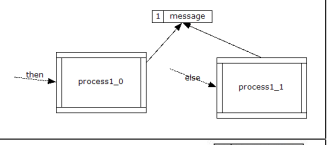
3.3. 反復構造の変換

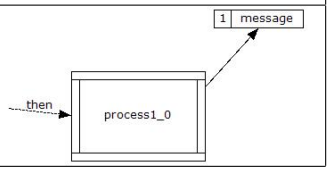
本節では反復構造における CDFD への変換規則の定義について述べる．ここで述べる選択構造は for 文，while 文，do-whle 文についてである．これらのステートメントは for 文なら初期化，条件，処理，更新の 4 種類から構成され，while 文と do-whle 文なら条件と処理の 2 種類から構成される．以上を踏まえて反復構造の変換規則を以下に定義し，その理由を述べる．なお，この規則は選択構造と一部重複する．

表 4. 分岐構造の変換例

ソースコード	CDFD への変換例

`public static String message;`
`public static void function(){`
`  String name="I";`
`  double num=Math.random()*10;`
`  if(num>5){`
`  message="Bye.";`
`  }else{`
`  message="Hello.";`
`  }`
`  String talk=name+" say "+message;`
`}`
	

`if(num>5){`
`  message="Bye.";`
`}else{`
`  message="Hello.";`
`}`
	

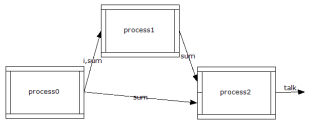
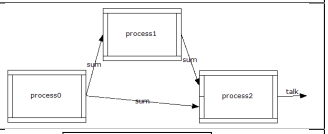
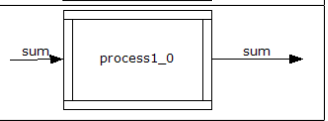
`if(num>5){`
`  message="Bye.";`
`}`
	

- 1 つのプロセスはそのプロセスを親プロセスとした複数の子プロセスを持つ場合がある．理由は選択構造の場合と同様である．
- 初期化，条件，更新は子プロセスとしてではなく，親プロセスの一部として変換する．理由は選択構造の場合と同様である．
- 子プロセスに相当するステートメントは該当する変換規則に沿って変換を行う．
- 選択に関係なくプロセスにつながる可能性のある全てのデータフローを表現する．反復構造の処理が 1 度も実行されない可能性も存在するが，静的なコード解析のみでは特定がほぼ不可能なためである．

- 子プロセスを持つ親プロセスは子プロセスへ接続データフローも親プロセスに接続しているものとして表現する．理由は選択構造の場合と同様である．

以上の定義を用いて反復構造を用いた Java ソースコードから CDFD に変換した場合の事例を表 5 に示す．こちらも連続した変数宣言は結合して 1 つのプロセスとして表現している．

表 5. 反復構造の変換例

ソースコード	CDFD への変換例
`public static void function() { int sum = 0; int i=0; while(i<=5){ sum ++; i++; } String talk="I have "+sum+" pens"; }`	
`public static void function() { int sum = 0; int i=0; do{ sum ++; i++; }while(i<=5); String talk="I have "+sum+" pens"; }`	
`public static void function() { int sum = 0; for (int i = 1; i <= 5; i++) { sum += i; } String talk="I have "+sum+" pens"; }`	
`do{ sum ++; i++; }while(i<=5);` / `while(i<=5){ sum ++; i++; }`	
`for (int i = 1; i <= 5; i++) { sum += i; }`	

4. 形式仕様への変換

本節ではソースコードから SOFL のプロセス仕様への変換における所見と方針について述べる．3 節で述べた CDFD への変換は自動的に行うことが可能だが，プロセス仕様への変換は全てを自動化することは難しい．機械的に作成した仕様は自然言語のような曖昧性は存在しないが，システムの概念や目的は機械的に判断できないため変換された仕様を開発目的に合致させるのは難しいからである．例えばプロセス仕様内に事前条件を記述する場合，あるプロセスに分岐するための条件を事前条件とするか他の箇所で決めた制限を事前条件とするかを機械に判断させるのは困難である．

以上よりプロセス仕様におけるプロセス名，入力，出力，データストアのような機械的に判断しやすい部分はツールが自動的に生成し，事前条件と事後条件のような人間の思考が必要な部分はユーザ自身が記述するものとする．これによりソースコードの内容を理解しながら記述を行うため，ユーザがバグを発見しやすくなる，あるいはエラーが発生しなくともバグとなるような使用する変数が過不足である場合に気づきやすくなるという利点がある．なお自動生成には 3 節の変換規則の利用が可能だが，データフローやデータストアに相当する変数の型を明確にする必要がある．

5. 支援ツールの紹介

定義した変換規則を用いて CDFD を作成する支援ツールを開発した．本ツールの開発環境は Eclipse Neon 3(4.6.3)[11] を用い，Eclipse プラグイン [12] としてエディタに付随させたビューを拡張する形で実装した．そのためこのツールを利用するためには開発したプラグインを Eclipse にインストールし，Java ファイルを開発したエディタで開けばよい．CDFD の表示には Zest Visualization Toolkit[13] というプラグインを使用した．プログラミング言語は Java を用いた．ソースコードを解析するために Eclipse JDT[14] が提供している Java のソースコードを抽象構文木として保持する AST を，解析結果を記録するためにデータベースとして Java DB[15] を用いた．

ツールの機能は 3 つに分けられる．図 2 に支援ツールの画面を示しつつ説明する．1 つ目は Java エディタ (赤枠) である．これは Eclipse に初めから備わっているものを使用している．2 つ目は CDFD ビュー (黄枠) である．Java ファイルを支援ツールで開くと同時にソースコードの読み込みと変換を行い，画面上に表示する機能である．このツールではプロセスはグレーの長方形で表現し，データフローはグレーの矢印で表現する．データストアはプロセスとの区別のために青色の長方形で表現し，それに対する接続は緑の矢印で表現する．データフロー名は重複を許すこととする．始点もしくは終点が同じ階層に存在しない場合はダミーとして作成したプロセスであるプロセス *No_To* もしくはプロセス *No_From* にデータフローを接続する．プロセス *No_To* はデータフローが存在するがフローの終点が不明な場合の終点として用い，プロセス *No_From* はデータフローが存在す

るがフローの始点が不明な場合の始点として用いる．ただし，終点のプロセスがどの階層にも存在しないデータフローは表現しないものとする．例えばソースコード内で変数を宣言したがその変数が後に使われなかった場合を考える．この時，変数を用いたフローの始点となるプロセスは存在するが，終点となるプロセスは存在しない．用いたプラグインの関係でフローは始点と終点の双方が存在しないと表示できないため，終点のプロセスがどの階層にも存在しないデータフローは表現しないこととする．プロセスの名前はステートメントの記述を用いるものとする．例えば $int\ a = 0;$ というステートメントが存在する場合，「$int\ a = 0;$」という名前のプロセスが存在することになる．ただし，メソッドの仮引数を表現する場合に限り「$main(String[]\ args)$」のように「メソッド名 (仮引数)」をプロセス名として扱う．ソースコード内の記述をプロセスの名前とすることにより，表示されたプロセスがソースコード上のどの部分であるかを直感的に理解しやすくするためである．また，可視性を向上させるために連続した変数宣言のプロセスは結合して 1 つのプロセスとし，if 文のプロセスの子プロセスには $then$ もしくは $else$ をプロセス名の頭に追加する．3 つ目はモジュールの構造ビュー (青枠) である．こちらも Java ファイルを開くと同時にソースコードからモジュールの構造を木構造として変換し，画面上に表示する．この機能は CDFD ビューに表示する階層の指定にも用いる．また，仕様記述エディタとしての機能も備える予定である．この 3 つの機能を用いることにより，CDFD を見ることでシステムの動きの把握を助け，ソースコードを確認することで詳細を理解しつつ，同時に仕様を記述を行うことが実現できる．

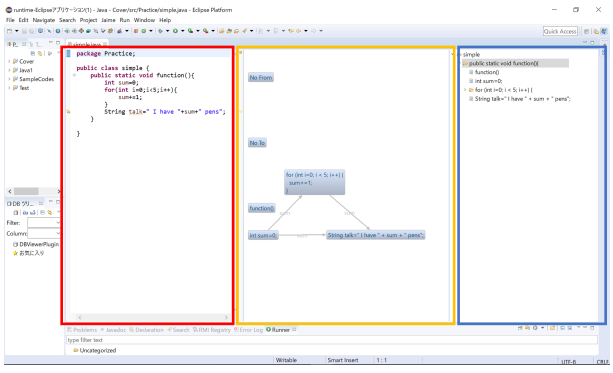


図 2. 支援ツールの画面

5.1. データベースの設計

変換に用いるためのデータを記録する必要がある．そこでリレーショナルデータベースを用いてデータの記録を行うためにデータベースの設計を行った．設計したデータベースの ER 図を図 3 に示す．

プロセステーブルはプロセスを記録するために用いる．バインディングのキーは AST がメソッド名やクラス名ごとに割り振ったキーであり，階層の異なる同名のメソッドやクラスを区別するために用いる．ただし，プロセスがメソッドやクラスではなくステートメントの場合はバインディングのキーは NULL である．また，プロセス名の文字数が型の制限である長さ 255 を超えた場合には一部を切り捨てて記録する．データフローテーブルはデータフローを記録するために用いる．これは同名の変数であっても始点となるプロセスと終点となるプロセスが異なれば別のデータフローとして扱うためである．そのため同じバインディングのキーを持っていても異なるデータフロー ID を持つ場合がある．バインディングのキーは AST が変数名ごとに割り振ったキーであり，階層の異なる同名の変数を区別するために用いる．始点プロセス ID はデータフローの始点がどのプロセスであるかを表す．データストアテーブルはデータストアを記録するために用いる．バインディングのキーは AST はデータフローテーブルと同様である．また，親プロセス ID はデータストアがどのプロセスの下で宣言されているかを表す．

階層関係テーブルはプロセス間の階層関係を記録するために用いる．親プロセス ID は子プロセスの 1 つ上の階層にあたるプロセス ID を表し，子プロセス ID は 1 つ上の階層にあたるプロセスのプロセス ID を表す．親プロセス ID は 1 つ以上の子プロセス ID を持つが，子プロセス ID は 1 つの親プロセス ID を持つ．データストア接続テーブルはプロセスからデータストアへの接続関係を記録するために用いる．データストア ID はプロセスに使用されるデータストアを表し，プロセス ID はデータストアを使用するプロセスを表す．書き込みは値が true であるならデータストアに対する書き込みが発生することを表し，値が false であるならデータストアの値を参照するのみであることを表す．データフロー接続テーブルはデータフローの終点を記録するために用いる．終点プロセス ID はデータフロー ID を持つデータフローの終点がどのプロセスであるかを表す．

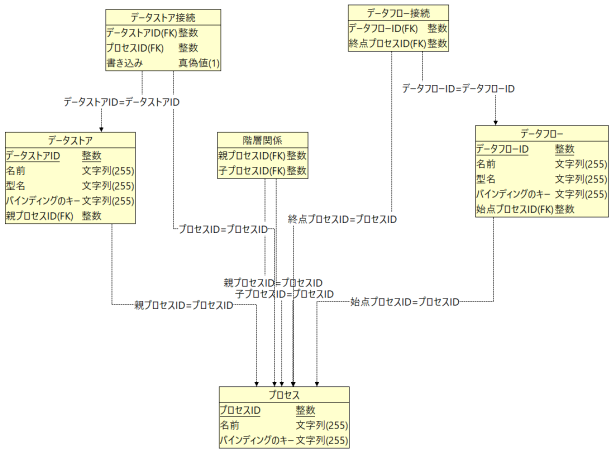


図 3. データベースの ER 図

5.2. クラスの作成

ツールの実装のためにいくつかのクラスを作成した．これらのクラスは主に 4 種類に分類される．すなわち Eclipse の画面に表示を行うクラス (AutoEditor クラス)，データベースに対して操作を行うクラス (DataManagement クラス，UseDB クラス)，SQL 文を生成してデータベースに対して操作を行うクラスに渡すクラス (Ext_DB クラス，Flow_DB クラス，Link_DB クラス，Parent_DB クラス，Process_DB クラス，Store_DB クラス)，ソースコードに対して構文解析を行うクラス (DecomExpression クラス，MyVisitorwithDecom クラス) である．作成したクラスの UML クラス図を図 4 に記す．ただしメソッド等は省略している．

AutoEditor クラスはプラグインを動かす際に最初に動くクラスである．このクラスでは DataManagement クラスのインスタンスを作成して必要なテーブルを作成する，対象となるソースコードを AST で抽象構文木とし，DataManagement クラスのインスタンスと併せて MyVisitorwithDecom クラスに渡す．MyVisitorwithDecom クラスでの解析が終了したら DataManagement クラスに CDFD を生成するように指示を出す．DataManagement クラスと UseDB クラスはデータベースのテーブルに対する操作を行うクラスである．UseDB クラスは MyVisitorwithDecom クラスと DecomExpression クラスから得た情報をテーブルに記録し，DataManagement クラスは CDFD を生成するためにテーブルから情報を得る目的で用いる．また，DataManagement クラスは UseDB クラスを継承している．DecomExpression クラスと MyVisitorwithDecom クラスは抽象構文木を探索するクラスである．CDFD を生成するために必要な情報を DataManagement クラスを用いて記録する．MyVisitorwithDecom クラスはステートメントを探索し，DecomExpression クラスはステートメント内にある式を探索するために MyVisitorwithDecom クラスから呼び出される．Ext_DB クラス，Flow_DB クラス，Link_DB クラス，Parent_DB クラス，Process_DB クラス，Store_DB クラスは DataManagement クラスと UseDB クラスから呼び出され，各テーブルを操作するための SQL 文を文字列として生成して返すクラスである．

6. 事例研究

本節では実際に Java ソースコードから支援ツールを用いて CDFD への変換を行う．テストに使用した Java ソースコードをプログラム 1 に示す．これは *main* メソッド内で台形則を用いた数値積分を計算し，コンソールに表示するプログラムである．フィールド変数 A, B は区間を表し，フィールド変数 N は区間の分割数を表す．ローカル変数 sum, h, t, w はそれぞれ積分の結果，幅，積分点，重みを表す．また，ローカル変数 i は for 文の条件に用いる．

プログラム 1. Java ソースコード

```
1  package Practice;
2
3  public class Trap {
4    public static final double A=0.,B=3.;
5    public static final int N=100;
6    public static void main(String[] args) {
7
8      double sum,h,t,w;
9      int i;
10
11     h=(B-A)/(N-1);
12     sum=0.;
13
14     for(i=1;i<=N;i=i+1){
15       t=A+(i-1)*h;
16       if(i==1||i==N)w=h/2.;else w=h;
17       sum=sum+w*t*t;
18     }
19     System.out.println(sum);
20   }
21 }
```

プログラム 1 の内，main メソッドを順序構造の変換規則で変換した CDFD を図 5 に，14 行目から 18 行目の繰り返し処理を反復構造の変換規則で変換した CDFD

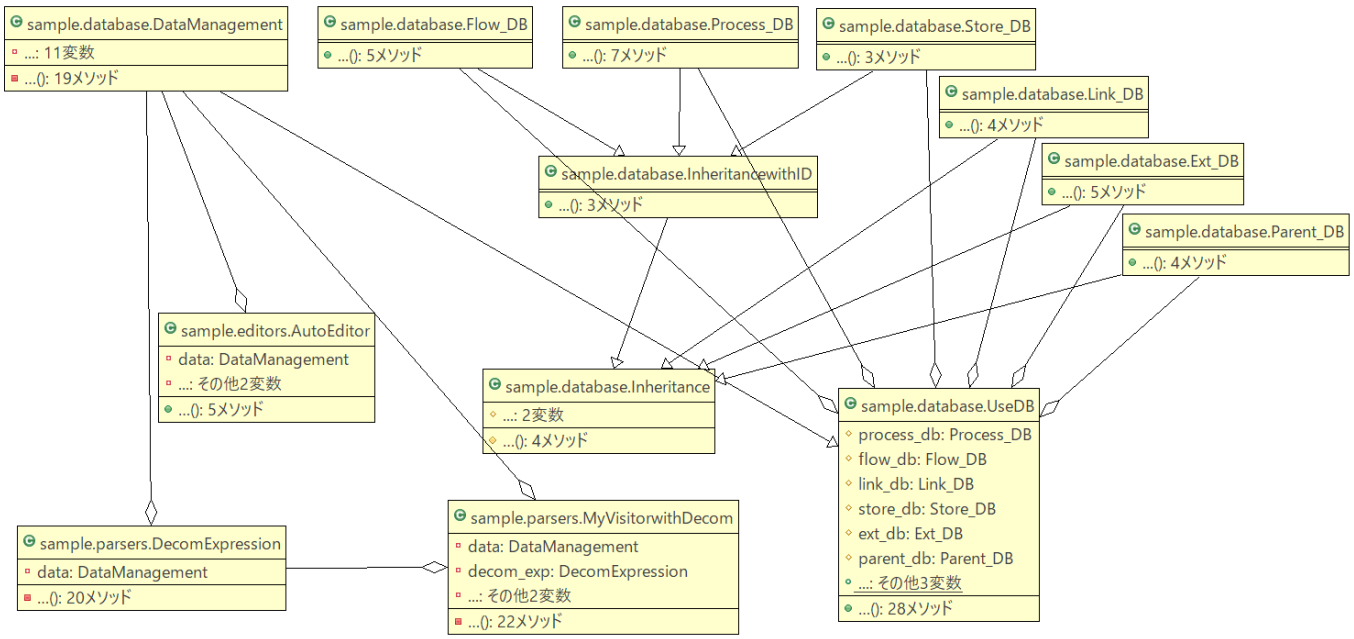


図 4. UML クラス図

を図 6 に，16 行目の条件分岐を選択構造の変換規則で変換した CDFD を図 7 に示す．これらの図からソースコードを分割して階層ごとに見ることが可能であり，プロセス間の依存関係が見て取れる．特に図 5 のプロセス $sum = 0;$ とプロセス $h = (B - A)/(N - 1);$，図 6 のプロセス $t = A + (i - 1) * h;$ と選択構造のプロセス，図 7 のプロセス $then\ w = h/2;$ とプロセス $else\ w = h;$ は互いに関係を持たず，依存関係がないと分かるからである．しかしツールとしての問題点も見受けられる．以下に問題点を記す．

1. データフロー名が見づらい．これは全ての図で見受けられる．これではデータが関係するか否かが明確にならない．解決策としてデータフロー名を設定する箇所でプロセスの始点と終点が同じデータフローがすでに存在する場合はデータフローを追加するのではなくデータフロー名を「a, b, c」のように更新するという方法が挙げられる．

2. 全ての選択を考慮して余分なデータフローを表現している．これは図 5 で見受けられる．今回の例では反復構造が必ず処理を行うためプロセス $sum = 0;$ で生成されたデータフローは反復構造のプロセス以外につながることはないはずである．解決策として反復構造および選択構造の条件を解析を行う際に真偽を計算し，最低 1 回は満たすかによってデータフローの終点を判断するということである．判断の結果をデータベースに記録すれば CDFD を表示する際に余計なデータフローを減らすことができる．ただし 4 節で述べたように静的なコード解析のみでは分岐先の特定がほぼ不可能なため推奨しない．

3. プロセスに CDFD のプロセスとしての要素が欠けている．これは全ての図で見受けられる．特に入力ポートと出力ポートが表現されていないという問題がある．プロセス $System.out.println(sum);$ のように同じ変数を表現するデータフローが複数入力もしくは出力された場合はポートを分割する必要がある．解決策としてまずプロセスを表す図を SOFL におけるプロセスを表すものに差し替える．その上で同じ変数を表すデータフローが複数入力もしくは出力された時にデータベースに記録し，プロセスを表す図に反映させるということが挙げられる．これを CDFD を生成する箇所に追加すれば表現可能である．

4. プロセス $System.out.println(sum);$ においてデータストア out へ書き込みがされている．これは図 5

で見受けられる．変数 *out* がデータストアとなっているため変換規則には反していないが，このプロセスはコンソールへの出力を行うプロセスであり表現としては不自然である．解決策として解析時にデータストアとなりうる変数の型を確認し，PrintStream 型であるならコンソールを表すプロセスを終点として設定する方法が挙げられる．

5. 選択構造や反復構造における条件が表現されず，理解の助けになりにくくなっている．これは図 6 と図 7 で見受けられる．解決策として条件を上のプロセスに含むのではなく条件もプロセスとして扱う，もしくは 4 節で触れた本来の CDFD における条件構造を用いることが挙げられる．条件をプロセスとして扱う場合は一度条件のプロセスをデータフローの終点とした上でそれぞれの分岐先に改めて新しくデータフローを生成するという方法がとれる．CDFD の条件構造を用いる場合は条件構造用のテーブルを新たに用意する必要がある．

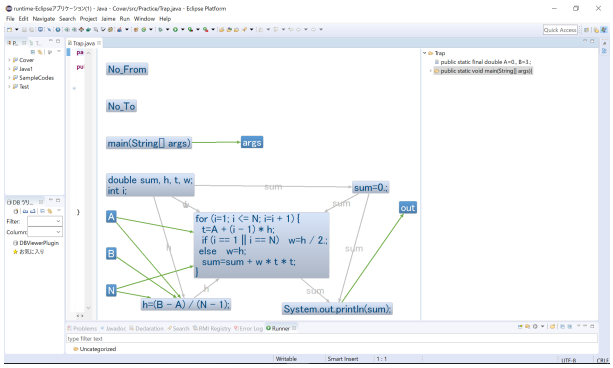


図 5. main メソッド (順序構造) の CDFD

7. 結論

本稿では Java ソースコードから CDFD を生成することで SOFL 形式仕様の記述を支援を行うための変換規則とツールの提案を行い，事例研究によってその有効性を確かめた．その結果，ソースコード中のプロセスにおいて関係性を視覚的に確認することができた．今後の課題は変換規則の追加とツールの完成の 2 つが挙げられる．1 つ目の課題である変換規則の追加について述べる．まず CDFD で表現できる範囲を広げる必要がある．今回の規

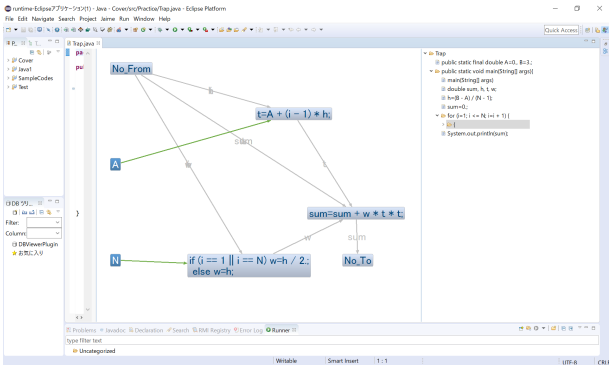


図 6. for 文 (反復構造) の CDFD

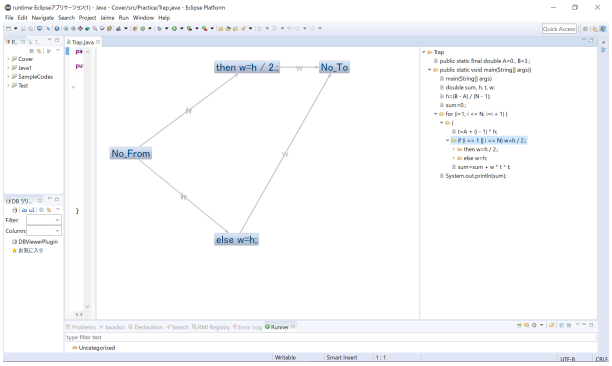


図 7. if 文 (選択構造) の CDFD

則では 1 つのメソッドに対する変換規則のみを定義している．そのためクラス，他メソッドとの関係，継承，カプセル化，コンストラクタといった 1 つのメソッドでは表現しきれない場合に対する変換を定義していない．これは選択構造に対する変換規則へ追加をすべきである．また，private や final といったメソッドや変数につける修飾子を考慮していない．これらはデータフローやデータストアに対する矢印の向きに直接関係する．例えば final は変更不可能な変数として扱うため，データストアとして扱う場合は変数の宣言時以外では書き込みが常に false でなくてはならないという制限を設ける必要がある．これはデータベースのテーブルに該当する列を設ければよい．そしてソースコード中のコメントに対する変換を無視している．これは SOFL 形式仕様の記述の際に自動で変換することの可能な箇所であるためコメントを抽出することで自動的な変換を実現する必要がある．このように，変換規則が不足していることが課題である．2 つ目の課題であるツールの完成について述べる．まず 6 節

で問題点として挙げた入力ポートと出力ポートに加え，データフローを表現する箇所に制御データフローを表現する記述を追加することで制御データフローを実装する必要がある．そして先に述べた変換規則の追加を随時実装していくこととなる．また，テストで使用したソースコードはコンパイルエラーが起きないものであり，コンパイルエラーが発生するようなソースコードに対するテストを行えていない．そのため本稿ではバグの検出に対する明確な有用性を示すことができていない．さらに紹介した支援ツールでは CDFD の表示のみで SOFL 仕様記述を行うための機能が実装されていない．そのため 5 節でも触れたように画面上のモジュールの構造ビューの下部に SOFL 仕様記述を行うためのエディタを設置する必要がある．エディタを設置すれば事前条件と事後条件を適用した場合の事例研究を行うことも可能となる．このように，画面上に SOFL 仕様記述のためのエディタを実装時に正しく CDFD および SOFL 形式仕様が表示されるように改善していくことが課題である．今後は以上の課題を解決した上で複数のメソッドやクラスに関わる事例研究および参考文献として示した先行研究との具体的な比較を行う必要がある．

8. 謝辞

本研究は JSPS 科研費 26240008 の助成を受けたものです．

参考文献

[1] Paolo Benedusi, Aniello Cimitile, and U De Carlini. A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance, 11 1989.

[2] A.B.O ’ Hare, E.W.Troan. Re-analyzer: From source code to structured analysis. *IBM Systems Journals*, Vol. 33, No. 1, pp. 110–130, 1994.

[3] 井垣宏, 木村隆洋, 中村匡秀, 松本健一. Dfd を利用したプログラム処理間の依存解析によるレガシーソフトウェアからのサービス抽出. ソフトウェアエンジニアリング最前線 ２００９ 情報処理学会ＳＥ, pp. 89–96. 近代科学社, September 2009.

[4] Shaoying Liu. *Formal Engineering for Industrial Software Development*. SpringerVerlag, 2004.

[5] 劉少英. 実用性が高い形式工学手法と支援ツールの研究開発, 2012. https://www.ipa.go.jp/sec/softwareengineering/reports/20130422.html.

[6] 葛山善基, 神代知範. データフロー図の構造化記述法について. 情報処理学会研究報告ソフトウェア工学（SE）, pp. 9–16, sep 1996.

[7] W. Reisig. *Petri Nets: An Introduction*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2012.

[8] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[9] 小田朋宏, 荒木啓二郎. Vdm-sl 仕様からの smalltalk プログラムの自動生成. ソフトウェア・シンポジウム 2016 in 米子 論文集, pp. 1–10, Jun 2016.

[10] 漢明張, 昌満野呂, 篤史沢田, 敦吉田, 吉成蜂巣, 励士横森. アーキテクチャ指向開発における形式手法の適用に関する考察 (ディペンダブルコンピューティング組込み技術とネットワークに関するワークショップ etnet2013). 電子情報通信学会技術研究報告 : 信学技報, Vol. 112, No. 482, pp. 61–66, mar 2013.

[11] Eclipse. https://www.eclipse.org/.

[12] 大城正典, 永井保夫. Eclipse 視覚化プラグインによる総合的なプログラミング教育支援システム. 情報教育シンポジウム 2015 論文集, 第 2015 巻, pp. 23–30, aug 2015.

[13] Zest visualization tool kit. http://www.eclipse.org/gef/zest/.

[14] Eclipse jdt project. https://projects.eclipse.org/projects/eclipse.jdt.

[15] Java db. http://download.oracle.com/javadb/index.html.

データ値の差異とデータフローの視覚化によるデバッグ補助手法の提案

神谷 年洋
島根大学大学院自然科学研究科
kamiya@cis.shimane-u.ac.jp

要旨

クラッシュしないバグのデバッグ，すなわち，プログログラムの実行中に不正な値が生成され，ヌルポインタエラーなどで中断されることなくプログラムの実行が続き，一見正常に終了するものの，不正な出力が得られる不具合を修正する状況を想定したデバッグの補助手法を提案する．

提案手法は動的解析手法の一種であり，対象となるプログラムを実行して実行トレースを収集し，実行トレースから不具合に関連するデータフローと関数呼び出しを含む部分列を抽出する．不具合に関連するデータフローを一覧できるようにすることで，開発者が不具合の原因を理解し，ソースコード上で修正を行うべき場所を検討する作業を補助する．抽出される実行トレースの部分列は，プログラムの実行開始から不具合に至った実行系列を含むものであるが，値の差異に着目した絞り込み，および，データフローによる絞り込みにより，もとの実行トレースよりも小さなものにすることで，一覧性を高める．

本提案手法におけるデータフロー追跡には，動的な解析を用いることにより，参照型（すなわち，ポインタで参照されるオブジェクトなど）のデータ以外にも，値型（整数やブーリアンなどの基本型を含む）のデータに関しても，データフローの追跡が可能になっている．値型のデータフローの追跡は，対象プログラムの実行プロセスを越えてデータが受け渡しされるものについても適用可能である．例えば，データベースやファイルにいったんデータを格納し，そのあと取り出すような場合でもデータフローを追跡することが可能である．

1. はじめに

ソフトウェアの不具合には，クラッシュするバグ (crashing bug) とクラッシュしないバグ (non-crashing bug) がある．クラッシュするバグとは，いわゆるヌルポインタの参照や配列の添字が範囲を超えていることにより，プログラムの実行が中断するものである．クラッシュするバグの場合には，クラッシュした実行時点でのスタックトレースを利用できる．スタックトレースには情報として，プログラムが中断した理由と該当するソースコードの場所（関数名やソースコードのファイル名，行番号など）に加えて，その関数を呼び出した関数，さらにその関数を呼び出した関数，などがエントリポイントまで順に含まれる．そのため，スタックトレースを手がかりとしてデバッグに着手することができる．それに対して，クラッシュしないバグでは，プログラムの実行中に不正な値が生成され，プログラムの実行状態中にその影響が伝播し，出力の一部が不正になる．プログラムが中断しないため，スタックトレースを手がかりとして利用することができない．

本稿では，クラッシュしないバグのデバッグを目的とするデバッグ補助手法を提案する．提案手法では，プログラムを実行して実行トレースを収集し，実行トレースから不具合に関連するデータフローと関数呼び出しを含む部分列を抽出する．実行トレースの部分列の視覚化を通じて不具合に関連するデータフローを一覧できるようにすることで，開発者が不具合の原因を理解しソースコード上で修正を行うべき場所を検討する作業を補助する．

以降，2 で関連する研究について説明する．3 ではトイプログラムを対象として手法の利用方法を説明する．4 で提案手法について説明し，5 では，提案手法を評価するための予備的な実験として，あるオープンソースプ

ロダクトと先のトイプログラムへの適用について，実行時性能も含めた評価を行う．最後に，6 でまとめと展望について述べる．

2. 関連研究

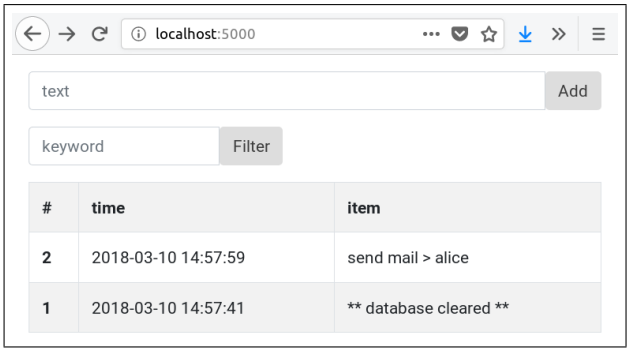
クラッシュしないバグのデバッグを補助するための手法が種々提案されている．

プログラムスライシング [15]，特に，バックワードスライシングとは，不具合（典型的にはプログラムが出力した不正な値）を格納した変数の値を計算するのに利用された文を抽出する手法である．プログラム中の文や式の間に，ある文や式を実行することを決定した文の関係（制御依存）や，ある変数の値を計算するために利用された変数の関係（データ依存関係）をプログラム依存グラフ (PDG) と呼ばれるグラフとして表現する．不具合が再現した（不正な値が代入された）変数をスライシング基準（起点）としてグラフをたどることで，不具合の原因となり得るプログラムの一部分を特定する．プログラムスライシングには様々な変種が提案されていて [14]，近年に至るまで技術的な改良が続けられている [3]．また，不具合の位置を特定する精度についても実験的な評価が行われている [12]．

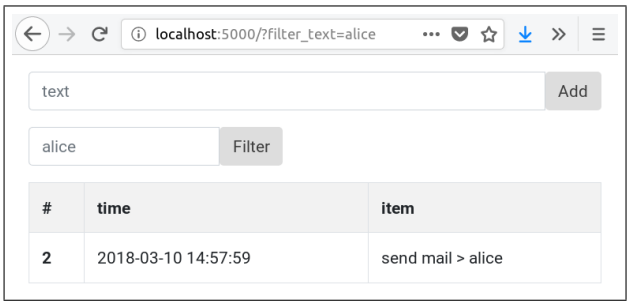
デルタデバッギングと呼ばれる手法には，プログラムの異なるバージョン（リビジョン）を同じ入力に対して実行し，不具合が再現するバージョンと再現しないバージョンを特定し，両者のソースコードの差分に不具合の原因があると推定する手法 [16] と，入力データを少しづつ変更していき，不具合を再現する最小の入力を生成する手法 [17] がある．

スペクトラムベース，あるいは，統計的デバッグと呼ばれる手法 [2][4] では，まず，対象となるプログラムに対する入力（テストケースなど）を多数用意する．それらの入力を与えてプログラムを実行し，プログラム中の文のそれぞれが実行されたか否かを記録しておく．不具合を起こした実行において，特に多く実行された文を不具合の原因であると推定する．

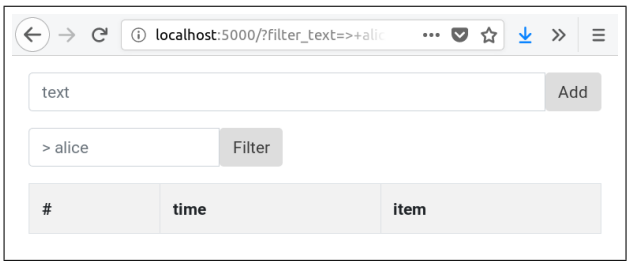
逆戻りデバッグ，あるいはキャプチャ＆リプレイと呼ばれる手法は，プログラムの実行トレースを記録しておくことで，従来のデバッガのステップ実行とは異なり，プログラムの実行時の任意の時点からプログラムの実行の順向きあるいは逆向きにステップ実行する（実行したかのように再現する）ことを可能にする [4][13]．



(a) メモ項目「send mail > alice」を追加した直後



(b) フィルタリング文字列「alice」を指定



(c) フィルタリング文字列「> alice」を指定（不具合）

図 1. メモアプリ実行例

データフローやデータの参照を主要な対象とする動的解析手法も提案されている．オブジェクトの参照を依存関係と捉え，ソフトウェアの機能間の依存関係を特定するオブジェクトフロー分析 [9]，不正なデータフローを検出する動的情報流解析 [10] が提案されている．近年では，静的解析と動的解析を併用し，依存関係を多段に辿って解析を進めていくハイブリッドなバグ位置特定の手法 [1][11] も提案されている．

3. トイプログラムへの適用例

あるトイプログラムに提案手法およびその実装を適用した例を示す．

```
@app.route("/")    ← トップページ
def index_page():
    cur = get_db(g).cursor()
    cur.execute("SELECT id, updated, item FROM memo ORDER BY updated DESC;")
    records = cur.fetchall()                                  } 記録日時の降順にデータベースからメモ項目を取得する

    filter_text = request.args.get('filter_text', None)
    if filter_text:                                           } フィルタリング文字列が設定されている場合、文字列を含むメモ項目のみを残す
        records = [r for r in records if filter_text in r[2]]

    …フォーマットして出力する（省略）…

@app.route("/add", methods=['POST'])    ← メモ項目の追加ボタンの処理
def add_request():
    item_text = request.form['item']
    item_text = bleach.clean(item_text.strip())               } フォームに記入されたメモ項目の文字列を取得し、サニタイズする

    if item_text:
        sql = "INSERT INTO memo (updated, item) VALUES (?, ?);"
        get_db(g).cursor().execute(sql, [datetime.now(), item_text])   } 空の文字列ではない場合には、データベースにメモ項目を追加する

    return redirect('/')    ← トップページにリダイレクト（してメモ項目を表示）する

@app.route("/filter", methods=['POST'])    ← フィルタリングボタンの処理
def filter_request():
    filter_text = request.form['filter']    ← フィルタリング文字列を取得する

    if filter_text:
        param_str = '?' + urllib.parse.urlencode({'filter_text': filter_text})
        return redirect('/' + param_str)                      } フィルタリング文字列を設定してトップページにリダイレクトする
    else:
        return redirect('/')
```

図 2. メモアプリのソースコード（抜粋）

3.1. 対象プログラム

対象となるプログラムはメモを書き留めるための web アプリケーションであり，web フレームワークとデータベースを再利用して実装されている．ユーザーがメモ項目を記入する機能，記入したメモ項目を一覧する機能，フィルタリング文字列を指定してその文字列が含まれるメモだけを一覧する機能を持つ．このフィルタリング機能には意図的に不具合が作り込まれている．図 1 の (a) から (c) に対象プログラムの画面のスクリーンショットを示す．(b) はフィルタリング機能が期待通りに動作している例，(c) はフィルタリング機能が不具合を起こしている例である．(c) では，フィルタリング文字列「> alice」を含むメモ項目を表示するように指定しているが，その文字列を含む「send mail > alice」というメモ項目が表示されていない．

図 2 に，対象プログラムのソースコードの一部を示す[1]．関数 `index_page` はメモ項目を一覧するページがリクエストされたとき，ページを生成して返す関数である．フィルタリング文字列が設定されていれば，メモ項目を選別してからページの生成を行う．関数 `add_request` はメモ項目を追加するボタン（画面の「Add」）が押されたときのリクエストを処理する関数である．入力されたテキストを新たなメモ項目としてデータベースに格納する．関数 `filter_request` はフィルタリングボタン（画

[1]対象プログラムのソースコード全体は `https://github.com/tos-kamiya/memo.py` を参照のこと．

```
def test_filter_entries_buggy(app):
    # メモ項目「send mail > alice」を追加する
    rv = app.post('/add', data=dict(
        item="send mail > alice"
    ), follow_redirects=True)

    # フィルタリング文字列「> alice」を設定する
    rv = app.post('/filter', data=dict(
        filter="> alice"
    ), follow_redirects=True)

    # フィルタリングの結果を取得する
    print(repr(extract_content_of_id(
        rv.data, 'items')))
```

図 3. メモアプリの不具合を再現するスクリプト (抜粋)

```
        1-1{    _:1 __main__//test_filter_entries_buggy
        1-2     t.py:17 .load
!2#2007 1-2:2   'send mail > alice'
        1-3-1{  t.py:19 werkzeug.test/FlaskClient/post
(snip)
!1#1010 1-4:1   'Ialice' '>␣alice'
        1-5-1{  t.py:22 werkzeug.test/FlaskClient/post
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-1{  L/flask/app.py:1598 memo//index_page
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-3-4-3-3-3-2-4-2-3}  L/werkzeug/urls.py:536 .ret
!1#1010 1-5-3-4-6-6-4-4-4-4-2-5-5-4-3-4-3-3-3-2-4-2-3:1 'Ialice' '>␣alice'
(snip)
        1-5-3-4-6-6-4-4-4-4-2-5-5-4-7}  T/memo.py:40 .ret
!1!2#1008  1-5-3-4-6-6-4-4-4-4-2-5-5-4-7:1 '⏭"Ialice⏭\n<tr><th>2</th><td>2018-03-11␣04:33:44</td⏭\n⏭
'⏭">␣alice⏭\nI\n⏭'
```

図 4. メモアプリのデータフローの視覚化（抜粋）

面の「Filter」）が押されたときのリクエストを処理する関数である．入力されたフィルタリング文字列をパラメータとして設定しつつメモ項目を一覧するページにリダイレクトする．

3.2. 不具合の原因

上述のフィルタリング機能の不具合の原因は，ユーザーが入力した文字列をサニタイズ（特殊な意味を持つ文字「&」や「>」などをエスケープ）する処理としない処理が混在するためにデータの一貫性がなくなることである．関数 `add_request` ではメモ項目の文字列をサニタイズしている（関数 `bleach.clean` を利用）のに対して，関数 `fiter_request` ではフィルタリング文字列をサニタイズしていない．結果として，関数 `index_page` でメモ項目をフィルタリングする処理ではサニタイズ済みの文字列の中から，サニタイズされていない文字列を検索することになる．不具合が再現する図 1 の (c) の実行例では，フィルタリング文字列にサニタイズによって置き換えられる文字「>」が含まれているため，検索結果が空になっている．

この不具合は，いわゆるクラッシュしないバグであり，プログラムは動作し続けるが出力は不正なものとなる．さらに，この例では，不具合の原因に関係する 3 つの関数の個々について単体では処理の誤りを議論することはできず，3 つの関数の仕様や処理，データフローを理解して突き合わせることが必要となる．不具合の修正としてフィルタリング文字列をサニタイズするよう処理を修正することにした場合には，修正される関数は `fiter_request` と `index_page` のいずれかとなる．

3.3. 提案するデバッグ補助手法の手順と適用例

提案手法では，対象プログラムを（不具合が再現する入力データを始めとして）いくつかの入力データを与えて実行し，取得した実行トレースからデータフローを特定し (4.1 で後述)，その後，データ値やデータフローにより実行トレースを絞り込み視覚化する (4.2 で後述) ことで分析を進める．ユーザー（開発者）は不具合に関係がありそうなデータ値とデータフローにより実行トレース内で絞り込みを行い，不具合の原因となり得る関数の候補を選びだし，そのソースコードを参照して不具合の原因であるかを確認する．

図 3 に，不具合を再現する入力を与えてメモアプリのプログラムを実行するスクリプトの一部を示す（このスクリプト，および，このスクリプトを修正したものを用意し，複数の実行トレースを取得して，提案手法の入力とした）．

図 4 に，提案手法を実装したツールによる解析結果を示す．実行トレースから差分データを含むデータフローを抽出し視覚化したものの一部である (抽出された関数呼び出しは 267 回と一見多く見えるが，web フレームワークによる関数間でのデータのとりまわしが大量に含まれており，ユニークなデータ値は 62 種類である．表 2 を参照)．各行に，データフローを識別するラベル（「!1」や「#1010」など．詳細は 4.1 で後述），コールツリー内での位置，呼び出された関数の名前（パッケージ名，メソッドの場合はクラス名も含む）や，実行されているソースコードの行（ファイル名と行番号），あるいは，引数や戻り値として現れたデータ値が示されている．

図中で「!2」が付加された行は，メモアプリに入力し

たメモ項目のデータに関するデータフローを示している．メモ項目のデータはデータベースを介してやり取りされる（一度データベースに格納され，その後表示のためにデータベースから取り出される）が，そのようなデータに対しても，提案手法ではデータフローを追跡することができている．

図中で「!1」が付加された行は，メモアプリに入力したフィルタリング文字列のデータに関するデータフローを示している．関数 index_page に，URL にエンコードされたパラメータとしてフィルタリング文字列が渡され（図中「!1#1010」が付加された行），検索結果の画面の HTML 記述が生成されている（図中「!1!2#1008」が付加された行）．さらに，正しい検索結果（「<tr>」で始まる文字列）と不具合である空の検索結果の差分も示されている．この関数 index_page は，不具合として現れるデータを生成した処理を含み，（前述のように）不具合の修正のために変更される関数の候補でもある．

4. 提案するデバッグ補助手法

提案するデータ値の差異とデータフローの視覚化によるデバッグ補助手法は，基本的には，データ値の差異に基づくデータフロー解析手法 [8] と，実行トレースの検索のための技術 [6], [7] を組み合わせたものであり，特にデバッグを補助するために応用したものである．

4.1. データ値の差異に基づくデータフロー解析

対象プログラムに対して，データ（入力や出力，変数）の値が異なる（**変異**させた）2 つの実行トレースを用意し，それらの実行トレース中の対応する位置に現れるデータ（関数の実引数や戻り値として）の値の「差異を調べる」ことで，プログラムの実行中にシステムの中でそのデータがどのように伝播していったかを特定する解析手法である．

(1) 複数の実行トレースから同じ関数呼び出し列を抽出する前処理

異なる入力を与えて実行したときの実行トレースの間では，一般に，呼び出される関数や呼び出しの順序が異なる．実行トレース中の対応する位置にに現れるデータ値を比較できるようにするために，呼び出される関数や呼び出しの順序が異なる部分を取り除く前処理を行う．この前処理は，与えられた 2 つの実行トレースをコールツリーに変換し，それらコールツリーの根から順に幅優先で節点同士を比較していき，異なる関数呼び出しが現れたら，その節点から葉まで（その節点を根とする部分木）を相違点であるとみなして取り除く．この前処理により，呼び出されている関数列（どの関数がどの順序で呼び出されているか）は全く同じで，引数や戻り値として渡されるデータ値のみが異なる 2 つの実行トレースを生成する．

(2) データ値の差異からのラベルの生成

2 つの実行トレースの間の差異を表すラベルとして「**変異効果ラベル**」と「**値ペアラベル**」の 2 種類がある [8]．2 つの実行トレースの間で，同じ位置にあるデータ（実行トレース間で対応する関数呼び出しの同じ引数）の値が異なるときに，それらデータ値の違いは実行トレースを生成するのに用いられた入力データの変異に起因することを意味する変異効果ラベルを付与する．図 4 の例では，左端にある「!1」や「!2」が変異効果ラベルである．

変異効果ラベルは変異（すなわち入力データ値の差異）ごとに 1 種類づつ生成されるため，より詳細にデータフローを区別するために値ペアラベルを導入する．変異効果ラベルが付与されたデータ値のそれぞれについて，2 つの実行トレース内の対応する位置のデータ値のペアを生成し，値のペアが等しい部分には同じ値ペアラベルを付与する．例えば，一方の実行トレース内で変異効果ラベルが付与されたデータ値が順に $1, 2, 1, 1$ であり，もう一方の対応する実行トレース内でのデータ値が順に $10, 20, 11, 10$ であるとき，これらのデータ値から順に $(1, 10), (2, 20), (1, 11), (1, 10)$ という値ペアが生成される．前者の実行トレース中で現れる 3 つのデータ値 1 のうち，1 番目と 3 番目には値ペア $(1, 10)$ に相当する値ペアラベルが付与されるが，2 番目の 1 には異なる値ペア $(1, 11)$ に相当する値ペアラベルが付与され，従って，2 番目のデータ値 1 は 1 番目や 3 番目のデータ値とは異なるデータフローにあると判定される．図 4 の例では，「#1008」,「#1010」,「#2007」が値ペアラベルである．

4.2. データ値の差異を利用したデータフローの視覚化

実行トレース内で呼び出しが起きた順に関数呼び出し（あるいは関数呼び出しからのリターン）を並べて表示し，呼び出しの深さに応じてインデントする（図 4 はこ

の視覚化の例になっている）．このとき，前述の変異効果ラベルや値ペアラベルを指定してフィルタリングを行うことで，着目しているデータ値に関連するデータフローのみを絞り込んで表示することができる．

実行トレースをフィルタリングする手順は次の通りである．指定されたラベルの集合を L として，

(a) ある関数の呼び出し c の引数や戻り値のデータ値にラベルが付与されていて L の要素であるる場合にはその関数の呼び出し c とそのリターン，引数や戻り値のデータ値を表示する．

(b) ある関数の呼び出し c の中で，直接または間接的に別の関数を呼び出し d を行っていて，d が上の (a) で表示される場合には，c やそのリターンを表示する．

提案手法を実装したツールでは，ユーザーがラベルを指定してフィルタリングすることにより，より少数のデータフローや関数呼び出しのみに着目して分析を進める（フィルタリングの利用例は 5.3 で後述）．

4.3. 変異させた入力データの準備

提案手法では，対象プログラムに対して異なった入力データを与えて実行することにより収集した複数の実行トレースが必要となる．不具合を再現する入力データを「少しだけ」(2 つの実行トレースの間で，なるべく，条件分岐の同じ枝が実行されるように）変更する（変異させる）ことで，複数の入力データを用意する．実行トレース間で呼び出される関数が異なっている部分は 4.1 の (1) の処理の対象となりデータフローの検出が行えないため，そのような部分を減らすことが望ましい．入力データを変異させる作業はユーザー（開発者）に任されている．その理由は，提案手法において，不具合に関係しつつ呼び出される関数に影響を与えないような入力データの変異を作成するためには，対象プログラムの処理内容やそのドメイン，不具合の内容に関する知識からの判断が必要となり，現状では自動的に行うことが難しいためである．

ただし，入力データの変異が小さいものであるほど，実行トレース中にデータ値の差異が現れる箇所は少なくなり，結果的に，特定できるデータフローも少なくなってしまうことが想定される．なるべく多くのデータフローを抽出するため，変異させた入力データを多数用意して，その結果をマージすることが可能になるツールを実装に含めた．これにより，実行トレース中のより広範囲にデータ値の差異を出現させることで，より広範囲にデータフローを抽出できるようにしている．例えば，3 の適用例では，メモ項目の文字列を変異させたものと，フィルタリング文字列を変異させたもの，2 つの変異を利用している（具体的な変異の内容については 5.2 で後述）．

5. 予備的な実験

提案手法を性質の異なる 2 つのプログラムに適用した例を示す．ひとつは 3 で前述したメモアプリに意図的に作り込まれている不具合を提案手法により特定できるか確認するための適用例であり，もうひとつは，オープンソースのライブラリの実際のデバッグに適用した例である．これらの実験では 3.3 の冒頭で示した手順を何度か繰り返して行うことにより不具合の原因とみなせるソースコードの記述を特定した．

5.1. 実装および実行環境

提案手法を実装したツールは，Python で記述されたプログラムを解析の対象とし，ツール自体も Python で記述されている．対象プログラムを実行し実行トレースを取得するツールは 1,574 行，実行トレースを解析しデータ値の差分からデータフローを抽出し視覚化するツールは 2,426 行である．

ツールを実行した PC は，CPU Intel Core i7-6800K CPU 3.40GHz，RAM 128GiB を備える．ツールはシングルスレッドで実行したため，以降で示すツールの実行時間（経過時間）は，ほぼそのまま CPU 時間となっている．対象プログラムの実行（および実行トレースを取得するツール）を実行するのに利用した VM（仮想機械，インタプリタ）は CPython 3.6.3 (with GCC 7.2.0) であり，データフローを抽出し視覚化するツールの実行に利用した VM は Pypy 3.6.3 (PyPy 5.10.1 with GCC 6.2.0) である．

5.2. メモアプリへの適用例

3 で導入したメモアプリは，提案手法の説明のために作成されたトイプログラムであり，ソースコードは主として Python で記述され，5 つの関数，全体で 102 行というごく小規模なものであるが，データベースや web フレームワークといったライブラリを利用しているため，

プログラムを実行すると多くの関数呼び出しが行われる．不具合の特定のためには，それらのライブラリの仕様や動作も含めて調査していく必要がある．

セットアップ

実行トレースを収集するために，対象プログラムの入力データとして，不具合を再現する入力データと，それを次のように変異させた入力データを準備した．

- **変異 m vs M** データベースに格納するメモ項目として「send mail > alice」，対，一部を大文字に変えた「send MAIL > alice」
- **変異 a vs g** フィルタリング文字列として「alice」，対,「> alice」

変異`m vs M`はメモ項目として入力されたデータのデータフローを追跡するため，変異`a vs g`はフィルタリング文字列のデータフローを追跡するとともに，不具合が再現する実行トレースと再現しない実行トレースを作るためのものである．

解析結果

変異`m+a`，`m+g`，`M+a`の組み合わせのそれぞれに対して対象プログラムを実行する（web アプリのサーバーにリクエストを送り，そのレスポンスを確認する）単体テストプログラムを作成して利用した．表 1 に実行トレースの取得とデータフローの抽出に要した時間やピーク（レジデント）メモリ消費量を示す．

表 2 に取得した実行トレースの大きさ，および，抽出されたデータフローを含む（すなわち，抽出された異効果ラベルや値ペアラベルのいずれかを含むという条件でフィルタリングした）実行トレースの部分列の大きさを示す．実行トレース全体ではのべ 21 万回以上の関数呼び出しがあり，モジュールのローディングのための処理などを取り除いた後の，エントリポイント以降の部分列でものべ 6 千回以上の関数呼び出しがある．提案手法により，ラベルが付加されたデータ値を含む関数呼び出しに着目することで，267 回にまで絞り込めたことがわかる．

この実験では，データフローを示すラベルのうち，基本型のデータ値に付加されたラベルのみを利用し，オブジェクトに付加されたラベルは利用しなかった（web アプリケーションであるため通信路でデータが変換が行われる，データベースへのデータの格納・取得に伴い異なるオブジェクトに変換される，などの理由により，オブジェクトによりデータフローを追跡するのは手間がかかるため）．表 2 でも，オブジェクトに付加されたラベルは数字に入れていない．

表 1. メモアプリの解析に要した実行時間およびメモリ消費量

ステップ	実行時間 (秒)	メモリ消費量 (KiB)
実行トレースの取得 (1 回あたり，最大)	101	44,220
データフローの抽出	60	126,496

5.3. DeepDiff への適用例

対象となったプロダクトである DeepDiff(`https://github.com/seperman/deepdiff/`) はオープンソースのライブラリであり，Python の種々のデータ，特にリストやタプルなどの構造を持つデータの差分を抽出する機能を持つ．ソースコードの規模はプロダクションコード 2,852 行，テストコード 2,716 行である．

5.2 のメモアプリと比較すると，DeepDiff はそれ自身が複雑なロジック（データ構造を再帰的に探索して差分を計算する）を持ち，再利用するライブラリが少ない（小さい）．また，データベースや web フレームワークなどのデータを変換するライブラリを利用していないため，オブジェクトに付加されたラベルをデータフローの追跡のために有効に利用できると想定される．また，不具合は実験のために作り込まれたものではなく，github 上ではプロジェクトの issue として報告されている，論文投稿時点で未解決のものである．

セットアップ

不具合の内容は，NumPy（数値演算ライブラリ）の配列の差分を DeepDiff で正しく求められないことがあるというものである（`https://github.com/seperman/`

表 2. メモアプリからのデータフローの抽出

実行トレース	関数呼び出し (回数)
m+g	212,601
エントリポイント以降	6,827
データフロー (ユニークなデータ値)	267 (62)

```
import deepdiff
import numpy as np

def main():
    d1 = np.array([2, 3], dtype=int)
    d2 = np.array([2, 3], dtype=int)
    ans = deepdiff.DeepDiff(d1, d2, verbose_level=2)
    print(repr(ans))

if __name__ == '__main__':
    main()
```

図 5. DeepDiff の不具合を再現するスクリプト

deepdiff/issues/97)．不具合を再現するスクリプトを図 5 に示す．このスクリプトは，要素の型として int を指定して生成した NumPy の配列 (numpy.array)2 つを比較する処理を行うものであり，全く同じ内容の配列を比較しているために「差分がない」という出力が期待されるが，実行すると「データが処理されなかった」という出力が得られた．このスクリプトの「np.array([2, 3], dtype=int)」の部分 2 箇所を，「np.array([2.0, 3.0], dtype=float)」に変更したものは，実行すると期待通りの「差分がない」という出力が得られるため，不具合の原因は「DeepDiff が NumPy の配列に未対応である」といった単純なものではないと推定された．

対象プログラムの入力データとして，不具合を再現する図 5 のスクリプト（**不具合再現スクリプト**とする）と，前述の int を float に変異させたもの（**変異スクリプト**とする）を利用した．

解析結果

不具合再現スクリプトと変異スクリプトを実行して取得した 2 つの実行トレースを入力として解析を行った．表 3 に実行トレースの取得とデータフローの抽出に要した時間やピーク（レジデント）メモリ消費量を示す．

ソースコード内の不具合の原因だと判断した部分に至るまでに，表示するラベルの範囲を変更してフィルタリングを行うことにより，計 3 回データフローの視覚化を行うこととなった．表 4 にこれら 3 回の視覚化によって得られたデータフローの部分列の大きさを示す．最初に視覚化したデータフロー (1) は 5.2 と同様に基本型のデータ値に付加されたラベルのみを利用したものである．この視覚化の結果では，対象プログラムの入力データ値の違い（int 型と float 型の違い）や，出力されるメッセージの違い（「差分がない」と「データが処理されなかった」）は確認できたが，差分の計算に相当する処理はデータフローには含まれなかった．次に，出力されるメッセージの生成に用いられたデータ値を確認するために，データフロー (2) として，基本型のデータのみではなくオブジェクトに付加されたラベルも含むデータフローの視覚化を行った．このデータフローを含む実行トレースの部分列は 203 回の関数呼び出しを含むが，分析に当たって参照したものは，メッセージの生成の直前のデータ値のみである．このデータ値 (後述の図 6 ではオブジェクト ID 7fce77d22b48) は「deepdiff.diff/DeepDiff」という型を持っていたため，対象プログラムの処理に深く関係していることが期待された．3 度目に，基本型のデータ値に付与されたラベルとこのオブジェクト 7fce77d22b48 に付与されていた値ペアラベル「#150」を含むデータフローの視覚化を行ったものは，データフロー (3) として示される 13 回の関数呼び出しを含んでいた．図 6 に視覚化の結果を示す．この図に含まれる関数呼び出しのうち，オブジェクト 7fce77d22b48 が 3 回目に利用されている箇所（メソッド__skip_this）の処理を確認したところ，差分計算の対象となるデータを型により識別するロジックが含まれていた．図 5 の不具合再現スクリプトおよび変異スクリプトで利用されているデータ型を（スクリプトに「print(type(d1[0]))」などの文を追加して実行することにより）確認したところ，配列の生成の直後に要素の型はそれぞれ numpy.int64 と numpy.float64 に

表 3. DeepDiff の解析に要した実行時間およびメモリ消費量

ステップ	実行時間 (秒)	メモリ消費量 (KiB)
実行トレースの取得 (1 回あたり，最大)	11	32,548
データフローの抽出	3.5	79,976

表 4. DeepDiff からのデータフローの抽出

実行トレース	関数呼び出し (回数)
不具合再現スクリプト	29,728
エントリポイント以降	548
データフロー (1)	8
データフロー (2)	203
データフロー (3)	13

```
                1-1{          _:1  __main__//main
                1-2           ddbad-target.py:5  .load
!1              1-2:1         2       2.0
!1              1-2:2         3       3.0
                1-3-1{        ddbad-target.py:7  DeepDiff/__init__
!1!2#150#2138   1-3-1:1       <DeepDiff 7fce77d22b48>  <DeepDiff 7f602eb48af0>
                1-3-4-1{      L/deepdiff/diff.py:646  deepdiff.diff/DeepDiff/__diff
!1!2#150#2138   1-3-4-1:1     ↑       ↑
                1-3-4-2-1{    L/deepdiff/diff.py:1073 deepdiff.diff/DeepDiff/__skip_this
!1!2#150#2138   1-3-4-2-1:1   ↑       ↑
(snip)
                1-4-1{        ddbad-target.py:8  agm_trace.hook//_repr
!1!2#150#2138   1-4-1:1       ↑       ↑
                1-4-2}        L/agm_trace/hook.py:126 .ret
!1#101          1-4-2:1       "{I'unprocessed':␣['root[1]:␣3␣and␣3',␣'root[0]:␣2␣▸▸  '{}'I
                1-5-1{        ddbad-target.py:8  agm_trace.hook/Out/write
!1#101          1-5-1:2       ↑       ↑
```

この図で「↑」とある部分は，直前と同じデータ値が出現していることを示す．

図 6. DeepDiff のデータフロー (3) の視覚化（抜粋）

```
>>> import numpy as np

>>> isinstance(np.int64(2), int)
False

>>> isinstance(np.float64(2.0), float)
True
```

図 7. NumPy の独自の型のデータと基本型との関係

なっていることが判明した．さらに，これらの型と上述のロジックで比較対象として記述されている型との関係を調査した（その一部を図 7 に示す）．

これらの調査から，不具合の原因として次が推定された．

- 不具合再現スクリプトおよび変異スクリプトにおいて，配列の要素は配列が生成された時点で NumPy ライブラリ独自の型（`numpy.int64` および `numpy.float64`）になっている．上述のロジックはこれらの型に関する記述を含まないため，これらの型自体は差分計算の対象ではない．
- ただし，`numpy.float64` 型のデータは対応する基本型 `float` のインスタンスであるとみなされる（それに対して，`numpy.int64` と基本型 `int` との間には関係は見られない）．`float` 型は上述のロジックにおいて差分計算の対象であるため，結果として，変異スクリプトでは差分が計算された．

この推定が正しいかを確認するために，上述のロジックのソースコードを修正して `numpy.int64` を差分計算の対象に含めるようにしたところ，不具合再現スクリプトの出力が変化し不具合が再現しなくなった．

6. まとめと展望

クラッシュしないバグのデバッグを目的とした動的解析手法として，実行トレース間のデータ値の差分に注目したデータフローの抽出手法を提案した．提案手法を実装したツールの予備的な適用実験を行い，あるオープンソースプロダクトのある既知（であるが未修正の）不具合に適用し，その原因を特定した．

提案手法は研究の初期段階にあり，適用対象のスケーラビリティやデータフロー抽出の精度の評価を含めて，より大きなプロダクトを対象とした実験による評価が必要である．また，提案手法の適用性や分析作業の容易さを改善するために，次のような課題に取り組む必要がある．

- 解析対象プログラムの入力（変異された入力データ）をユーザー（開発者）が手作業で準備する必要があるため，どのような変異を行うべきかの知見を積み上げる（4.3 を参照）．
- 解析対象プログラムの入力として単体テストを流用することを想定し，デバッグに有効なデータフローを抽出できる単体テストを選択する手法を提案する．
- 変異ペアラベルや値ペアラベルが多くなった場合の分析作業を容易にするため，ラベル間の関係（階層

性など）に基づいてラベルを整理したり追跡したりするためのツールを提供する．

謝辞

本研究は JSPS 科研費 16K12412 の助成を受けたものである．

参考文献

[1] Baah, G., Podgurski, A., Harrold, M., “The Probabilistic Program Dependence Graph and its Application to Fault Diagnosis”, *IEEE Trans. Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[2] Dallmeier, V., Lindig, C., Zeller, A., “Lightweight Defect Localization for Java”, *Proc. the 19th Annual Meeting of the European Conference on Object-Oriented Programming (ECOOP ’05), LNCS 3568*, Springer-Verlag, pp. 528-550, July, 2005.

[3] 石尾, 仁井谷, 井上, “プログラムスライシングを用いた機能的関心事の抽出”, コンピュータソフトウェア, 26 巻, 2 号 pp. 2_127–2_146, 2009.

[4] Jones, J.A., Harrold, M.J., “Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique”, *Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering* (ASE ’05), pp. 273-282, 2005.

[5] Joshi, S., Orso, A., “SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions”, *Proc. the 23ed IEEE International Conference on Software Maintenance* (ICSM ’07), pp. 234–243, 2007.

[6] 神谷 年洋, “And/Or/Call グラフの提案とソースコード検索への応用”, 電子情報通信学会技術研究報告, vol. 113, no. 269, pp. 173–178, 2013.

[7] 神谷 年洋, “逆戻りデバッグ補助のための嵌入的スパイの試作”, 電子情報通信学会技術研究報告, vol. 116, no. 127, SS2016-9, pp. 87–92, 2016.

[8] 神谷 年洋, “実行トレース間のデータの差異に基づくデータフロー解析ツール”, 電子情報通信学会技術研究報告, vol. 116, no. 136, pp. 55–60, 2017.

[9] Lienhard, A., Greevy, O., Nierstrasz, O., “Tracking Objects to Detect Feature Dependencies”, *Proc. the 15th International Conference on Program Comprehension* (ICPC ’07), pp. 59–68, 2007.

[10] Masri, W., Podgurski, A., “Algorithms and Tool Support for Dynamic Information Flow Analysis”, *Information and Software Technology*, vol. 51, no. 2, pp. 385–404, 2009.

[11] 中野, 大沼, 小林, 石尾, “動的データ依存集合の発生確率を用いた欠陥箇所特定支援手法の実装及び評価”, 電子情報通信学会技術研究報告, vol. 114, no 510, pp. 19–24, 2015-03.

[12] 西松, 西江, 楠本, 井上, “フォールト位置特定におけるプログラムスライスの実験的評価”, 電子情報通信学会論文誌 D, vol. J82-D1, no. 11, pp. 1336–1344, 1999.

[13] 櫻井, 増原, 古宮, “Traceglasses : 欠陥の効率良い発見手法を実現するトレースに基づくデバッガ”, 情報処理学会論文誌プログラミング (PRO), 3 巻, 3 号, pp. 1–17, 2010.

[14] Tip, F., “A Survey of Program Slicing Techniques”, *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[15] Weiser, M., “Program Slicing”, *IEEE Trans. Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[16] Zeller, A., “Yesterday, My Program worked. Today, It Does Not. Why?”, *Proc. the 7th European Software Engineering Conference* (ESEC/FSE-7), pp. 253–267, 1999.

[17] Zeller, A., Hildebrandt, R, “Simplifying and Isolating Failure-Inducing Input”, *IEEE Trans. Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

不具合混入コミットの推定手法間での整合性比較と考察

北村 紗也加
京都工芸繊維大学 大学院工芸科学研究科 博士前期課程 情報工学専攻
s-kitamura@se.is.kit.ac.jp

水野 修
京都工芸繊維大学 情報工学・人間科学系
o-mizuno@kit.ac.jp

要旨

ソフトウェアの複雑さと重要性は日々増してきており，ソフトウェアの品質を高い水準で保つことが重要視されている．このような現状においてはソフトウェアの品質予測は重要な研究テーマであり，どのような手法で品質予測を行うかに注力されてきた．不具合混入コミットを推定する手法では，その評価を行うためには正解データが必要であり，不具合の正解データ *(*真値*)* として *Commit Guru* による不具合混入コミットの情報が多く利用されている．しかしながら，*Commit Guru* の不具合混入コミットの情報の正解データとしての信頼性は不明である．

本研究では，その信頼性に対する検証を行った．不具合混入コミット推定手法である *SZZ* アルゴリズムを用いて，同じ不具合データに対する結果の整合性を比較し，その結果を考察した．*Commit Guru*，および *SZZ* アルゴリズムを用いた不具合コミット推定結果の差異において検証を行った結果，*Commit Guru* の方がより優れた不具合コミット推定の結果を示し，正解データとしての可能性を示すものとなったが，その信頼性は十分であるとは言い難い．

1 はじめに

近年，ソフトウェアの重要性と複雑さは増しつつある．このような現状におかれたソフトウェア開発の現場では，ソフトウェアに混入した不具合を修正するために膨大な時間とエフォートが割かれており，またそれに伴うコストも膨大であるため [1]，ソフトウェアの品質を高い水準で保つことが重要視されている．したがって，最近のソフトウェア工学の研究ではソフトウェアの分析とソフトウェアの品質予測に焦点を当てたものが多く見受けられる [2].

ソフトウェア工学における品質予測の研究を活性化させたものとして，Śliwerski，Zimmermann，Zeller によって提案された，ソースコードやリポジトリデータを用いて不具合を混入したと推定されるコミットを発見する SZZ アルゴリズム [3] がある．SZZ アルゴリズムはバージョン管理システムにおいて変更履歴を辿り，不具合を混入したコミットを特定する．この SZZ アルゴリズムとはまた別のアルゴリズムで不具合コミットを特定するツールを公的に入手可能にしたものとして，Commit Guru [4] 等が存在する．

Commit Guru は，登録された Git リポジトリにおける全てのコミットに対し分析を行い，13 のメトリクスを計算し，不具合を混入したコミットを推定する．また，その分析結果を利用者らに無償で提供している．

Commit Guru による不具合混入コミット（以降，不具合コミットと呼称）の情報は，昨今の研究において不具合の正解データとして利用されることが多い．しかしながら，Commit Guru が推定する不具合コミットは簡易的な手法を用いており，正解データとしての信頼性は不明である．そこで，本研究では Commit Guru および SZZ アルゴリズムを用いて，分析結果の整合性を比較し，Commit Guru の正解データとしての信頼性について考察を行う．

2 準備

2.1 バージョン管理システム

バージョン管理システムとは，コンピュータ上で作成・編集されるファイルの変更履歴を管理するためのシステムである．本研究で用いた SZZ アルゴリズム，および Commit Guru はバージョン管理システムの情報を用いて不具合コミットを推定しており，バージョン管理システ

ムとして分散型バージョン管理システムである Git を使用した．

2.2 バグトラッキングシステム

バグトラッキングシステムとはプロジェクトのバグを登録し，そのバグの修正状況を追跡するシステムであり，バグの履歴管理や検索を行うことができる．本研究で用いた SZZ アルゴリズムは，バグトラッキングシステムから取得した対象プロジェクトの不具合データを用いて不具合コミットを推定しており，バグトラッキングシステムとして Bugzilla と Apache's JIRA issue tracker を使用した（表 1）．

2.3 SZZ アルゴリズム

SZZ アルゴリズムは不具合混入コミット推定手法の中で最もよく知られている手法である．

本研究で使用した SZZ アルゴリズムは，バージョン管理システムのバージョンアーカイブとバグトラッキングシステムの不具合データを用いて，次の 2.3.1 節，2.3.2 節で述べる 2 つのステップを経ることにより不具合コミットの推定を行う．

2.3.1 修正された箇所の特定

バージョン管理システムのバージョンアーカイブには，変更に関する情報が含まれている．バージョン間において同じ意図で行われてた変更を特定するため，バージョンアーカイブから不具合が混入されたコミットの日時，不具合を最後に修正されたコミットの日時を抽出し，バグトラッキングシステムの不具合データと照らし合わせる．抽出されたデータが不具合の修正と予測されるのであれば，この不具合を修正したであろうコミットに対し，修正したことを示す FIX タグをつける．

バージョン管理システムのバージョンアーカイブには変更の目的が欠けており，行われた修正が不具合を修正したのか，機能追加であるか判断することができない．ログメッセージのみから目的を予測することは出来る [5] が，不具合に関する簡単なレポート，不具合のステータスや解決策が格納されている不具合データを組み合わせることで，不具合予測の精度向上を図る．

[1]問題とは不具合，機能要求，改善，タスクなど不具合以外にも様々なものを指すが，本研究においては不具合のみを対象としたバグトラッキングシステムとして用いている．

2.3.2 不具合が混入された箇所の特定

FIX タグがつけられたコミットを取得する．このコミットにおいて変更されたファイルに対し，バージョン管理システムの diff コマンドを用いてコミットにおける変更箇所を抽出する．

もし変更箇所が見つからなければ FIX タグの削除を行い，変更箇所が見つかれば，その変更はどのコミットにおいて行われたのかを特定する．変更箇所に対してバージョン管理システムの blame コマンドと正規表現を用いて，その変更を行ったコミットの日時を抽出する．抽出されたデータから不具合を混入したと予測されるコミットに対し，不具合コミットであることを示す BUG タグをつける．

2.4 Commit Guru

Commit Guru は Kamei らの変更レベルの不具合予測モデル [6] を Web アプリケーションとして実装したものであり，Rosen らによって公開されている．本研究において，Commit Guru での不具合混入ラベル，メトリクスを分析データの整合性比較に用いる．Commit Guru の機能である (1) 不具合コミットのラベル付け，(2) メトリクスの測定，および (3) ダンプデータの提供について次に述べる．

2.4.1 不具合コミットのラベル付け

Commit Guru はコミットに対し，不具合を混入したコミット (Buggy) とそれ以外のコミット (Clean) を不具合混入ラベルにおいて TRUE/FALSE でラベル付けを行う．不具合コミットは不具合を修正したコミットから推定できる．

始めに，Commit Guru は Hindle らの研究 [7] におけるコミットを分類するためのキーワードリストを元に，コミットメッセージの解析を行い，各カテゴリに分類する．ここでは，' bug ', ' fix ', ' wrong ', ' error ', ' fail ', ' problem ', ' patch ' といったワードを含むコミットが不具合を修正しているであろうコミットとし，Corrective に分類する．

次に，修正を行ったコミットから不具合を混入したであろうコミットを特定する．まずバージョン管理システムの diff コマンドを用いて，どの行が修正を行ったコミットによって変更されたのかを特定する．ここで変更された行

表 1. 使用したバグトラッキングシステムとそのメトリクス

システム	システム概要	メトリクス	メトリクス概要
Bugzilla	Mozilla Foundation によって開発されたウェブベースのバグトラッキングシステム	Bug ID Opened Changed	不具合に対して一意に与えられる ID 不具合が発見された日付 不具合が最終的に修正された日付
Apache's JIRA issue tracker	Atlassian によって開発された商用の問題トラッキングシステム[1]	Issue key Created Resolved	問題に対して一意に与えられるキー 問題が作成された日付 問題が解決された日付

に対して，バージョン管理システムの annotate/blame コマンドを用いることで，それらの修正されたソースコードがどのコミットで追加されたのかを特定する．ここで見つかったソースコードを組み込んだコミットを，不具合を混入したであろうコミットとしてラベル付けする．

2.4.2 メトリクスの測定

測定されるメトリクスを表 2 に示す．Kamei らの不具合予測モデルで利用されているものと同様の数値メトリクス 13 個，および論理メトリクス 1 個から成る変更に関するメトリクスである．

2.4.3 ダンプデータの提供

分析した Git リポジトリのダンプデータは CSV 形式で提供されている．このダンプデータには，2.4.2 節において述べた 14 個の計測したメトリクスに加え，コミットハッシュやそのコミットの分類カテゴリなどが格納されている．本研究では，このダンプデータを用いて分析を行っている．

2.5 マン・ホイットニーの U 検定

マン・ホイットニーの U 検定 (MannWhitney U test) とは，ノンパラメトリック検定のひとつであり,『対応のない 2 群が同じ分布の母集団から構成されている』とする帰無仮説に基づいて検定する．

データサイズ n_1 のデータ D_1，およびデータサイズ n_2 のデータ $D_2(n_1 \leq n_2)$ から成る 2 つのサンプルデータ群に対し，両郡のデータの値が小さい順に順位を割り当て，D_1, D_2 のそれぞれの順位和 R_1, R_2 を求める．同じ値のデータが存在した場合には，それらが異なると考えた場合の順位の平均値を割り当てる．このとき，D_1, D_2 の順位和の合計は次のようになる．

$$R_1 + R_2 = \frac{N(N+1)}{2} \tag{1}$$

これより，両郡のデータサイズ，順位和を用いて，次の式を用いて統計量を求める．

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2} \tag{2}$$

$$U_2 = R_2 - \frac{n_2(n_2+1)}{2} \tag{3}$$

式 (2)，式 (3) によって求めた U_1, U_2 において小さい方の値を U とする．U は有意性を調べるときに用いられる．サンプルサイズが大きい場合，U は正規分布に近似できる．このとき，z は次式で表される．

$$z = \frac{U - m_U}{\sigma_U} \tag{4}$$

ここで，m_U および σ_U は U の平均および標準偏差であり，有意性を調べることができる標準正規偏差である．m_U および σ_U は次の式で表される．

$$m_U = \frac{n_1 n_2}{2} \tag{5}$$

$$\sigma_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}} \tag{6}$$

タイ値が存在した場合，次の式で σ の補正を行う．

$$\sigma_{\text{corr}} = \sqrt{\frac{n_1 n_2}{12}\left((n+1) - \sum_{i=1}^{k} \frac{{t_i}^3 - t_i}{n(n-1)}\right)} \tag{7}$$

ここで，$n = n_1 + n_2$ であり，t_i はランク i を共有するタイ値の数であり，k はランクの数である。

本研究においては，Python の scipy パッケージを用いてこのマン・ホイットニーの U 検定を使用した．

3 実験

3.1 研究設問

先に述べたとおり，不具合混入の正解データ (真値) を作成する手法として，SZZ アルゴリズムや Commit Guru

表 2. メトリクス一覧

名前	定義	説明	関連研究
la	変更によって追加されたコード行数	追加されるコード行数が多いほど不具合が混入しやすい.	コード行数の相対的なメトリクスは欠陥モジュールにおいて効果の高い指標である [8,9].
ld	変更によって削除されたコード行数	削除されるコード行数が多いほど不具合が混入しやすい.	
lt	変更前の総コード行数	大きいファイルほど，そのファイル内の変更は不具合を混入している可能性が高い.	大きいファイルほど不具合を多く含む [10].
ns	変更されたサブシステム数	修正したサブシステム数が多い変更ほど，不具合が多くなりやすい.	変更における不具合の確率はサブシステム数が増えるにつれ高くなる [11].
nd	変更されたディレクトリ数	修正したディレクトリ数が多い変更ほど，不具合が多くなりやすい.	変更されたディレクトリの数が多いほど，不具合を混入する機会が増える [11].
nf	変更されたファイル数	関係するファイル数が多い変更ほど，不具合が多くなりやすい.	モジュール内のクラス数はリリース後における欠陥が現れやすい特徴である [12].
ndev	修正されたファイルを変更したことがある人数	コーディングデザインの違いの観点から，ファイルの開発に携わった人数が多いほど不具合が混入しやすい.	ファイルに携わった人数が多いほど多くの不具合を含んでいる [13].
age	修正された全てのファイルにおいて前回の変更から経過した平均日数	直近の変更ほど，不具合を混入する原因になりやすい.	昔の変更より最近の変更の方がより不具合に寄与する [14].
nuc	修正されたファイルにおける変更の個数	以前の変更を探し出す必要があるため，nuc が大きいほど不具合を混入しやすい.	修正したファイルの範囲が広いほど，複雑さが増す [15].
exp	変更を行った開発者の総コミット数	経験豊富な開発者ほど不具合を混入しにくい.	プログラマの経験は不具合の確率を著しく低下させる [11].
rexp	age によって重み付けされた exp	最近そのファイルに対しよく変更を行なっている人物はそのファイルに対しての知識が深いと考えられるため，不具合を混入しにくい.	
sexp	変更されたサブシステムに対するこれまでの変更の数	サブシステムについてよく知っている人物は不具合を混入しにくい.	
entrophy	変更が多くのファイルに渡って行われているほど大きくなる値	ファイルに跨って分散している多くの変更を探し出す必要があるため，高いエントロピーを持つ変更は不具合が多くなりやすい.	分散した変更は不具合を混入しやすい [15, 16].
fix	変更が不具合を修正したかどうか (TRUE / FALSE)	不具合修正は初期実装における欠陥を意味するため，その近辺で欠陥が存在する可能性が高い.	不具合を修正した変更は新機能を実装した変更よりも不具合を混入しやすい [17,18].

が知られているものの，手法間ので正解データの違いについては検証が進んでいない.

本研究では次に示す研究設問（Research Question）を設け，検証を行う.

- **RQ1: Commit Guru と SZZ アルゴリズムにおいて，それぞれの不具合混入コミット推定の結果にはどのような特徴が存在するか.**
- **RQ2: Commit Guru と SZZ アルゴリズムにおいて，不具合混入コミット推定性能の差はどの程度か.**

3.2 対象プロジェクト

本実験では，表 3 に示すプロジェクトを対象に分析を行った.

3.3 準備

使用したバグトラッキングシステムの不具合データを表 4，Commit Guru より得られた対象プロジェクトのダンプデータの詳細を表 5 に示す.

以降において，指定したコミット群を表 6，表 7 のように呼称する．また，(BUG_{szz}, $CLEAN_{szz}$)，(BUG_{guru}, $CLEAN_{guru}$) のペアを，それぞれ CF_{szz}，CF_{guru} と呼称す

表 3. 対象プロジェクト

Project	機能	リポジトリ URL
Log4j	Java のロギングツール	https://github.com/apache/log4j
OpenJPA	Java Persistance API 仕様のオープンソース実装	https://github.com/apache/openjpa.git
Camel	Java ベースのオープンソース統合フレームワーク	https://github.com/apache/camel
HBase	列指向データベース管理システム	https://github.com/apache/hbase.git

表 4. 使用した不具合データ

Project	Bug-tracking System	Resloved	Resolution	Issue Type	status	# of bugs
Log4j	Bugzilla	≤ 2018-01-22	FIXED	–	RESOLVED, VERIFIED, CLOSED	305
OpenJPA		≤ 2018-01-17				1162
Camel	JIRA	≤ 2018-01-22	–	bug	closed	1457
HBase		≤ 2018-01-29				5466

表 5. 得られたダンプデータ

Project	# of Total Data	Buggy rate	Date dumped
Log4j	3275	45.6%	2018/01/17
OpenJPA	4864	11.1%	2018/01/10
Camel	30739	20.7%	2018/01/10
HBase	14745	31.9%	2018/01/23

表 6. 推定結果に基づくコミット群 (1 手法)

Method	Classify as Buggy	Clean
SZZ	BUG_{szz}	$CLEAN_{szz}$
Commit Guru	BUG_{guru}	$CLEAN_{guru}$

表 7. 推定結果に基づくコミット群 (2 手法)

Method		Commit Guru	
	Classify as	Buggy	Clean
SZZ	Buggy		$DIFF_{szz}$
	Clean	$DIFF_{guru}$	$BOTH_{clean}$

表 8. SZZ アルゴリズムから得られた結果

Project	BUG tags (commits)	Buggy rate (Weighting)
Log4j	531 (296)	9.0% (16.2%)
OpenJPA	3156 (1301)	26.8% (64.9%)
Camel	4473 (2171)	7.1% (14.6%)
HBase	4257 (2408)	16.5% (28.9%)

る．

4 結果

SZZ アルゴリズムの実行結果を表 8 に示す．SZZ アルゴリズムは 1 つのコミットに対し複数の BUG タグを付けるため，Buggy rate として BUG タグが付けられたコミット数と，コミットについた BUG タグ数で重み付けしたコミット数の 2 つを算出した．

5 考察

5.1 RQ1: Commit Guru と SZZ アルゴリズムにおいて，それぞれの不具合混入コミット推定の結果にはどのような特徴が存在するか．

不具合コミットはなんらかの形で不具合を混入していないコミットとの間に差が生じていると考えられる．ゆえに，不具合コミット推定手法においても，推定された不具合コミットとそれ以外のコミットには差が生じていると思われる．ここでは，Commit Guru，および SZZ アルゴリズムの不具合コミットの推定結果，すなわち BUG_{szz}，BUG_{guru} において，それらを比較した際にどのような特徴が得られるかについて考察する．

各メトリクスの傾向 Commit Guru によって測定されたメトリクスは，表 2 において示した通り，そのコミットが不具合コミットとなるリスクの減少要因または増加要因と考えられている．ここでは，CF_{szz}，CF_{guru} において，Clean とされたコミットの中央値に対する Buggy なコミットの大小関係を各メトリクスごとにオッズを用いて評価し，各メトリクスがリスクの減少要因，増加要因のどちらとして働く傾向があるかを分析した．

その結果，オッズそのものの値に差はあるものの，ほとんどのメトリクスに対し両手法を通じて同様の働きが得られ，それは Buggy と推定されたコミットに対し期待されるものであった．

表 9. 外れ値検出の結果

project		LOF	method isolationForest	oneClassSVM
Log4j	# of commits	29	33	42
	BUG_{guru}	0.40%	0.70%	0.64%
	BUG_{szz}	0.18%	0.34%	0.24%
	$DIFF_{guru}$	0.21%	0.37%	0.40%
	$DIFF_{szz}$	0.00%	0.00%	0.00%
OpenJPA	# of commits	41	49	50
	BUG_{guru}	0.29%	0.56%	0.29%
	BUG_{szz}	0.35%	0.66%	0.43%
	$DIFF_{guru}$	0.062%	0.041%	0.021%
	$DIFF_{szz}$	0.12%	0.14%	0.16%
Camel	# of commits	271	308	309
	BUG_{guru}	0.09%	0.27%	0.28%
	BUG_{szz}	0.07%	0.16%	0.19%
	$DIFF_{guru}$	0.06%	0.14%	0.12%
	$DIFF_{szz}$	0.04%	0.03%	0.03%
HBase	# of commits	128	148	151
	BUG_{guru}	0.27%	0.75%	0.49%
	BUG_{szz}	0.11%	0.38%	0.27%
	$DIFF_{guru}$	0.20%	0.38%	0.24%
	$DIFF_{szz}$	0.03%	0.01%	0.02%

外れ値 先述した通り，不具合コミットはなんらかの形で不具合を混入していないコミットとの間に差が生じていると考えられるため，1 クラス SVM (One Class SVM), アイソレーションフォレスト (Isolation Forest), LOF (Local Outlier Factor) の 3 手法を用いて外れ値検出を行った．

各コミット群において，検出された外れ値をどの程度含んでいるか，その結果を表 9 に示す．全手法において，BUG_{guru} の方が BUG_{szz} よりも外れ値と判断されたコミットを含んでいた．

分類カテゴリ 不具合コミットの分類カテゴリの観点から，Commit Guru と SZZ アルゴリズムはどのような傾向があるかを分析した．結果を表 10 に示す．

その結果，BUG_{szz} は'Merge' に分類されたコミットを含まなかったことに対し，BUG_{guru} はわずかではあるが，'Merge' に分類されたコミットを含んでいた．また，BUG_{szz} と BUG_{guru} には' Corrective' に分類されたコミットに対して顕著な差が見られた．$DIFF_{szz}$ において Feature Addition に分類されたコミットの割合が高くなっているプロジェクトも存在するが，そもそもの不具合コミット推定の数量が違うため，ここにおいて比較するには疑念の余地が残る．

5.2 RQ2: Commit Guru と SZZ アルゴリズムにおいて，不具合混入コミット推定性能の差はどの程度か.

全コミットは，表 7 のように分類される．不具合コミット推定性能の差を調べるため，ここでは手法間で推定結果が異なるコミット $DIFF_{guru}$ および $DIFF_{szz}$ の比較を行った．また，それとともに Clean と推定されたコミット群にどれだけ差がないかを分析するため，$BOTH_{clean}$ に対して，$CLEAN_{guru}$, $CLEAN_{szz}$ の比較を行った．

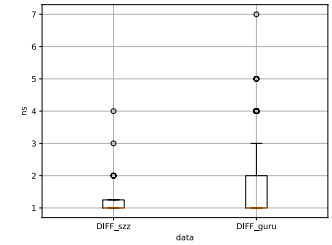
メトリクスの傾向 $DIFF_{guru}$，$DIFF_{szz}$ ともに，RQ1 より判断された Buggy と推定されたコミットに期待される傾向に対して，おおよそ沿った傾向を持っていた．また，Buggy と推定されたコミットに期待される傾向により近い傾向をもつコミット群は $DIFF_{szz}$ であった．

有意差 $BOTH_{clean}$ に対して，$CLEAN_{guru}$, $CLEAN_{szz}$ それぞれをマン・ホイットニーの U 検定を用いて比較した結果，Log4j, Camel, HBase は $CLEAN_{guru}$, OpenJPA は $CLEAN_{szz}$ の方がより多くのメトリクスにおいて有意差が存在しなかった．

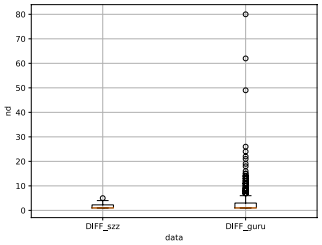
外れ値 全手法において，外れ値と推定されたコミットをより含んでいるのは Log4j, Camel, HBase においては $DIFF_{guru}$，OpenJPA においては $DIFF_{szz}$ であった．

また，箱ひげ図を用いて $DIFF_{szz}$，$DIFF_{guru}$ における各メトリクスごとの値の分布を評価したとき（図 1），$DIFF_{guru}$ の方が外れ値が明らかに多く，第一四分位点，第三四分位点の間が大きく開いていることがわかる．Buggy と推定されるコミットは，Clean と推定されるコミットと比較すると何らかの異常な特徴が現われると考えられる．そのため，比較的まとまりがある $DIFF_{szz}$ は $DIFF_{guru}$ よりも Buggy であるコミットが含まれている可能性は低いと考えられる．ここではページ数の関係より，コミット数が少ない Log4j のみを示したが，コミット数が比較的多い他のプロジェクトにおいても同様の結果が得られた．

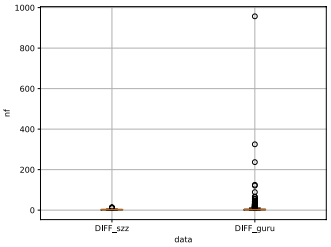
以上より，推定コミット数の多さにも関わらず，Buggy と推定されたコミットに期待される傾向におおよそ沿っており，外れ値をより含んでいる $DIFF_{szz}$ の方が不具合コミット推定結果において優れていると考えられる．また，Clean と推定されたコミット間にほとんどのプロジェクトにおいて有意差が存在しなかったことから，Clean と推定されたコミット群はある程度似通っていると考えられる．このため，その中に含んでいる Buggy と推定されるコミット数は少ないと考えられる．以上より，不具合コミットである可能性が高いものをより推定でき



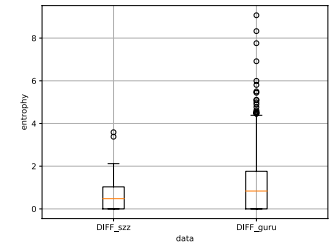
(a) ns



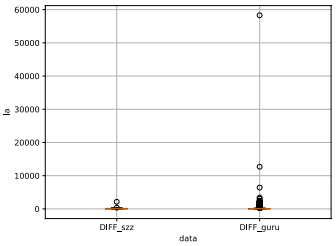
(b) nd



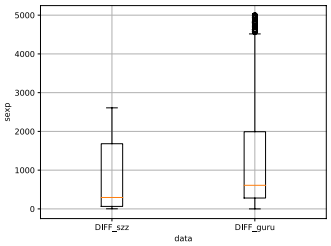
(c) nf



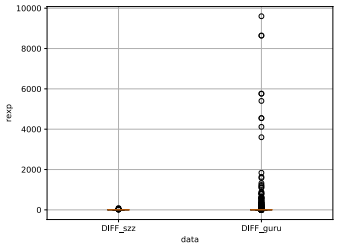
(d) entrophy



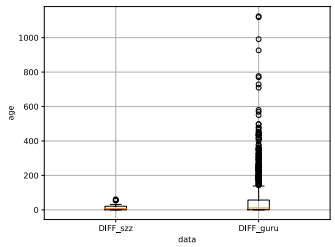
(e) la



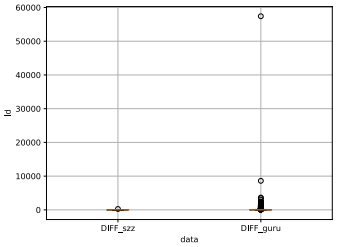
(f) sexp



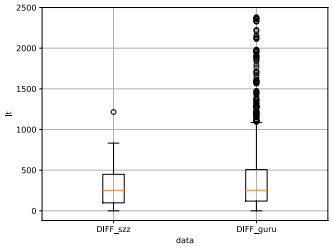
(g) rexp



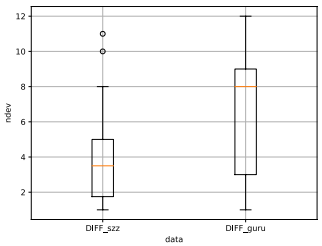
(h) age



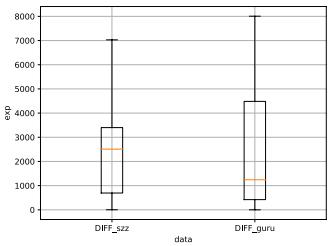
(i) ld



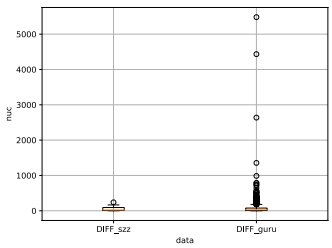
(j) lt



(k) ndev



(l) exp



(m) nuc

図 1. SZZ アルゴリズムと CommitGuru の間でバグ推定の結果が異なるものにおけるメトリクス値の分布の比較 (Log4j プロジェクト)

表 10. 各コミット群における分類カテゴリの割合

project	commits group	# of commits	None	Feature Addition	Corrective	Non Functional	Preventative	Perfective	Merge
Log4j	BUG_{guru}	1495	29.70%	23.08%	34.45%	6.76%	4.62%	1.40%	0.00%
	BUG_{szz}	297	28.62%	25.93%	30.64%	6.73%	7.41%	0.67%	0.00%
	$DIFF_{guru}$	1218	29.72%	22.82%	35.06%	6.90%	3.94%	1.56%	0.00%
	$DIFF_{szz}$	20	15.00%	50.00%	15.00%	15.00%	5.00%	0.00%	0.00%
OpneJPA	BUG_{guru}	540	58.89%	15.19%	14.63%	1.11%	7.96%	2.22%	0.00%
	BUG_{szz}	1301	62.18%	14.60%	11.84%	4.53%	5.30%	1.54%	0.00%
	$DIFF_{guru}$	216	50.93%	12.96%	23.15%	1.39%	10.65%	0.93%	0.00%
	$DIFF_{szz}$	977	61.51%	13.92%	12.79%	5.73%	5.02%	1.02%	0.00%
Camel	BUG_{guru}	6360	57.89%	19.47%	16.84%	1.18%	3.57%	1.02%	0.03%
	BUG_{szz}	2171	61.72%	20.68%	12.76%	1.01%	2.49%	1.34%	0.00%
	$DIFF_{guru}$	4970	57.32%	18.71%	17.85%	1.27%	3.86%	0.95%	0.04%
	$DIFF_{szz}$	781	64.92%	18.05%	11.91%	1.28%	2.43%	1.41%	0.00%
HBase	BUG_{guru}	4698	56.85%	15.11%	18.69%	1.60%	5.26%	2.38%	0.11%
	BUG_{szz}	2408	60.09%	15.49%	15.49%	2.12%	4.44%	2.37%	0.00%
	$DIFF_{guru}$	3072	54.33%	15.66%	20.21%	1.43%	5.86%	2.34%	0.16%
	$DIFF_{szz}$	782	56.91%	18.41%	14.83%	2.56%	5.12%	2.17%	0.00%

るのは Commit Guru であると考えられる．しかしながら，プロジェクトによって結果が非常に左右されており（OpenJPA プロジェクト），表 10 より Commit Guru は Merge と推定されるコミットを Buggy として推定している．このため，Commit Guru の情報を正解データとして利用するとしても，そのデータのみを十二分に信頼するのではなく，様々な手法を用いて検証を行った方が良いと考えられる．

6 結言

ソフトウェアの複雑さと重要性が日々増してきている昨今，ソフトウェアの品質予測は重要な研究テーマであり，その研究は活発化している．その中で，不具合の正解データとして用いられることの多い Commit Guru の不具合混入コミットの情報に対する信頼性は不確かなものであった．そこで，本研究では Commit Guru の不具合混入コミットの情報における正解データとしての信頼性検証のため，不具合混入コミット推定手法である SZZ アルゴリズムとの結果の整合性を比較し，考察を行なった．その結果，Commit Guru の不具合混入コミットの情報は不具合の正解データとして可能性を示すものの，その信頼性は十分であるとは言い難い．

参考文献

[1] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” Technical report, National Institute of Standards and Technology, 2002.

[2] E. Shihub, “An exploration of challenges limiting pragmatic software defect prediction,” PhD thesis, Queen’s University, 2012.

[3] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” Proceedings of the 2005 International Workshop on Mining Software Repositories, pp.1–5, Proc. of 2nd International workshop on Mining software repositories, ACM, New York, NY, USA, 2005.

[4] C. Rosen, B. Grawi, and E. Shihab, “Commit guru: Analytics and risk prediction of software commits,” Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp.966–969, ESEC/FSE 2015, ACM, New York, NY, USA, 2015.

[5] A. Mockus and L.G. Votta, “Identifying reasons for software changes using historic databases,” Proceedings of the International Conference on Software Maintenance (ICSM’00), pp.120–130, ICSM ’00, IEEE Computer Society, Washington, DC, USA, 2000.

[6] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” IEEE Trans. Softw. Eng., vol.39, no.6, pp.757–773, June 2013.

[7] A. Hindle, D.M. German, and R. Holt, “What do large commits tell us?: A taxonomical study of large commits,” Proceedings of the 2008 International Working Conference on Mining Software Repositories, pp.99–108, Proc. of 5th International workshop on Mining software repositories, ACM, New York, NY, USA, 2008.

[8] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” Proceedings of the 30th International Conference on Software Engineering, pp.181–190, Proc. of 30th International Conference on Software Engineering, ACM, New York, NY, USA, 2008.

[9] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” Proceedings of the 27th International Conference on Software Engineering, pp.284–292, ACM, New York, NY, USA, 2005.

[10] A.G. Koru, D. Zhang, K.E. Emam, and H. Liu, “An investigation into the functional form of the size-defect relationship for software modules,” IEEE Transactions on Software Engineering, vol.35, pp.293–304, 2009.

[11] A. Mockus and D.M. Weiss, “Predicting risk of software changes,” Bell Labs Technical Journal, vol.5, no.2, pp.169–180, aug 2002.

[12] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” Proceedings of the 28th International Conference on Software Engineering, pp.452–461, Proc. of 28th International Conference on Software Engineering, ACM, New York, NY, USA, 2006.

[13] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, “An analysis of developer metrics for fault prediction,” Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp.18:1–18:9, PROMISE ’10, ACM, New York, NY, USA, 2010.

[14] T.L. Graves, A.F. Karr, J.S. Marron, and H.P. Siy, “Predicting fault incidence using software change history,” IEEE Trans. Software Eng., vol.26, pp.653–661, 2000.

[15] A.E. Hassan, “Predicting faults using the complexity of code changes,” Proceedings of the 31st International Conference on Software Engineering, pp.78–88, Proc. of 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2009.

[16] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp.31–41, 2010.

[17] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows,” Proceedings of the 32th International Conference on Software Engineering (ICSE), pp.495–504, ACM, May 2010.

[18] R. Purushothaman and D.E. Perry, “Toward understanding the rhetoric of small source code changes,” IEEE Transactions on Software Engineering, vol.31, pp.511–526, 2005.

不具合誘発パラメータ組み合わせ特定三手法の比較評価

渡辺　大輝
京都工芸繊維大学 大学院工芸科学研究科
博士前期課程 情報工学専攻
d-watanabe@se.is.kit.ac.jp

西浦　生成
京都工芸繊維大学 大学院工芸科学研究科
博士後期課程 設計工学専攻
k-nishiura@se.is.kit.ac.jp

水野 修
京都工芸繊維大学 情報工学・人間科学系
o-mizuno@kit.ac.jp

要旨

組み合わせテストによる不具合誘発パラメータ組み合わせの特定は，ソフトウェア開発者が不具合誘発の原因となる要因を特定する上で重要な役割を果たす．近年，様々な研究者によって組み合わせテストの手法が数多く提案されている．一方で，不具合の個数や不具合誘発条件の複雑さ，用いるシステムの規模などで示されるある特定の状況下において，実際どの手法を用いれば最も効率よく正確に不具合誘発パラメータ組み合わせを特定できるのかという疑問が抱かれる．本論文では，これまでに提案された 3 種類の従来手法を用いて，組み合わせテストにかかる処理時間，必要な追加テストケースとその実行回数，不具合特定成功率といった 3 つの観点を中心に比較評価を行った．実験の結果，用いたテストスイートの変化による同一手法内でのデータの変化や，同一のテストスイートにおける 3 種類の従来手法の実験結果の差異について収集することが出来た．また，得られたデータを元に比較を行い，3 種類の従来手法の有用性の差別化や，テストスイートの変化が引き起こす影響についての結論を示した．

1　はじめに

ソフトウェアの開発は，必ずしも最初から最後まで思い通りに進むとは限らない．開発を行っている最中，または世の中に出回った後に不具合が見つかる可能性も大いに存在する．この不具合が現代社会に及ぼす影響は大きいものであるがゆえに，その不具合を検出するテスト手法の一つである組み合わせテストは重要な役割を果たす．

ソフトウェアにおける不具合は，ある単一のものから誘発されるとは限らず，ある条件の組み合わせが存在することによって不具合が誘発されるということが起こりうる．ソフトウェアにおいて不具合が誘発されることが発覚したとき，我々はどの組み合わせが不具合を誘発させているのかどうかを特定する必要がある．近年，多くの研究者がパラメータ集合を元に，それらを全て正確に導き出し，不具合誘発パラメータ組み合わせとして出力する組み合わせテストの手法を様々に提案している．その手法は，元のテストスイートに大量にテストケースを追加していき不具合誘発パラメータ組み合わせを特定する手法から，たった一つのテストケースからそのパラメータを変更していき不具合誘発パラメータ組み合わせを特定する手法まで様々である．一般に，組み合わせテストによって不具合誘発パラメータ組み合わせを特定することを FIL（Fault Interaction Location）と呼ぶ．では，実際に FIL を行うとなったとき，一体どの手法を選択すればいいのだろうか．例えばあるテストスイート下において 100%正しく FIL を正確に行うことが出来る手法があったとしても，テストの実行にかかる時間が長ければ使用者のストレスの増加や作業の非効率化が予測される．逆にどんなに速やかに FIL を行うことが出来る手法があったとしても，その出力結果の正確さが欠けている

と不具合修正の容易さに悪影響を及ぼしてしまう．ゆえに，対象とするテストスイートの性質に対して最適な手法を選択することが出来れば，組み合わせテストの効率化を図ることが出来ることが期待される．

そこで本論文では，次の 3 つの従来手法の実装を行った: 2016 年に Zheng らによって提案された comFIL（complete Fault Interaction Location）[1]，2011 年に Zhang らによって提案された FIC（Faulty Interaction Characterization）[2]，2012 年に Ghandehari らによって提案された identifying inducing combinations [3]．また，20 種類のテストスイートを用意し，それぞれを処理時間，成功率，テストの実行回数の 3 つの観点から比較を行い，3 手法の有用性を調査した．

2 準備

2.1 組み合わせテスト

組み合わせテストは，パラメータが取る値の組み合わせに注目し，ある組み合わせ数のパラメータ間で取る値のパターンの全てを網羅するテストのことであり，組み合わせ数を t としたとき，t-way テストや t-wise テストと呼ぶ．以後，あるパラメータが取る値のことをパラメータ値と表記する．例えば，表 1 のパラメータ仕様が与えられたときの 2-way テストを生成する．まず総テストケースなら，3 パターンのパラメータ値を取るパラメータが 1 つ，2 パターンのパラメータ値を取るパラメータが 2 つなので，テストケース数は $3\times2\times2=12$ となる．対して 2-way テストの組み合わせテストなら，表 2 のテストスイートになり，テストケース数は 6 となる．考えられる全ての 2 つの組み合わせ (X,Y)，(Y,Z)，(Z,X) のパラメータ値を確認すると，確かに全てパターンを網羅している．組み合わせテスト生成ツールには，Microsoft 社の Czerwonka ら [4] が開発した PICT[1]，産業技術総合研究所の Choi ら [5] が開発した pricot，Bryce ら [6] が開発した DDA などがある．

2.2 FI(Fault Interaction)

FI (Fault Interaction) とは，テストケースにおける，不具合を誘発させる原因となる組み合わせのことである．以後，FI で統一する．FI は単一のパラメータによって構成されることもあれば，複数パラメータ (組み合わせ) によって構成されることもある．この論文では，FI を次のように表記する．

表 1. パラメータの仕様

parameter	value
X	1,2,3
Y	1,2
Z	1,2

表 2. 表 1 の 2-way 組み合わせテスト

テストケース	X	Y	Z
t_1	1	1	1
t_2	1	2	2
t_3	2	1	2
t_4	2	2	1
t_5	3	1	1
t_6	3	2	2

- FI が単一のパラメータであるとき，(変数名=パラメータ値)
- FI が複数のパラメータであるとき，(変数名=パラメータ値, 変数名=パラメータ値, ...)

またこれ以後，FI を含んでいるときの実行結果を Fail，FI を含んでいないときの実行結果を Pass と表記する．更に，この論文では「誘発」という言葉を「FI を含んでいることが原因となってテストの実行結果が Fail となること」という意味で定義している．

2.3 従来手法

2.3.1 comFIL(complete Fault Interaction Location)

comFIL(complete Fault Interaction Location) [1] は，2016 年に Zheng らによって提案された組み合わせテストの一種である．以後，comFIL で統一する．

この手法では，まずテストスイートを入力する．例えば，2 パターンのパラメータ値を取るパラメータが 2 つ，3 パターンのパラメータ値を取るパラメータが 1 つ，4 パターンのパラメータ値を取るパラメータが 1 つの 2-way テストの入力は表 3 のテストスイートになる．同時に，そのテストケースが FI を含んでいるかどうかを一つずつ判定していく．表 3 の実行結果は，FI が $(a=0,c=0)$，$(a=0,d=3)$ であったときを示している．

[1] https://github.com/Microsoft/pict

表 3. 2-way テストの実行結果

テストケース	a	b	c	d	実行結果
1	0	0	0	0	Fail
2	1	1	1	0	Pass
3	0	1	2	0	Pass
4	1	0	0	1	Pass
5	0	0	1	1	Pass
6	1	1	2	1	Pass
7	0	1	0	2	Fail
8	1	0	1	2	Pass
9	0	0	2	2	Pass
10	0	1	0	3	Fail
11	1	0	1	3	Pass
12	1	0	2	3	Pass

次に，Fail したテストケースの部分集合を考える．その中から，Pass した全てのテストケースの部分集合でないものが，FI の候補となる．例えば，$(a = 0, b = 0)$ は Pass したテストケ ースの部分集合になっているので FI の候補とはならないが，$(a = 0, d = 0)$ は Pass したテストケースの部分集合でないので FI の候補となる．この場合，全ての FI の候補は表 4 の通りになる．

次に，FI の候補を元にテストケースを追加していく．追加テストケースの条件は以下の通りである．

- 最初に入力したテストスイートとは被らないようにする．
- 追加テストケースに真の FI が追加されることはないものとする．例えば，表 4 の T_{11} には $a = 0$ は追加されない．
- 複雑化を防ぐため，追加テストケースを作成したときに新たな FI は生まれないものとする．

FI の候補を 1 つずつ取り出し，追加テストケースを作成したあとに実行する．Fail だった場合，その FI 候補を部分集合に持つ他の FI 候補を FI 候補から取り除く．例えば，表 4 の T_1 は実行後 Fail となり，T_1 を部分集合に持つ T_8 が FI 候補から取り除かれる．Pass だった場合，その FI 候補自身とその FI 候補自身の部分集合を FI 候補から取り除く．例えば，表 4 の T_2 は実行後 Pass となり，T_2 自身と T_2 の部分集合である T_6 が FI 候補から取り除かれる．なお，この T_6 や T_8 のように追加テストケースを作成する前に取り除かれる FI 候補には追加テストケースを作成しない．これを順番に繰り返し，最終的に取り除かれることなく残った FI 候補が真の FI として特定される．

表 4. 表 3 の FI 候補

FI 候補	a	b	c	d
t_1	0	0	0	-
t_2	0	0	-	0
t_3	0	-	0	0
t_4	-	0	0	0
t_5	0	-	0	-
t_6	-	0	-	0
t_7	-	-	0	0
t_8	0	0	0	0
t_9	0	1	0	2
t_{10}	0	1	0	-
t_{11}	-	1	0	-
t_{12}	0	-	0	2
t_{13}	-	-	0	2
t_{14}	-	1	0	2
t_{15}	-	1	-	2
t_{16}	0	1	-	2
t_{17}	0	1	0	3
t_{18}	0	-	0	3
t_{19}	-	-	0	3
t_{20}	0	-	-	3
t_{21}	-	1	-	3
t_{22}	-	1	0	3
t_{23}	0	1	-	3

2.3.2 FIC (Faulty Interaction Characterization)

FIC(Faulty Interaction Characterization) [2] は，2011 年に Zhang らによって提案された組み合わせテストの一種である．以後，FIC で統一する．

この手法では，最初に入力するテストケースはたった 1 つであり，テストケースのパラメータ値そのものを変更していくことによってその中に含まれる FI を特定する．他に，テストケースの要素数と，パラメータ値のパターン数を入力する．今回の例では，入力したテストケースを $[1, 2, 2, 1, 2, 2, 1, 1]$(このときテストケースの要素数は 8)，パラメータ値のパターン数は 8 個とも 2 パターンとする．また，FI は $(c = 2, f = 2)$，$(d = 1)$ とする．

次に，FI を特定するためにテストケースのパラメータ値を順番に変更していき，1 回ずつ実行していく．その実行結果が Fail のときは，変更したパラメータ値はそのままに次の要素のパラメータ値を変更していく．Pass のときはパラメータ値を変更する前に戻し，次の要素のパラメータ値を変更していく．今回の例での推移を表 5 に

表 5. FIC の推移例

テストケース								実行結果
a	b	c	d	e	f	g	h	
1	2	2	1	2	2	1	1	Fail(Skip)
2	2	2	1	2	2	1	1	Fail
2	1	2	1	2	2	1	1	Fail
2	1	1	1	2	2	1	1	Fail
2	1	1	2	2	2	1	1	Pass
2	1	1	1	1	2	1	1	Fail
2	1	1	1	1	1	1	1	Fail
2	1	1	1	1	1	2	1	Fail
2	1	1	1	1	1	2	2	Fail
1	2	2	1	2	2	1	1	Fail
2	2	2	2	2	2	1	1	Fail
2	1	2	2	2	2	1	1	Fail
2	1	1	2	2	2	1	1	Pass
2	1	2	2	1	2	1	1	Fail
2	1	2	2	1	1	1	1	Pass
2	1	2	2	1	2	2	1	Fail
2	1	2	2	1	2	2	2	Fail
1	2	1	2	2	1	1	1	Pass

示す．まず最初に a のパラメータ値を 2 に変更し，その後再度実行する．実行結果は Fail のため，a のパラメータ値は 2 のままに，次は b のパラメータ値を 1 に変更する．これを繰り返していくと，d を 2 に変更したときに初めて実行結果が Pass になる．よって，d のパラメータ値は 1 に戻され，再び e からパラメータ値を変更していく．その後は最後まで Fail であり続けるため，これにより一つ目の FI が $(d = 1)$ と特定される．

1 週目が終了した後，元のテストケースより FI である $(d = 1)$ を取り除くため，d のパラメータ値を 2 に変更し，テストケースを $[1, 2, 2, 2, 2, 2, 1, 1]$ とする．その後再度実行すると結果は Fail となるため，このテストケースにはまだ FI が存在しているということになる．よって，再びこのテストケースのパラメータ値を順番に変更していく．しかし，d のパラメータ値を 1 に変更してしまうと FI である $(d = 1)$ を含んでしまうため，d のパラメータ値の変更は行わない．すると表 5 に示された通り，c と f のパラメータ値を変更したときに実行結果が Pass になる．よって二つ目の FI が $(c = 2, f = 2)$ と特定される．

2 週目が終了した後，今度は $(c = 2, f = 2)$ を取り除くために c のパラメータ値を 1 に，f のパラメータ値を 1 にそれぞれ変更し，テストケースを $[1, 2, 1, 2, 2, 1, 1, 1]$ とする．その後再度実行すると結果は Pass となるため，このテストケースには FI が存在していないことになる．以上から，真の FI は $(c = 2, f = 2)$，$(d = 1)$ と特定される．

表 6. 表 3 の FI 候補 (IIC)

FI 候補	a	b	c	d
t_1	0	-	0	-
t_2	-	1	0	-
t_3	-	-	0	0
t_4	-	-	0	3
t_5	-	-	0	2
t_6	0	-	-	3
t_7	-	1	-	3
t_8	-	1	-	2
t_9	-	0	-	0

2.3.3 Identifying Inducing Combinations

Identifying Inducing Combinations [3] は，2012 年に Ghandehari らによって提案された組み合わせテストの一種である．以後，IIC で統一する．

この手法は，FI 候補となる組み合わせを，ある計算式によって真の FI である可能性をランク付けし，その可能性の高いものから順に追加テストケースを作成していくというものである．

まずテストスイートを入力する．今回は 2.3.1 節で使用したテストスイートと同じものを使用するものとし，FI も 2.3.1 節と同様に $(a = 0, c = 0)$，$(a = 0, d = 3)$ とする．このとき，テストスイートとそれぞれの実行結果は表 3 に示される．

次に，comFIL と同様に，Fail したテストケースの部分集合の中から，Pass した全てのテストケースの部分集合でないものを，FI の候補とする．しかし，comFIL は部分集合のパラメータ数に決まりがなかったのに対し，IIC では部分集合のパラメータ数は t-way テストの t の値のものに限る．つまり，今回の例ではパラメータ数が 2 である部分集合を考える．この場合，全ての FI の候補は表 6 の通りになる．

次に，FI 候補の構成パラメータ一つ一つを，後述の式 1 によって，その疑わしさを値として算出する．

$$\rho(o) = \frac{1}{3}(u(o) + v(o) + w(o)) \quad (1)$$

ここで，$u(o)$，$v(o)$，$w(o)$ は以下の式で算出される．

$$u(o) = \frac{|\{f \in F_i | r(f) = fail \wedge o \in f\}|}{|\{f \in F_i | r(f) = fail\}|} \quad (2)$$

$$v(o) = \frac{|\{f \in F_i | r(f) = fail \wedge o \in f\}|}{|\{f \in F_i | o \in f\}|} \quad (3)$$

$$\rho(o) = \frac{|\{c | o \in c \wedge c \in \pi\}|}{|\pi|} \quad (4)$$

$u(o)$ は，Fail したテストケースに含まれている総数の値から，Fail したテストケースの総数の値を除算する．$v(o)$ は，Fail したテストケースに含まれている総数の値から，全てのテストケースに含まれている総数の値を除算する．$w(o)$ は，FI 候補に含まっている総数の値から，FI 候補自体の総数の値を除算する．例えば，$\rho\ (c \to 0)$ は式 5 の通りに算出される．

$$\rho(c \leftarrow 0) = \frac{1}{3} * (1 + \frac{3}{4} + \frac{5}{9}) = 0.7685 \quad (5)$$

同様に，全ての FI 候補の構成パラメータについて式 1 の値を算出する．その結果を表 6 に示す．

次に，真の FI である可能性をランク付ける 2 つの値を算出する．それらは式 6 と式 7 で与えられる．

$$\rho_c(c) = \frac{1}{|c|} \sum_{\forall o \in c} \rho(o) \quad (6)$$

$$\rho_e(c) = Min(\sum_{o \in f \wedge o \notin c} \rho(o), \forall f \in F) \quad (7)$$

ρ_c は，ある FI 候補の構成パラメータ 1 つ 1 つの $\rho(o)$ の平均値を取る．例えば，$\rho_c(a \to 0, c \to 0)$ は，$\rho(a \to 0)$ と $\rho(c \to 0)$ の値の平均値を取る．また，ρ_e は，ある FI 候補の構成パラメータ以外で構成されている ρ_c の中の最小値を取る．例えば，$\rho_e(a \to 0, c \to 0)$ は $\rho_c(b \to 0, d \to 0)$，$\rho_c(b \to 1, d \to 2)$，$\rho_c(b \to 1, d \to 3)$ の中から最小の値を取る．これらを全ての FI 候補について算出し，ρ_c はその値の高い順に，ρ_e はその値の低い順にそれぞれランク付けを行う．以下，その結果をそれぞれ R_c，R_e と呼ぶこととする．

次に，R_c と R_e の値を加算した値を算出し，その値の低い順にそれぞれランク付けを行う．これを R と呼ぶこととする．この R の値の低い FI 候補が，真の FI である可能性が高いものとして疑われることになる．以上をまとめた結果を表 7 に示す．

表 7. IIC のランク付け

FI 候補	ρ_c	R_c	ρ_e	R_e	$R_c + R_e$	R
t_1	0.6713	1	0.2460	1	2	1
t_2	0.6176	2	0.4352	3	5	2
t_3	0.5324	4	0.3849	2	6	3
t_4	0.5509	3	0.5204	4	7	4
t_5	0.5324	4	0.5204	4	8	5
t_6	0.4537	5	0.6176	5	10	6
t_7	0.4000	6	0.6713	6	12	7
t_8	0.3815	7	0.6713	6	13	8
t_9	0.2460	8	0.6713	6	14	9

次に，R の値の低い FI 候補から順に追加テストケースを作成し，実行を行う．実行自体は全ての FI 候補に対して行う．追加テストケースの作成方法は以下の通りである．

- 最初に入力したテストスイートとは被らないようにする．
- 追加する部分のパラメータ値は，$\rho(o)$ の値が 1 番低いものを採用する．

これにより，真の FI を特定する．

3 実験

本節では，研究設問，実験環境の詳細な明記，テストスイートなどの実験対象の説明，実験手順や留意点，得られた実験結果について説明する．

3.1 研究設問

本実験を行うにあたって，以下の研究設問を設定した．

- RQ1:同一の手法のなかで，パラメータ数の大小とパラメータ値の種類数が与える影響は何か．
- RQ2:異なる手法のなかで，それぞれのテストスイートを用いるときの他の手法との差は何か．

これらを回答するために，まず 1 つの従来手法において様々なテストスイートとシステムモデルを用いて実行し，それと全く条件で他の 2 つの従来手法においても実行すればいいことが分かる．

今回はそれぞれのテストスイートにおいて，以下の 3 つの観点について測定を行い，これらから 3 種類の従来手法の有用性について差別化を行う．

- 処理時間
- 成功率
- 追加テストの実行回数

3.2 実験準備

3.2.1 実験環境

本実験で用いたプログラミング言語，開発環境，PC は以下の通りである．

- PC:MacBook Pro(Retina 13-inch，Late 2013)，Sierra(バージョン 10.12.6)，2.8 GHz Intel Core i7，8 GB 1600 MHz DDR3
- 言語:Python 3.6.3

3.2.2 実験対象

用いるテストスイートのパラメータ数，パラメータ値のパターン数の 2 つの要素の設定をシステムモデルと表記する．本実験では，4 種類のシステムモデルに対してそれぞれ 5 通りの方法で作成された計 20 種類のテストスイートを実験対象として用いる．

- ランダムテスト (100 個)
- ランダムテスト (500 個)
- 2-way テスト
- 3-way テスト
- 4-way テスト

ここで，ランダムテストとは，ランダムにパラメータの値を決定して作成された n 個のテストケースをテストスイートとしたものである．本実験では $n = 100$ の場合と $n = 500$ の場合を用いる．また，t-way テストの生成には Microsoft の組み合わせテスト生成ツールである PICT を用いた．t-way のそれぞれのテストケース数を表 8 に示す．

表 8. t-way テストのテストケース数

	2-way	3-way	4-way
システム A	13	43	127
システム B	24	111	455
システム C	19	69	233
システム D	31	163	795

表 9. 対象システムモデルの詳細

システム	パラメータ数	パラメータ値のパターン数
A	10	2,2,2,2,2,3,3,3,3,3
B	10	3,3,3,3,3,4,4,4,4,4
C	20	2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,3,3
D	20	3,3,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,4

組み合わせテストのテストスイート設計法の 1 つである直交表を利用したテストは，指定の大きさの組み合わせ因子を網羅しつつテストケース数をできるだけ少なくする点で t-way テストと同じであり，目的とする役割が t-way と同じであるため本実験では使用しなかった．

また，使用する対象システムモデルを表 9 に示す通りに設定する．このシステムは，本実験用に作成した架空のものである．例えば，システム A は 2 種類の値を取るパラメータを 5 個，3 種類の値を取るパラメータを 5 個，合計 10 個のパラメータを持つ．

3.2.3 実験手順

2 節で述べた通り，既存の FIL 手法である comFIL，FIC，IIC の 3 手法を用いて比較を行う．

まずプログラムを実行する前に，FI の数，FI の大きさ，FI のパラメータ値の設定を行う．FI の数は全てのテストにおいて 1 から 3 までの値でランダムに設定する．FI の大きさはランダムテストは 3，t-way テストは t で統一する．FI のパラメータ値は各対象システムモデルにおいて設定されているパラメータ値のパターン数の中からランダムに与える．ただし，少なくとも 1 つのテストケースは失敗するように選択する．

FI の設定後，それらに対して各手法を適用させて設定した FI を特定した．その後，処理時間，成功回数，追加テストケースの実行回数を測定した．これらの 1000 種類のテスト結果のサンプルを用意し，処理時間とテストの実行回数の平均値，最小値，第 1 四分位数，中央値，第 3 四分位数，最大値を算出した．成功率は百分率で値

を算出した．このテスト結果はランダムに設定された FI による作用である．

また，comFIL では本来その大きさに関わらず，Fail したテストケースに部分集合として含まれ，かつ Pass したテストケースには部分集合として含まれないパラメータの全ての組み合わせを FI 候補とするが，テストケースのパラメータ数が大きいほど組み合わせ数は膨大になり処理に大きな時間がかかると予想される．そこで本実験では実験と比較の簡易化のため，それぞれ対象とするテストスイートが網羅する組み合わせの大きさ以下の部分集合のみを考える．

また，FI の数は分かっていないことを前提とし，IIC においては全ての FI が特定出来たとしても追加テストケースを作成していない FI 候補が残っている場合は全て終わるまで実行を続けることとする．

3.3 実験結果

実験結果を表 10 に示す．ここで，ave(t)[s] は平均処理時間を，ave(n)[回] は追加テストの平均実行回数を表す．

成功率は，各従来手法の出力によって導き出された FI と，予めランダムで決定されて与えられた FI が完全に一致している場合を成功とし，その割合を算出する．テストケースの実行回数は初めにテストケースを Fail したものと Pass したものに分けるための実行は含めず，追加のテストケースを実行した回数を測定する．以下，実行回数と表記されているものも同様である．また，今回は 3 種類の従来手法全てにおいて，追加のテストケースを実行するためにかかる時間を 0.1 秒と仮定する．しかし，そのままソースコードに 0.1 秒待機する命令を加えてしまうと本実験に費やされる時間が膨大になることを予測し，本実験では効率化のため実行回数に 0.1 を乗算した値を測定された処理時間に加算し，その値を処理時間として算出した．

実験結果では，同じ従来手法でもテスト種類によって大きく異なる結果が得られた．また，それぞれの従来手法においても大きく異なる結果が得られた．

また，テストの成功率は全て 100%を記録していた．このことから，3 種類の手法は全て FI を正確に特定出来ていることが分かった．

4 考察

4.1 研究設問への回答

考察は表 10 の実験結果を元に行う．

RQ1：同一の手法のなかで，パラメータ数の大小とパラメータ値の種類数が与える影響は何か．

最初に，comFIL について考える．ランダムテストにおいては，4 種類全てのシステムにおいて，テストケースを多く用いる場合に処理時間と追加テストの実行回数が減少することが分かった．comFIL は実行前に全てのテストケースを Fail したテストケースと Pass したテストケースに分類し，その後 Fail したテストケースのみの部分集合になっているパラメータ値の組み合わせを FI 候補とするが，本実験では FI の個数は多くても 3 つであるため，テストケースの数が多くなると同時に Fail したテストケースと比べて Pass したテストケースの個数が相対的に増える幅が大きいことが予想される．これにより，FI 候補の数が減少することが必然的に予想され，その結果追加テストケースの個数が減少したことで処理時間と追加テストの実行回数が減少したと考えられる．

次に，t-way テストは処理時間についてはシステム B においてのみ，2-way テストが最も処理時間が大きい結果となった．それ以外のシステムにおいては t の値が大きくなるにつれて大きくなる傾向が見られた．また，追加テストの実行回数においては 4 種類全てのシステムにおいて t の値が大きくなるにつれて多くなる傾向が見られた．これは，組み合わせ数が大きい方が Pass したテストケースの部分集合になりにくくなり，FI 候補の数が増加するためであると考えられる．異なるシステム間においては，パラメータ数とパラメータ値のパターン数が多いほど処理時間と追加テストの実行回数が増加することが分かった．またシステム B と C を比較すると，パラメータ値のパターン数が与える影響よりもパラメータ数が与える影響の方が大きいことが分かった．これは，パラメータ値のパターン数が増加するよりもパラメータ数が増加する方が，考えなければならない部分集合の数が爆発的に多くなることからこのような結果になったと考えられる．

次に，FIC について考える．同一のシステム内においては 5 種類全てのテストにおいてほぼ全く同じ結果が得られた．また，パラメータ数が同一のシステム A，B とシステム C，D において，処理時間と追加テストの実行

表 10. 実験結果 (3 手法のシステム毎の処理時間と追加テストの実行回数の平均値)

システム	手法	テスト	ave(t)	ave(n)	手法	テスト	ave(t)	ave(n)	手法	テスト	ave(t)	ave(n)
		ran100	2.797	27.668		ran100	2.241	22.396		ran100	1.391	13.844
		ran500	1.723	15.869		ran500	2.228	22.242		ran500	0.274	2.062
A	comFIL	2-way	5.769	57.565	FIC	2-way	2.174	21.736	IIC	2-way	4.139	41.321
		3-way	7.710	76.839		3-way	2.204	22.033		3-way	4.094	40.828
		4-way	8.007	79.410		4-way	2.207	22.055		4-way	4.238	42.253
		ran100	8.878	88.428		ran100	2.089	20.867		ran100	4.267	42.520
		ran500	2.580	24.761		ran500	2.167	21.637		ran500	0.254	2.323
B	comFIL	2-way	11.310	56.503	FIC	2-way	2.235	22.341	IIC	2-way	3.663	36.574
		3-way	7.430	73.971		3-way	2.176	21.747		3-way	4.174	41.632
		4-way	8.285	81.512		4-way	2.238	22.341		4-way	4.385	43.698
		ran100	13.758	135.257		ran100	4.192	41.895		ran100	4.301	42.517
		ran500	4.203	31.051		ran500	4.148	41.412		ran500	0.338	2.039
C	comFIL	2-way	12.505	124.434	FIC	2-way	4.122	41.202	IIC	2-way	4.702	46.752
		3-way	12.742	195.257		3-way	4.086	40.845		3-way	4.490	44.143
		4-way	25.705	239.703		4-way	4.052	40.488		4-way	4.380	42.532
		ran100	67.331	664.086		ran100	3.904	39.018		ran100	4.324	35.938
		ran500	5.955	52.162		ran500	4.271	42.651		ran500	0.210	1.280
D	comFIL	2-way	15.776	157.012	FIC	2-way	4.214	42.126	IIC	2-way	4.887	48.501
		3-way	22.856	225.418		3-way	4.140	41.370		3-way	4.494	44.087
		4-way	31.988	275.066		4-way	4.258	42.483		4-way	4.202	40.510

回数の値も同様な結果が得られた．FIC はテストケースのパラメータ値を 1 つずつ順番に変えて実行をしていき，FI を 1 つ特定する手法なので，追加テストの実行回数はパラメータ数の値と FI の個数に完全に依存するものである．このことから，得られる結果の固定化を招いたと考えられる．また，システム A，B と比べてパラメータ数が 2 倍であるシステム C，D は処理時間と追加テストの実行回数もほぼ 2 倍になっていることから，処理時間と追加テストの実行回数はパラメータ数と比例していると考えられる．

最後に，IIC について考察する．comFIL と同じく，ランダムテストを対象とした場合には，4 種類全てのシステムにおいて，テストケースを多く用いる場合に処理時間と実行回数が減少することが分かった．これは，実験前に FI 候補を割り出す手順が comFIL と同じであるため，同様の理由でこのような結果が得られたと考えられる．t-way テストにおいては，comFIL と比べると t の値によって明白な差が生まれることはなかった．しかし，この論文の表中には示していないが，2-way テストでは処理時間と追加テストの実行回数のデータに大きなばらつきが見られた．特に，データの最大値が 3-way テストと 4-way テストに比べて高くなっていることが分かった．2-way テストは他のテストと比べてテストスイートの総テストケース数が少ないので，FI の個数が多いと Pass したテストケースの割合が他のテストに比べて減少する傾向にあると考えられる．これにより，FI 候補の数が多くなる可能性があることが予測される．IIC は comFIL と違い，最終的に FI 候補は全て追加テストケースを作成し実行する．よって，FI 候補の増加が必然的に処理時間と追加テストの実行回数に直結することになる．これが最大値を突出させた原因であると考えられる．異なるシステム間においても，処理時間と追加テストの実行回数に明白な差は生まれなかった．これにより，パラメータ数とパラメータ値のパターン数は生成される FI 候補の数には依存しないことが分かった．comFIL は本実験では FI の大きさの値以下の要素数を持つパラメータ組み合わせを全て考えている．対して，IIC は FI の大きさの値と同等の要素数を持つパラメータ組み合わせのみを考える．このことから，comFIL と比べて考える部分集合の数の伸びが少なくなることからこのような結果が得られたと考えられる．

RQ2：異なる手法のなかで，それぞれのテストスイートを用いるときの他の手法との差は何か．

本実験の結果から，ほぼ全てのテストスイートを対象とした場合において，処理時間と追加テストの実行回数が最も少なくなるのは FIC であることが分かった．先述

の通り，FIC の処理時間と追加テストの実行回数は追加で行うテストのテストケースのパラメータ数に依存し，comFIL と IIC は FI 候補の数に依存する．仮にパラメータ数が 1 増加したとすると，FIC の実行回数は 1×（FI の個数）回しか増加しない．これに対し，comFIL と IIC はパラメータ数が 1 増加するだけでも考えなければならない部分集合の数が膨大に増加するため，追加テストの実行回数も大きく増加する．このことから，多くの場合においてはテストケースをどんどん追加していく手法を用いるより，1 つのテストケースを用いて FIL を行う手法を用いる方が効率が良いことが分かった．しかし，テストケースが多量のランダムテストにおいてはその限りではないことが分かった．

次に，comFIL と IIC を比較し，その差別化について考察する．本実験の結果では，一般的に全てのテストにおいて，IIC の方が処理時間と追加テストの実行回数が少なかった．これは先述の通り，IIC が FI の大きさの値と同等の要素数を持つパラメータ組み合わせのみを考えるのに対し，comFIL は FI の大きさの値以下の要素数を持つパラメータ組み合わせを全て考えているので，考えなければならない部分集合の数が IIC の方が少なくなることからこのような結果となった．しかし，これは今回のように FI の大きさが事前に見当がつく場合に限り，現実のソフトウェア開発においては必ずしも FI の大きさが事前に見当がつくとは限らない．その場合においては，IIC とは違い全ての部分集合を考えることの出来る comFIL を用いる方が汎用的に優れていると考えられる．以上のことから，本実験から得られた 3 種類の従来手法の差別化は以下の通りに考えられる．

- IIC: FI の大きさに見当がつき，多量のランダムテストケースを扱うとき．
- comFIL: FI の大きさに見当がつかず，多量のランダムテストケースを扱うとき．
- FIC: t-way テストや少量のランダムテストケースを扱うとき．

また，成功率については 3 種類全ての手法で 20 種類全てのテストスイートにおいて 100%となった．このことから，得られる出力結果の精度についてはどの従来手法も同等であると考えられる．

4.2 処理時間と追加テストの実行回数

本実験の結果から，処理時間と追加テストの実行回数は正の相関関係があることが分かった．従って，追加テストの実行回数の削減が処理時間の削減に直結することが分かる．本実験では追加テストの実行時間を 0.1 秒と仮定しているが，もしこの実行時間が更に長くなったとき，テストにかかる総時間は追加テストの実行回数が多ければ多いほど大幅に長くなることが予測される．このような状況下で本実験で用いた 3 種類の従来手法から 1 つを選ぶときは，次の 2 つの判断基準が考えられる．

- FI の大きさに見当がつく場合，IIC を用いてランク付けを行い，順番に実行していく．FI を全て特定をし終えたら実行をやめる．そうすると残りを実行しなくて済むので，追加テストの実行回数の削減に繋がる．
- 本実験のほぼ全てのテストにおいて最も優れており，かつ処理時間がある程度予測できる FIC を用いる．

comFIL はどうしても追加テストの実行回数が多くなってしまうので，実行時間が長くなったときには不向きであると考えられる．

4.3 妥当性の検証

4.3.1 ランダム性

今回の実験では，FI の数，FI のパラメータ値を 1 回のループごとにランダムに与えている．FI の数が大きくなればなるほど，どの手法でも処理時間は長くなるので，与えられた FI の数に偏りが生じれば，テスト結果に少なからず影響が及ぶと考えられる．また，ランダムテストに用いたテストスイートにも偏りが生じている可能性も考えられる．総テストケースは 1 番少ないシステム A でも $2^5 \times 3^5 = 7776$ 通り存在する．1 番多いシステム D だとおよそ 619 億通りにも及ぶ．ランダムテストに用いたテストケース数の 100 や 500 という数はこれに比べるとかなり小さい値なので，テストスイートに偏りが生じる可能性は十分に存在する．

4.3.2 アルゴリズムの正当性

3 種類の従来手法の各アルゴリズムの実装は，他言語で記載されているアルゴリズムを参考に，Python 3.6.3

で書き換えた．論文に記載されているアルゴリズムは大まかに記載されており部分が少ないため，我々が実装したソースコードと異なってしまっている可能性がある．その場合も，得られる実験結果に誤差が生じる可能性が存在する．

4.3.3 comFIL の簡易化

本実験では実験の簡易化のため，comFIL で作成する FI 候補の要素数に FI の大きさの値までという制限をかけた．本来であれば，comFIL は FI の大きさに寄らずに考えられうる部分集合全てを元に FI 候補を作成するため，この制限により実験結果に誤差が生じる可能性が存在する．

4.3.4 出現する FI の大きさ

本実験では出現する FI の大きさをランダムテストでは 3，t-way テストでは t と固定した値を用いている．しかし，例えば 2-way テストにおいて FI の大きさが 3 以上のものが出現した場合，これは必ずしも特定できるとは限らない (テストスイートにその FI が含まっていない可能性があるため)．これはテストの成功率に影響を及ぼすと考えられる．

4.4 今後の課題

本実験における今後の課題として，与える FI の大きさなどといったランダム性の強い部分を出来るだけ平等になるような実装をすることが挙げられる．また，本実験では比較する要因として処理時間，成功率，テストの実行回数の 3 つを測定したが，この他にも従来手法の有用性を差別化できる要因を多く探し出すことが挙げられる．

5 まとめ

本論文では，これまでに発表された 3 種類の組み合わせテストによる不具合誘発パラメータ特定の従来手法について，処理時間，成功率，テストの実行回数の結果からの比較を行い，それぞれの有用性について差別化を行った．それぞれの従来手法のアルゴリズムの実装を行い，計 20 種類のテストスイートを用いて測定し，この 3 つの観点に関するデータを収集した．得られた結果から，それぞれの従来手法がどのような状況下において有用性が生まれるかについての結論の考察を行った．

考察の結果，comFIL は FI の大きさに見当がつかず多量のランダムテストケースを扱うとき，FIC は t-way テストや少量のランダムテストケースを扱うとき，IIC は FI の大きさに見当がついており多量のランダムテストケースを扱うときに有用性が生まれることが分かった．

今後の課題として，より正しいデータを得るために，与える FI などといったランダム性のある要因を消去出来るような実装や，有用性をより詳細に示すために他の比較すべき観点の探索が挙げられる．

参考文献

[1] D.H. Wei Zheng, Xiaoxue Wu, and Q. Zhu, “Locating minimal fault interaction in combinatorial testing,” Advances in Software Engineering, vol.2016, pp.1-10, 2016.

[2] Z. Zhang and J. Zhang, “Characterizing failure-causing parameter interactions by adaptive testing,” Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp.331-341, ISSTA ’11, ACM, New York, NY, USA, 2011.

[3] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker, “Identifying Failure-Inducing Combinations in s Combinatorial Test Set,” Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), pp.370-379, 2012.

[4] J. Czerwonka, “Pairwise testing in the real world: Practical extensions to test case generators,” Microsoft Corporation Software Testing Technical Articles, 2008.

[5] E. Choi, T. Kitamura, C. Artho, A. Yamada, and Y. Oiwa. “Priority integration for weighted combinatorial testing,” Proc. of 39th Annual International Computer Software and Applications Conference, pp. 242-247, 2015.

[6] R. Bryce and C. Colbourn, “Prioritized intersction testing for pair-wise coverage with seeding and constraints,” Information and Software Technology, Vol. 48, No. 10, pp.960-970, 2006.

モデリングによる暗黙知分解とスキル補完への取り組み
～共感と共創をつくり，人材不足解消と多能工を促進～

三輪　東　　清田　和美
SCSK 株式会社
azuma.miwa@scsk.jp
kazumi.kiyota@scsk.jp

與儀　兼吾
SCSK ニアショアシステムズ株式会社
kengo_yogi@scsk-nearshore.co.jp

日下部　茂
長崎県立大学
kusakabe@sun.ac.jp

要旨

大規模システムを広範囲に担当している現場では，複数の異なる作業や工程に対応できる多能工人材の存在は貴重であり，我々は積極的にその育成を行っている．そのような多能工人材の育成には，複数の異なる作業や工程の経験が必要であり，次のようなことが重要と考えている．長年の経験などで積み上げられたルールや作法が暗黙知となり第三者から分かりにくい点を分解・可視化し新規参入者への理解を容易にする，関係者のフォローや協力によってスキル不足を補完する合意形成をする，などである．

本経験論文では，システム理論に基づく事故モデル STAMP とその分析手法である STPA や CAST が，暗黙知の理解を助けスキル補完の合意形成を促進し，多能工を促進する組織のツールとして役立った一例を報告する．

1. はじめに

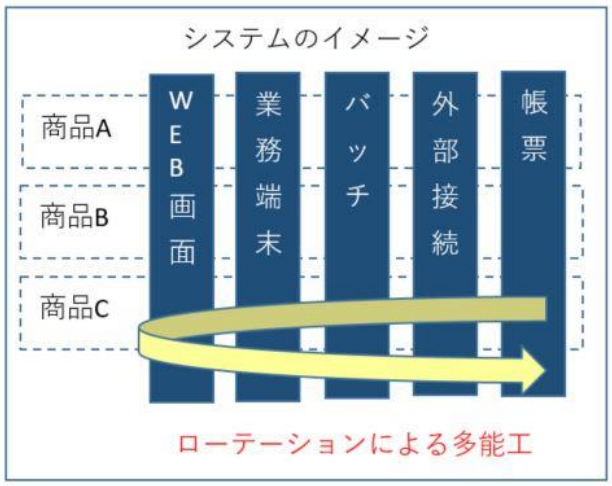
1.1. 継続保守・開発，大規模ゆえの中期人材育成

我々は基幹系システム全体，アプリケーション・インフラ・運用の開発・保守を担当している．開発案件の担当工程は立ち上げから導入まで全て，保守は 24 時間 365 日である．担当領域が非常に広く，様々な知識と技術を必要とする．毎月なんらかの改修を行っている機能もあれば，1年に1回改修が入るか入らないかの機能があるなど，頻度や仕事量は一定ではない．法令改正や新サービス追加で，ある時期だけ仕事量が急激に増えることもあり，要員配置の苦労は尽きない．そのような中で，様々な仕事に適応できる多能工人材の存在は，人材不足解消・ノウハウ維持・生産性と品質維持など多数の有益な価値を生むため，積極的な育成を委託先とも協力しながら行っている．

1.2. 本経験論文の多能工

多能工人材のパターンは一つではないが，本経験論文における多能工人材とは，WEB 画面・バッチ・外部接続機能など，複数サブシステムの開発工程を担当できるエンジニアのことを指す．図1．では，5つのサブシステムを持つシステムを示している．2 つ以上のサブシステムで開発を行える人材を，多能工人材とする．

図 1. 本経験論文における多能工前提



1.3. 本経験論文のねらい

本経験論文では，そのような多能工推進の中で発生した，暗黙知とスキル不足を要因とする事故の分析に STAMP 手法を活用してモデリングした事例を通じて，以下を考察する．

STAMP モデリングが，
Q1. 暗黙知の分解にどのように役立ったか

Q2. スキル不足の補完にどのように役立ったか
Q3. 共感と共創に役立つものか
Q4. 多能工に好影響を与えるのか

あわせて本事例の STAMP 手法を用いたモデリングの経緯を報告することで，同手法やモデリングの導入に役立つ情報提供を目指す．

2. 多能工促進方法と課題

2.1. 多能工促進の方法

多能工人材を育てる一般的なアプローチは，あるサブシステムで経験と実績を積んだメンバーを他チームへローテーションさせる方法である．ローテーション前後の開発言語は同一とは限らない．サーバアプリとクライアントアプリでは開発環境も異なる．オンラインとバッチでは設計に対する基本的な考え方も同じではないなど，様々な違いがある．別のサブシステムで一定の成果を出してきたメンバーやリーダークラスであった場合でも，ほぼゼロからのスタートというケースも少なくない．

なお，プロジェクト管理方法・課題管理方法・レビュー実施における必須成果物群・他サブシステムと協業で作成する成果物群などは全プロジェクト共通だが，プロジェクトで管理するタスクの詳細やタイミングなどはサブシステムの裁量に任せられており必ずしも共通ではない．

2.2. 多能工促進の課題

多能工推進を組織として明示的に掲げてから 5 年以上になるが，経験則から問題となりやすい点が二つあると感じている．一つは暗黙知の問題．もう一つが暗黙知と合わさった複雑なスキル不足の問題である．今回は，この二つの課題に焦点をあてる．

2.2.1. 暗黙知が新規参入を難しくする

本経験論文では暗黙知を次のように定義する．「新規参入者には理解しにくい，新規参入者を受け入れたチームではあたりまえの知の集合体と現状．ルール・規律，言語パッケージやフレームワーク構造，作法・判断基準など，普段は言及されないもの全般を含む．」 例えば，外部接続先とのインターフェース仕様書に記載されている情報は暗黙知には含まないが，その設計思想や，様々な実装者が修正していったソースコードの経緯など，知の集合体を背景に持つファイルそのものと，そのファイルを取り扱う前提となる様々な作法や期待される行動などの全般を指す．

暗黙知に起因する問題の一例を示す．新規参入者が新しいチームのルールや作法が分からず，過去所属していたチームのご作法で成果物作成を行った結果，そもそもの前提が誤っていたためチームの期待に応えられないケースがある．例えば，成果物の求める達成レベル（早く出してチェックを繰り返す，じっくり最後でチェックする違いなど）の考え方が異なり，双方が期待したタイミングで成果物が作成されない状況などである．受け入れたチームが早く出してチェックを繰り返すスタイルにもかかわらず，参入者がじっくりと時間をかけてしまう場合，新規参入者が不慣れであることを考慮すれば，初回のチェックで十分な結果が得られることは予想しにくい．結果，余計な時間をかけたことによるコストや納期への影響，納期ひっ迫原因による品質への影響などの悪い要因を生むことは容易に想像できる．逆にチームがじっくり最後でチェックするスタイルで参入者は早く繰り返しスタイルの場合，チームメンバーが「新規参入者にはこの程度のスキルしかないのか」と誤解し，新規参入者の今後の協業に様々な支障が出る可能性もある．いずれにせよ，プロジェクト QCD に何らかのかたちで影響し，受け入れチームと新規参入者の双方にストレスになることは間違いない．

問題を複雑にするのは，こういった暗黙知は長年チームで蓄積されたノウハウや認知的側面[2] が多いことによる．SECI モデル[3] のように歴史と共に変化・進化し，プロセスや成果物フォーマットなどにちりばめられつつ，その現場の慣習や文化になったものは，可視化が難しい．7S の「ソフトの 4S（Shared value , Style , Staff , Skill）」[4] に染み渡った状況であり，プロセス資産，個人のスキルや意志・行動，文化といった組織の環境要因に相互依存しながら混じり合い，境界線を引くことも難しい．そこにいる当事者であっても，その構造を端的に説明することは難しい場合も多い．受入チームメンバーが言語化出来ないものを，新規参入者が簡単に理解できるはずもなく，問題をより複雑化させる．

文献[2] (野中郁次郎と竹内弘高の著書『知識創造企業』，梅本勝博訳，東洋経済新報社 P9）では，暗黙知の認知的側面を次のように述べている．

> 暗黙知には重要な認知的側面がある．これに含まれるのが，スキマータ，メンタル・モデル，思い，知覚などと呼ばれるもので，無意識に属し，表面に出るこ

とはほとんどない．この認知的側面は，我々が持っている「こうである」という現実のイメージと「こうあるべきだ」という未来へのビジョンを映し出す．簡単には言い表せないこれらの暗黙的モデルは，我々が周りの世界をどう感知するかに大きな影響を与える．

2.2.2. 十分条件にならないスキル要件

例えば Java での開発業務にもかかわらず Java 自体を知らないような純粋なスキル不足はここでは扱わない．現場で発生しやすいスキル不足の問題は，必要条件としては整っているものの十分条件まで満たされておらず，問題になるケースである．

例えば，SQL を使って開発しているオンラインアプリとバッチアプリがある．どれも使う SQL 構文は変わらない．しかし，オンラインアプリであれば，ロックを最小限にする工夫を求められたり，バッチであれば短時間に大量データを処理ことが必須であったりする．非機能面の要求は「SQL」という言語記号だけでは表現することが出来ない．長年継続開発・保守を担当しているチームでは，非機能要件を満たすための工夫が暗黙知になっており，全体の構造まで明示的に説明されないケースも多い．事前に「性能を意識したコーディングをして下さい．」と伝えても，具体的な How が暗黙知となっており，新規参入者に正確に伝わらず，受入チームの期待に応えられないケースもある．

Java のように，たくさんの機能拡張されている言語分野でも問題が発生しやすい．「Java を使います」では Java EE や J2EE のどの機能まで必要なスキルになるのかを十分に説明出来ていない．開発環境はどうなっているか，自動テストをどのように行っているかなどでも同様のことが起こる可能性がある．受入チームからすればあたりまえの事実になってしまっていることが事前に説明されず，開発をスムーズにスタート出来ず，問題となる場合もある．

3. モデリング対象選定の経緯

あるチームで発生した事故の原因分析と再発防止策策定を対象にモデリングを行った．この事例では関係者が納得のいく再発防止策をなかなか策定できず，困り果てていた．

事故の直接原因は「ソースコードの不適切な修正とテスト未実施による検出漏れ」であり，その報告だけを聞いた段階では，モラルハザードの発生が疑われた．しかしながら，詳細なヒアリングにより背景にはスキル不足があり，加えて，次の気になる点があることが分かった．

3.1. 指示・受入側もスキル不足は理解しており，フォローに対して積極的であったにも関わらず，事故が発生してしまった．
3.2. 作業実施側も何とか期待に応えようと努力しており，チェックリストやレビューも実施済であったが，事故になってしまった．
3.3. 双方，やるべきことを適切にやっていたと信じており「まさかこんな事故になるとは」という気持ちと共に，自信を失っていた．

ヒアリングを通じて，現場から感じられたのは真剣さと善意であり，開発現場でモラルハザードが起きていないことは，すぐに理解できた．委託先も関連しており，慎重に対処する必要がある．委託先とは長年良い協業を続けていることもあり，実態を明らかにして，今回の失敗を教訓としてより良い状況に発展させたいと考えた．

そのため，納得できる再発防止策を策定してもらおうと，当事者で会議を繰り返してもらったが，お互いが納得できる解決策まで得られず，これまでとは異なったアプローチを必要としていた．

このような背景の下，今回の事故の特徴と STAMP の相性が良いと感じ STAMP を選択した．今回の事故は関係者も「まさか」と感じる意外性を持つ事故であった．原因も一つの単純な問題から派生したものではなく，いくつかの相互に依存する問題が重なり合った結果として事故になっているように感じた．これが STAMP の事故（望んでもないし計画もしてない損失につながり得るイベント），ハザード（環境のある最悪な条件の集合と重なると事故になり得る，システムの条件もしくは条件の集合）[5] (P16) と非常に似通っていると感じ STAMP を採用した．

4. モデリング経緯と結果

4.1. 登場人物

事故が発生したプロジェクトのチームの登場人物は次の通り．

① 指示・受入管理者：全体責任者

② 指示・受入責任者：実務遂行する責任者
③ 作業実施管理者：委託先責任者
④ 作業実施責任者：作業実施責任者
⑤ レビューア：主に設計書・ソースコード・テスト結果レビューを担当
※作業実施責任者が兼務する
⑥ 作業担当者：作業実施者の指示のもとで，設計書修正，コーディング，テスト実施

表1 登場人物のスキル

	所属	スキル充足度	特記事項
①	委託元	問題なし	
②	委託元	問題なし	
③	委託先	問題なし	
④ ⑤	委託先 （兼務）	問題あり 新規参入者	技術要素が異なるローテーションで不慣れ
⑥	委託先	問題なし	スキルに問題はないがミス多い

4.2. モデリングの流れ.

以下の順序でモデリングを行った.
4.3. モデリング前の再発防止策を整理
4.4. 初回モデリング
4.5. 初回 CAST アプローチ
4.6. 経緯と歴史の紐解き
4.7. CAST アプローチ再実施
4.8. STAMP モデリング実施

4.3. モデリング前の再発防止策を整理

モデリング開始前に委託先，委託元の関係者を集めてヒアリングを行った．獲得した情報は以下の通り．「→」に続く個所は考察を含む．

a. モラルハザードはない．
b. 言いたいことは言える関係がある．
c. 委託元作業指示以外の作業は行わないルール．
※c. は事実だが一部情報が欠落，後の過程で情報が追加される．
d. ソースコードの不適切な修正は，作業担当者の思い込みによるミスで，指示・受入責任者が指示した箇所以外を不用意に修正してしまった．
→ ミスは誰でもあることなので完全には防げない前提で考え，レビューやテストの安全制約があることを確認した．
e. テストが未実施
→ 役割分担としてやるべきことが出来ていない，プロセス違反（なぜ？）
※e. この段階の見解で事実と異なる．後の過程で見解が変わる．
f. レビューア d., e. の欠陥を，レビューで取り除く役割だが，指摘出来ず．スキル不足が背景にある．
※f. この段階の見解で事実とは異なる．後の過程で見解が変わる．
g. レビューアは多能工推進で別サブシステムからローテーションされたメンバーで経験が浅い．
→ 多能工は双方合意の事項であり，e.を発見できなかった原因である f. が問題とは単純に言えない点で，双方に相当のストレスがある．
h. DIFF による対象範囲外修正の有無は委託先で行っており，実施したが「問題なし」と判断してしまった．委託元（受入側）では実施済の確認をいつも通り行っていた．
→ 長年の経緯で，数年前から委託先で行われることになっている．これまで事故はなかった．委託元でもチェックすべきではないか？
i. 再発防止策として，委託元でも委託先と同様の DIFF チェックを行うように変更．

4.3.1. モデリング開始前の関係者心境

この時点の関係者の心境を纏めておく．

委託元には委託先の不満が溜まっている．関係者がやるべきことすら普通にやれない状況と理解しており，何を改善すべきかと途方に暮れている状況だ．

委託先では，決められたルールが守れない自己嫌悪と，不慣れなメンバーがいる中での失敗をどう捉えるべきなのかの迷いがある．双方，どこかで納得できない部分があり，解決に向かうにしても，何に焦点をあてて対話していけば良いか分からない状態である．

再発防止策で，委託元で確認するプロセスを追加（i.）したことで，ひとまずの解決策までは出来上がり，顧客へ

の説明も出来たが，委託元が委託先に抱く不信感は根強いものになってしまった．委託先も自信を失っており状況は好ましくない．長年の良い協業関係を今後も続けていくためには，双方が納得できる現状分析が必要である．

4.4. 初回モデリング

ここまでの情報でモデリングしたものを，図 2．初回モデリングに示す．

この段階では，ヒアリングや当時の成果物チェックで得た情報から，安全制約は何かに着目し作成した．結果はシンプルなものとなった．しかし，STAMP/STPA 導入を決めた際に着目したハザードが捉えられていない点で，期待どおりとは言い難い．モデリングは 1 回で全て出来上がるものでは無いので，少し視点を変えて臨むことにした．

初回モデリングでは安全制約に着目したので，次は CAST アプローチを用い，ハザードと思われる状況を現場目線からのボトムアップで確認することにした．

特に以下の 3 点に着目した．

4.4.1. e. の問題がなぜ起きているのか不明
4.4.2. f. の配置は適切だったのか
4.4.3. なぜ，g. であることは理解していながらも f. の可能性を予想できなかったのか

4.5. 初回 CAST アプローチ

まず 4.4.3.「なぜ，g.であることは理解していながらも f. の可能性を予想できなかったのか」を紐解くためにヒアリングを行い，以下 c.「→」以降の追加情報を得た．

c. 委託元作業指示以外の作業は行わないルール
→ 委託先からのフィードバックや問い合わせをもとに作業指示に追加される場合がある．
委託元はフィードバックの有無で，作業指示の妥当性を確認している側面がある．

追加情報から派生して j. を獲得した．

j. 受入まで問い合わせが無かったので，適切に理解できていると思っていた．「何か分からなければ聞いて下さい」と伝えてあり，これまでそのスタンスで問題なかった．

「聞いてくれるものだ」との期待が j. にあり，これまでのベストプラクティスであることがうかがえる．なぜ，この期待に沿えないのかを安全制約の観点から考えてみる．

委託元には暗黙の安全制約（分からないことは聞いてもらえる）が存在していることになるが，委託先ではこの安全制約を適切に運用できていない状態と言える．コミュニケーション問題の有無については，b. からも双方良好なコミュニケーションは確立できていると認識しており，会議

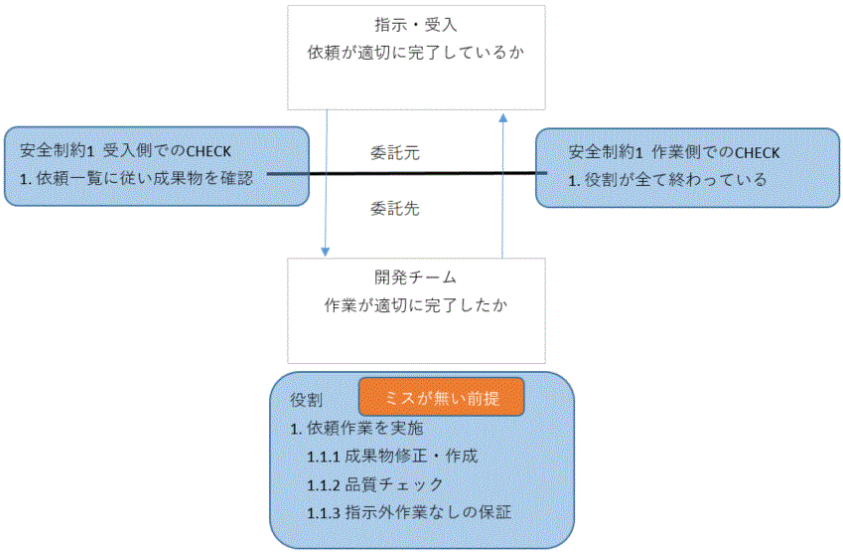


図 2．初回モデリング

の発言などを第三者がみても，心理的な障壁などで対話にならない障壁があるとは感じられなかった．

次のような仮説を立てた．「何を知っている必要があるか，何が期待されているのかといった前提の暗黙知を知らず，分かっているか否かを感知できていない，そもそも理解していない状況」と仮定した．片方は期待している一方，片方は期待を理解するレベルではなく期待の存在すら分からないとすると，双方がかみ合うことは難しい．こういった暗黙知を前提とする安全制約のずれに着目することで，真因にたどり着けると考えた．さらなる考察のために少し視点を変え，これまでの経緯をさかのぼってみることにした．

4.6. 経緯と歴史の紐解き

暗黙知を満たす期待を紐解くため，これまでの経緯，委託先と取引を開始した当初の約 10 年前までさかのぼって考えてみたのが k., l., m. である．

k. 協業開始当初，委託先リーダーは委託元の個別指導とともに仕事を一通り覚えた経緯がある．そこで，プロセス・成果物・開発環境など見えやすいものと，必要なスキル・コミュニケーション方法・判断基準などの見えにくいものを獲得し，委託先へ戻った．
l. 委託先が仕事を覚える過程で，「これで良いですか？」という細かな問いが頻繁に繰り返された．双方のフィードバックで暗黙知を築き上げていくコミュニケーションがあった．その中で，委託元は伝えられているか，委託先は理解しているかを繰り返し確認しながら，伝わりにくい部分をドキュメントに残すようにし，齟齬が発生しやすいタイミングで会議体を設定するなど，双方でより良い仕事の仕方を構築しながら，互いの暗黙知を築き上げたと言える．SECI モデル[3]の実践である．
m. 委託先メンバーが仕事を覚え，委託元と同じ基準で仕事ができる段階になり，仕事の役割分担を委託先に徐々に移譲していった．状況ごとの判断基準などの期待も含めて，委託元が理解していることが仕事の前提になっており，双方の暗黙知になっていた．

k., l., m. を SECI モデル[3] で再考する．SECI モデルは，共同化，表出化，連結化，内面化を繰り返す．そこには対話，形式知の結合，行動による学習，場作りがある．([2] の「図 3-4 知識スパイラルとその内容の変化」が分かり易い) k., l., m. を通じて共同化があり，「問い」を通じて表出化されている．その中で，どのように協業していくのがベターなのかを連結化し，成果物やプロセスという内面化を通じて個と組織にナレッジを蓄えていく．この暗黙知をベースに品質と生産性を作りこんでいる．

現状の c., j. との違いを考えてみる．c., j. は共同化と表出化が弱くなっていると言える．「知識創造企業」[2]（P360）では，トップダウンは連結化と内面化に焦点が当たり，ボトムアップは共同化と表出化に焦点が当てられていると述べられている．この考えを取り入れると，ボトムアップの取り組みが弱くなっていると言える．かつてはこの両極を統合するミドル・アップダウン・モデル[2](P360) だったと言えるだろう．暗黙知の共有が崩れたことで，そのバランスが崩れた状態と言える．

過去の過程を振り返る中で，もう一つ明らかになったことがある．h. の情報を分解した結果だ．

n. DIFF による対象範囲外修正の有無は，委託先と協業を始めたころは指示・受入担当者が行っていたが，最近の数年間は委託先に移譲しており事故は発生していない．

モデリング前に，DIFF による対象範囲外修正の有無確認を委託元が行っていないことに，やるべきことをやれていないのではとの違和感があった．実際には，m. から過去の経緯で役割を委譲したことが分かり，以後も事故は起きていないことから，しばらくは問題が無かったと言える．それが意図した運営が機能しなくなるにつれ，「より高いリスクに向かう経時的な推移 –例えばより高い効率や生産性の追求で何かを軽視」[5] (P8) に近い状況に遷移してしまったことが理解できた．当初は暗黙知とスキル充足でコントロール出来ており，生産性追求のために役割を委譲したが，経年の体制変更や仕事量の増加などで，コントロールが不十分になったと言える．

ここで難しいのは，意図的にルールを変え，高いリスクに向かう経時的な推移を推進したわけではないことだ．前提とした暗黙知が機能しなくなったことで，状態がハイリスクに遷移したとも言える．この件は課題とし，本件のモデリングからは除外することにした．

4.7. CAST アプローチ再実施

ここまでの過程で理解できたことを整理する．スキル不

足とは一般的には個人に依存するように語られるが，実際には組織構造からくる暗黙知不理解といった，安全制約をコントロールできないスキル不足も存在すると言える．であれば，知らないことを個人の問題としてとらえるのではなく，構造の問題としてとらえることが必要だ．「何が／誰が悪い？」ではなく「なぜ／どのように？」コントロール出来ていない状態なのかを追求することが必要である．この分析に強いのが STAMP の特徴 [6] (P8) だ．その視点で，再度，インタビューを行い，以下の 3 つの疑問を解決する解を得られた．

4.4.1 e. の問題がなぜ起きているのか不明
4.4.2 f. の配置は適切だったのか
4.4.3 なぜ，g. であることは理解していながらも
f. の可能性を予想できなかったのか

o. e. の原因は，テストは出来ていると作業担当者が勘違いした，が正しい．修正モジュールは画面から必ず呼び出されるものと勘違いし，画面テストは実施済なので問題なしとした．しかし，該当モジュールは画面から直接は呼び出されないコードで，別の方法でのテストが必須であった．プロセス違反ではなく，思い込み・勘違いによるミスであった．

p. e. のもう一つの原因として，レビュー時に問題を検知出来ていたが，対処が為されなかった事実が確認できた．レビューアは作業者に対し，d. 指示箇所以外の修正の妥当性確認を指摘しており，レビューアとして機能している．その指摘を作業担当者が c.「→」以降のケースもあるため，問題ないと回答してしまったことが一因である．当初の様子を，「不慣れな私でもおかしいと思ったが，私よりも経験の長い担当者が『問題ない』と言ったことで，担当者の方が正しいと判断してしまった」という背景があったことを理解できた．

q. レビューアは他サブシステムの一定実績を評価されローテーションされており，o. の実績からもレビュースキルは保持している．レビューアが多能工のローテーションから経験期間が短く，技術スキルは十分と言い切れないことを，委託元も把握済，要フォローのステータスであった．その為，本作業をお願いする前に，修正箇所を事前にチェックし，実装難易度も高くないものに限定して仕事を卸した経緯があった．その為，技術的な問題は発生しないと判断．何かあった場合は，j. のとおり問い合わせが発生する想定で可能性は予見していた．

以下にまとめを述べる．

4.4.1. o. から思い込み・勘違いによるミスと分かる．
→ ミスは誰でもあることなので，完全には防げない前提で考え，レビューやテストの安全制約があることを確認した．
4.4.2. p., q. から最適とは言えないが，最善であったと言える．
4.4.3. 委託先と委託元ともに予見していた．

4.8 STAMP モデリング実施

これまでの経緯から，再度モデリングしたものが，図 3. STAMP モデリングである．

4.8.1 モデリング実施後の関係者心境

初回モデリングから STAMP モデリングに行きつくまで，問題の抽出とそれらの相互依存関係を明らかにするため，かなりのヒアリングを要した．結果としてプロセス上省略した箇所はないことが分かり，関係者の中で暗黙知に起因したハザード（環境のある最悪な条件の集合と重なると事故になり得る，システムの条件もしくは条件の集合）[5] (P16) に起因していると合意形成できた．ミスを前提としたレビューなどの安全制約でいつもなら防げるものが，偶然にも複数の状況が重なり発生した事故であることが共有でき，事故の複雑性を関係者が理解できたことで，不信感が消えていった．長年の良い協業関係を今後も続けていくため，難しい問題に立ち向かう気概が出てきたことがとても大きい．

結果として，本件をケーススタディとしながら，ふりかえりや勉強会を実施し，暗黙知を意識しながらプロセス改善やコミュニケーション改善を行っていくという再発防止策を取り入れることが出来た．

5. 考察

「1. はじめに」で設定した，STAMP モデリングに関する Q1～Q4 の考察を述べる．最後に STAMP モデリングの感想を述べる．

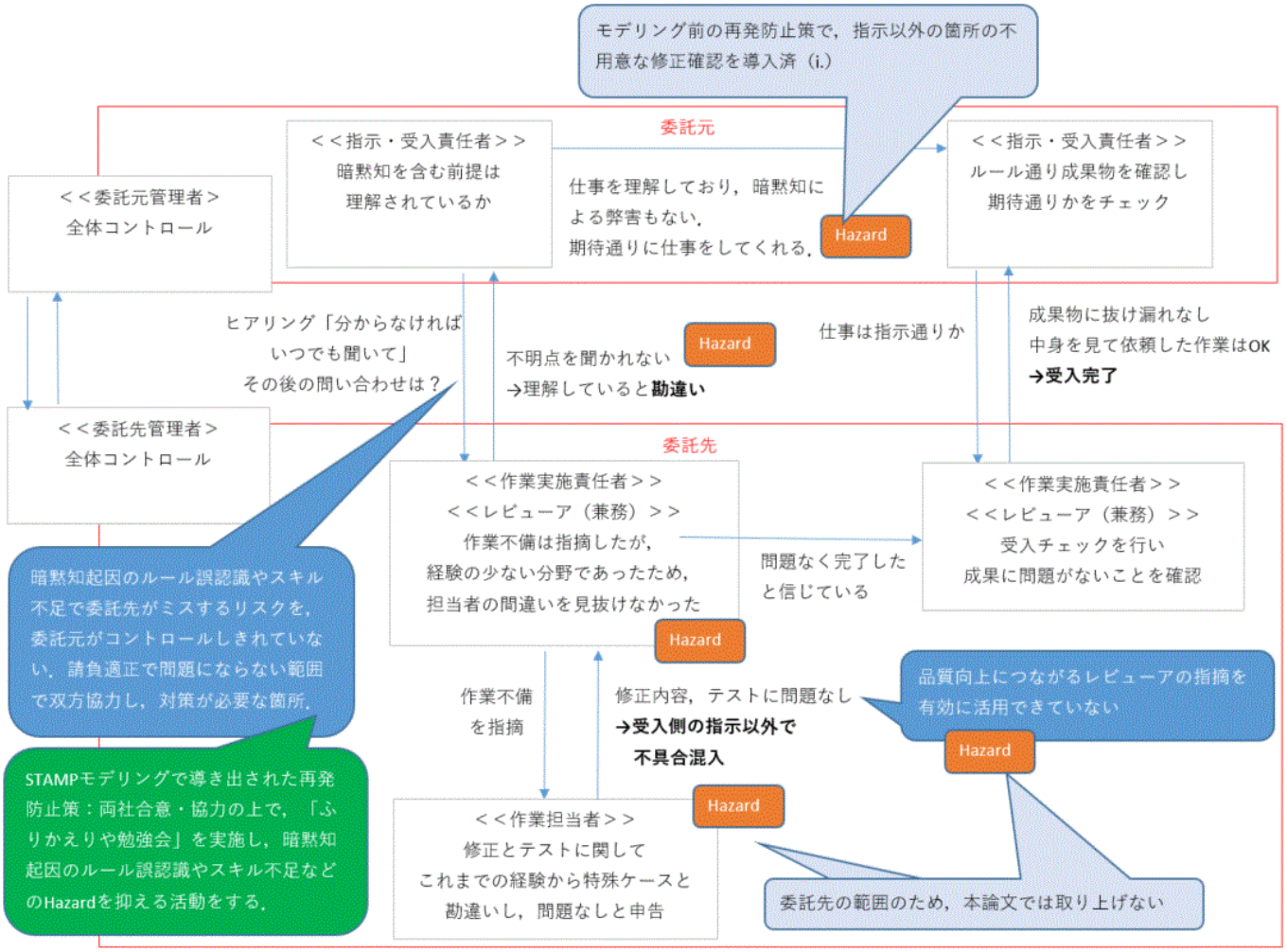


図 3. STAMP モデリング

5.1. Q1.暗黙知の分解にどのように役立ったか

暗黙知分解は時間軸をさかのぼり，かつ複数の関係者がどのように共同化，表出化，連結化，内面化を繰り返していたかを可視化する必要がある．一方向的な思考だけでは捉えることが難しい．そこで STAMP の特徴である非線形的なループ構造，コントローラーと被コントロールプロセスという視点で着目していくことが役立った．相互依存で関係者がどのように合意形成したのか，個人の成長とともに組織にどのようなドキュメントやプロセスを残しながら暗黙知として品質や生産性を上げていったのかを分解していく思考に役立った．

今回行った，情報収集→初回モデリング（ラフスケッチ）→CAST→暗黙知分解→CAST→STAMP の流れで考えていくと，暗黙知分解を効率よくできることが分かった．特に，暗黙知分解→CAST→STAMP と続くことで，相互依存関係を明記する必要があり，「なぜ／どのように？」コントロール出来ないかを書かなければモデリングが終われない．暗黙知が存在するだけで終わらせず，暗黙知がコントロールに及ぼす影響まで可視化される．この点で STAMP/STPA は強力なツールと言える．

5.2. Q2.スキル不足の補完にどのように役立ったか

組織全体にちりばめられている暗黙知を把握することで，構造からくる暗黙知不理解といったスキル不足が存在することを，関係者で合意形成できる．

スキル不足は努力不足とも捉えられ易い側面がある．暗黙知不理解は個人の努力だけでは情報を入手することも難しく，周りの協力が必要であることを可視化できる．

暗黙知不理解に対するスキル補完は，個人の努力に加えて，組織のサポートが必要だということを可視化し，合意形成のツールとして利用できる．

新規参入者がチーム仕事の暗黙知理解を助けるツールとして利用できる．合わせて新規参入者を受け入れるチームに対し，暗黙知がどのようなリスクをもたらすかの可視化に役立つ．

5.3. Q3.共感と共創に役立つものか

共感と共創に役立つものである．

ルール違反という一見単純に見えた事故であったが，STAMP モデリングで分析した結果，ハザードが複数重なり合ったことに起因しており，その複雑性を可視化することができる．ルール違反で止まってしまうと，モラルハザードの発生や個人が悪いという判断になり易く，共感と共創とは逆の方向に行きやすい．モデリングで複雑な構造を明らかにしていくことで，モラルハザードではない構造上の複雑な問題であることを可視化できる．関係者同士の相互依存性も明らかになり，共感につながる．共感が関係者の協調を促進し，事故対策でも具体的なより良い施策を明確にすることができ，共創を促進するものである．

5.4. Q4.多能工に好影響を与えるのか

多能工の複雑性について，関係者が合意形成でき，リスクを可視化できるという点で，好影響をあたえる．

5.5. STAMP モデリングを終えて

実施後の全体的な所感を述べる．

STAMP は非常に強力なツールであることは間違いない．しかし，誰もが簡単に使えるものでは無いと感じた．実は，図 2. 初回モデリングも，自分の中では STAMP で作成したつもりであった．構造だけみればコントローラー，被コントロールプロセス，コントロールアクション，フィードバックデータに基づいた記載になっている．しかし，この段階ではハザードも相互依存関係も明らかに出来ていない．STAMP の良さは，誰が／何が悪いのかではなく，なぜ／どのようにコントロール出来ていないのかを明らかに出来ることだ．そこまで行きつくために，試行錯誤を繰り返した．実は，これが STAMP の良いところだと感じている．構造的には書けていても，そこで終わらせないところである．なぜ／どのようにコントロール出来ていないのかまで明らかにしようとすると，必然的に依存関係まで明らかにする必要があり，そこまで書かないと納得できるものが作成できないのが利点だと感じた．

単純な線形思考の場合，先に答えを仮定して，その答えに合う情報で肉付けして終わらせてしまうことも出来る．STAMP では，フィードバックプロセスがあるので，それが難しいと感じた．仮定がある側面だけから導き出されたものだとすると，フィードバックサイクルの中で矛盾がきちんと出てきてくれる．そこにハザードが潜んでいるケースが多かった．それらを紐解くと新たな矛盾が出る．それらを繰り返すと，全体がつながるようになった．

暗黙知分解の箇所は，当初を経験していたメンバーの存在も大きいと言える．STAMP の持つコントローラー，被コントロールプロセス，コントロールアクション，フィードバックデータモデリングが思考と理解を促進したのは事実であるが，経験メンバーが不在であれば，詳細の可視化までは難しかったかもしれない．

6. おわりに

システム理論に基づく事故モデル STAMP/STPA の手法が，現場の複雑な構造を可視化出来ることが分かった．暗黙知の分解とスキル不足補完といった複雑な問題にも役立つ結果が得られた．共感と共創をつくり，多能工を促進するツールの一つであると言える．多能工推進は人材不足解消に結びつくことを考慮すると，現場の人材不足解消を促すものとも言える．

効率性を追求する組織では，組織の暗黙知が大きくなり，新規参入者のハードルを上げ，同時に既存チームの負荷も上げてしまう．これらの問題を解決するのに，STAMP モデリングが有益なツールであることが分かった．

引き続き，暗黙知の構造分析とともに継承に役立て，組織における構造の問題，そこに潜むリスク可視化，問題発生時の真因特定に役立てていきたい．

プロジェクトマネジメントやプログラムマネジメントといった管理分野でも，構造理解に利用できると考えており，積極的に活用したいと考えている．

参考文献

[1] はじめての STAMP/STPA ～IoT 時代の新しい安全性解析手法～, 石井正悟
https://www.ipa.go.jp/files/000053857.pdf , アクセス日時 2018 年 3 月 10 日 10:00

[2] 知識創造企業, 野中郁次郎・竹内弘高, 訳者:梅本勝博訳, 東洋経済新報社,
ISBN978-4-492-52081-9

[3] SECI model of knowledge dimensions,
https://en.wikipedia.org/wiki/SECI_model_of_knowledge_dimensions , アクセス日時 2018 年 3 月 12 日 19:00

[4] 7S,
https://mba.globis.ac.jp/about_mba/glossary/detail-12513.html , アクセス日時 2018 年 3 月 12 日 19:00

[5] システムモデリングの新潮流:STAMP/STPA システム理論に基づく新しい安全解析手法,
日下部茂,
https://www.ipa.go.jp/files/000053967.pdf , アクセス日時 2018 年 3 月 9 日 20:00

[6] 宇宙機における不具合分析手法 CAST の適用,
寺田庸弘・星野伸行,
https://www.ipa.go.jp/files/000036240.pdf , アクセス日時 2018 年 5 月 12 日 19:00

アジャイルの振り返りとシステム・シンキングの有効性について ～ロジカル・シンキングは万能ではない～

日山　敦生
緑ビジネスコーチ研究所
hiyama@k04.itscom.net

要旨

アジャイル開発の振り返りには，フレームワークの KPT がよく使われる．問題の分析手法であるロジカル・シンキングは，技術的な問題には非常に有効であるが，万能ではない．人や組織の問題には，システム・シンキングを基本とする解決志向アプローチが，有効である．原因を究明しない解決志向アプローチを用いることにより，チームの強味の再利用や，問題の迅速な解決が容易となる．アジャイル開発の振り返りに，解決志向アプローチを用いることで，設計品質のさらなる向上に，貢献できる．

1. はじめに

近年，益々，巨大化，複雑化するソフトウェアに反比例して，今まで以上に短納期が求められている．複雑化するなかで，要件定義を開発初期段階で，明確に定義することが困難な状況がある．そこで，反復開発を行うアジャイル開発が注目されている．アジャイル開発は，反復開発のサイクル毎に，振り返りを行うことにより，設計品質を高めていくことができる．この振り返りで使われているのが，振り返りのフレームワークである KPT である．振り返りでは，技術的な問題だけではなく，人や組織の問題についても，振り返りが行われている．

技術的な問題については，原因究明を基本とするロジカル・シンキングが有効であるが，人や組織の問題については，原因究明を行わないシステム・シンキングが有効である．日本では，幼いころから社会人に至るまで，問題を解決するためには，原因を究明し，原因を取り除くことによって，問題は解決できると学んできた．バグのような技術的な問題には，原因究明を基本とするロジカル・シンキングが有効であるが，人や組織の問題は，複雑で，原因と結果が相互に悪循環になっているとするシステム・シンキングが，問題解決に有効である．

心理療法から生まれたシステム・シンキングを基本とする，解決志向アプローチがある．解決志向アプローチを，ビジネスの人や組織の問題解決に用いると，迅速に，問題解決ができる．

また，良かったことは，項目としてあげるだけではなく，深堀して，再利用することが重要と考える．苦労して上手くできたことを，再利用しないで使い捨てにしてしまうのは，もったいないと考える．

アジャイルの振り返りのフレームワークである KPT に，システム・シンキングを基本とする解決志向アプローチを適用した振り返りのフレームワーク ROTT を考案し，ワークショップを開催しているので，ご紹介する．

2. KPT と ROTT

2.1. KPT とは

KPT（図 1）は，アジャイルの振り返りでよく使われるフレームワークであり，Keep：良かったこと，Problem：悪かったこと，Try：次回，試してみることで，構成されている．

KPT は，悪かったことだけではなく，良かったことにも注目する振り返りの優れたフレームワークである．

KPT は，ロジカル・シンキングの視点から見ると優れたフレームワークであるが，システム・シンキングを基本とする解決志向アプローチの視点からみると，残念な点が2つある．1つは，良かったこと（Keep）の深堀が，不明確であること．もう1つは，悪かったこと（Problem）が，物や技術の問題と，人や組織の問題に分かれていないことである．そこで，KPT の改良版として ROTT を提案したい．

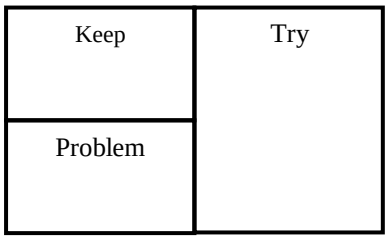


図 1　振り返りのフレームワークの KPT

2.2. ROTT とは

KPT に，解決志向アプローチのキーワードである「成功の責任追及」を用いたフレームワークとして，ROTT（図 2）を考案した．ROTT は，Recycle：良かったこと（成功究明，目的究明），Organizational Problem：組織的問題（解決究明），Technical Problem：技術的問題（原因究明），Try：次回，試みることで，構成されている．

良かったことは，Keep ではなく，Recycle とする．Keep には，維持するという意味合いが強く，深堀して再利用するという意味合いが乏しい．苦労して出来たことを深堀せずに，次回も，ゼロから始めるのは，もったいないと考える．Problem は，解決究明が重要な人や組織の問題（Organizational Problem）と，原因究明が重要な技術的問題（Technical Problem）に分けることが必要である．解決究明と原因究明では，問題解決のアプローチが大きく異なる．

Recycle	Organizatonal Problem
Try	Technical Problem

図 2 振り返りのフレームワークの ROTT

ROTT の作業手順（図 3）は，一度に，③組織的問題と④技術的問題に分けることが難しいので，②悪かったこととして，一度，項目出しを行ってから，分離作業を行う．

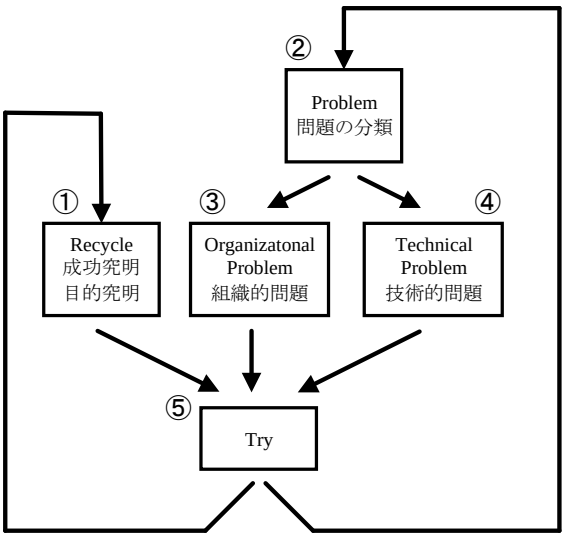


図 3 ROTT の作業手順

3. ROTT

3.1. Recycle

良いことは維持するのではなく，良かったことは深堀を行い，良い循環を強化することである．良い循環を強化するキーワードとして，「成功の責任追及」がある．失敗の責任追及があるように，成功の責任追及があって当然と考える．失敗の責任を追及されることは辛いが，成功の責任追及は，自慢話を聞かせてくださいということで，いくら追及されても辛くはない．ヒヤリ，ハットしたことも，最終的にカバーできたということであり，ヒヤリ，ハットしたことをどのようにカバーできたかの，貴重な対策資料となる．

良かったことは，2 種類に分類できる．抽象的なことと具体的なことである．抽象的な例として，「顧客と信頼関係が築けた」，具体的な例として，「議事録を書くことができた」を取り上げる．

抽象的なことである「顧客と信頼関係が築けた」（図 4）の成功究明は，どのような自分たち行動が，顧客の信頼関係を築くことに，貢献したのかの究明を行うことである．顧客の信頼関係を築くために，有効であった行動が明確になれば，既に自分たちが出来ていることなので，次回も，有効であった行動を容易に繰り返すことができる．

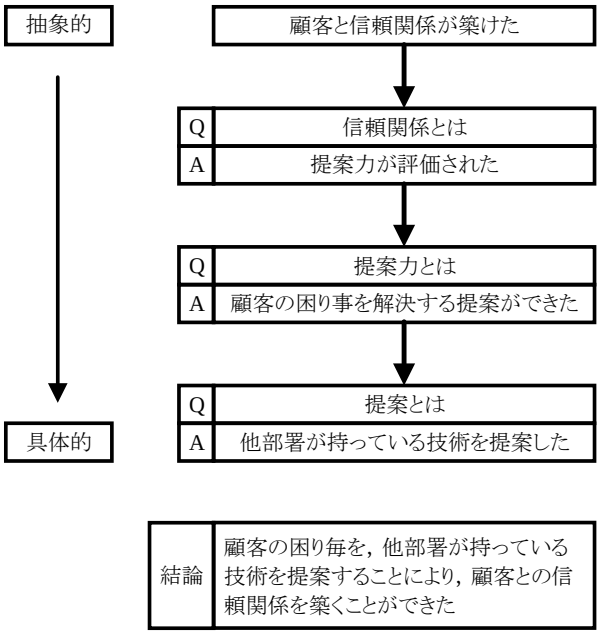


図 4 顧客と信頼関係が築けた

具体的なことである「議事録を書いた」（図 5）であれば，

目的究明を行い，仕様に関して顧客と相違が生じた場合に，自分たちが正しいことを証明する目的が，今回，有効に機能した．さらに，深堀すると，顧客に勝つことが目的ではなく，仕様に関して顧客と相違が生じないようにするためには，何が必要かという問題が浮かび上がる．

判りにくい所には，具体例を入れた仕様書にするという対策が生まれる．

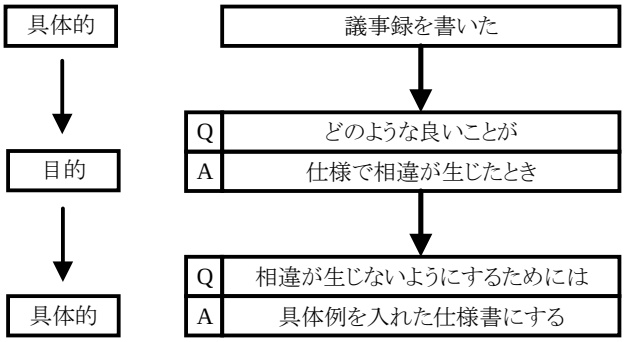


図 5 議事録を書いた

このように，良かったことは，項目としてあげるだけではなく，再利用することが重要だと考える．成功究明は，すでに出来ていることであり，目的究明は新たな問題に気づく可能性があるが，簡単に再利用できるにも関わらず，再利用ができていない．

3.2. Problem

問題は 2 種類に分類できる．ソフトのバグのような技術的問題と，チームワークが悪いというような人や組織の問題である．

問題の原因と結果に注目すると，技術的な問題は，原因と結果が直線的に結びつく，直線的因果律である．人や組織の問題は，原因と結果が悪循環になっているとする円環的因果律がある．

直線的因果律の例として，パソコンから資料が印刷できないという問題（図 6）を取り上げる．この問題では，原因の切り分けを行っていくことになる．第1段階として，パソコンとプリンタで，どちらに原因があるかを分析し，パソコンではなく，プリンタに原因があると判明したとする．次に，プリンタのどこに原因があるかを分析するとインクがなかったと判明する．したがって，インクを交換することにより，この問題は解決できる．

直線的因果律の特徴は，原因と結果に時間遅れがないこと，原因究明を行い，原因を取り除かない限り，問題が解決できないことである．真の原因は何かということが，重要となり，ロジカル・シンキングが有効である．

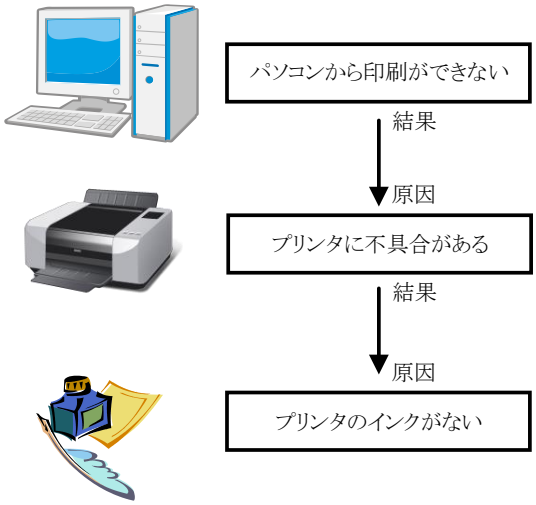


図 6 直線的因果律

円環的因果律の例として，2008 年に起きたリーマン・ショックを契機とした世界的金融危機によって，不況になり商品が売れない問題（図 7）を取り上げる．

この問題は，図 7 のように，アメリカで金融危機による不況が起きると，日本の消費者の消費マインドが冷え込んで，商品を買わなくなる．消費者が商品を買わなくなると，企業では売り上げが下がり，経営者がすぐに行う行動は，研究開発費と研修費の削減となる．研修部門では，研修費を削減されて，社員の研修が出来なくなる．技術部門では，社員の研修が行われなくなるので．やがて技術力が低下して，魅力的な商品の開発ができなくなる．消費者は，魅力的な商品が提供されないので，物を買わなくなるという悪循環になる．

円環的因果律の特徴は，原因と結果に時間遅れがあり，関係者全員が原因であり，結果でもある．原因と結果に時間遅れがあるとは，経営者が研修費を削減して，社員研修を止めても，すぐには，技術部門の技術力低下には，繋がらないということである．関係者全員が原因であり，結果でもあるとは，例えば，図 7 の経営者の視点から原因を追究していくと，消費者，技術部門，研修部門，経営者の順に原因をさかのぼることができ，原因が自分に戻ってくる．同様に，経営者の視点から，結果をさかのぼっていくと，自分に戻ってくる．人や組織の問題で，原因究明を行うと，原因が判らなかったり，犯人捜しになりがちで，問題解決には有効とは限らない．したがって，原因究明しないシステム・シンキングが有効である．

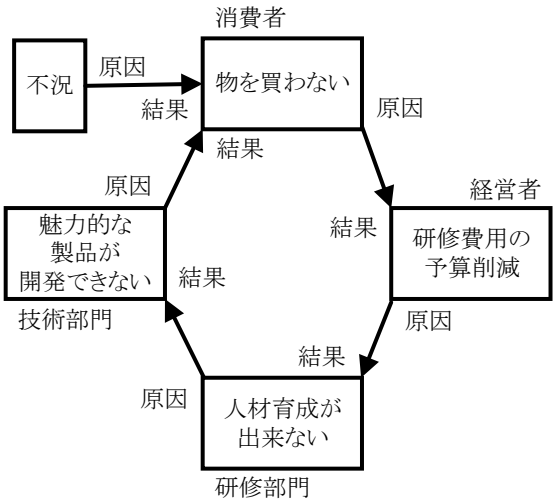


図 7 円環的因果律(その1)

もう1つ円環的因果律の具体例として，OJT の問題(図8)を取り上げる．OJT リーダーである課長が，忙しくて新人の指導ができない．先輩は，指導してもらえない新人を可哀そうに思い，代わりに新人を指導しているため，先輩の本来の仕事が回らなくなっているという問題を取り上げる．

先輩が新人の OJT を指導すればするほど，課長は，新人の OJT の指導をしなくて済んでしまう．課長が OJT の指導をしてくれないので，益々，新人は先輩に頼ることになる．この関係者(課長，新人，先輩)のそれぞれの行動が，問題を維持する要因になっている．

この問題では，それぞれの責任の度合いは異なるが，このように関係者全員(課長，新人，先輩)が原因でもあり，結果でもある．

円環的因果律の問題を解決する方法は，2つある．1つは，悪循環を断つ「MRI アプローチ」であり，もう1つは，この悪循環に注目するのではなく，稀に起こる良い循環に注目する「解決志向アプローチ」である．

MRI アプローチの具体例として，この悪循環を断ち切る方法の1つは，別の課長のもとに，新人を人事異動することである．MRI アプローチを用いる場合は，犯人捜しとならないように，関係者への配慮が必要となる．

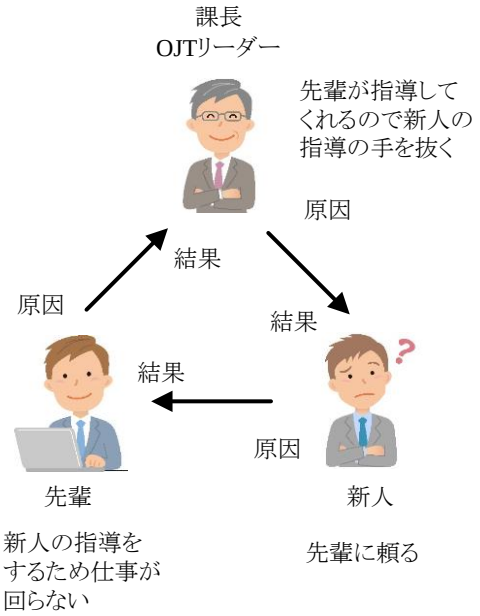


図 8 円環的因果律(その2)

通信プロトコルの OSI 参照モデルを参考に，人や組織の問題解決レイヤーを考えると図 9 のようになる．

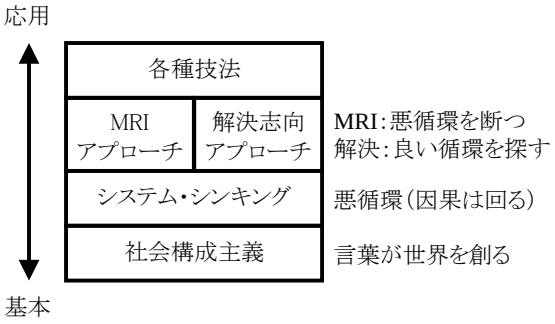


図 9 問題解決レイヤー

3.2.1. Organizational Problem(組織的問題)

人や組織の問題を組織的問題(Organizational Problem)と名付けた．人や組織の問題は，原因と結果が円環的因果律となっていて，原因究明が有効とは限らないことを説明した．システム・シンキングを基本とした解決志向アプローチを用いた問題解決法について，説明する．

解決志向アプローチは，原因究明ではなく，解決究明に焦点をあてる．出来ていないところではなく，出来ているところに焦点をあてる問題解決法である．

解決志向アプローチの問題解決方法の例として，遅刻防止対策を考えてみる．会社に遅刻しないように出社したいと思っているが，よく遅刻してしまう．

原因究明型の問題解決法を問題志向アプローチと呼ぶ．この問題志向では，遅刻した日に焦点をあて，遅刻した原因を究明して，原因を取り除く問題解決法である．出来ていないことを出来るようにするためには，時間も費用も掛かる方法である．

解決志向アプローチでは，遅刻した日ではなく，遅刻しなかった日に焦点をあてる．どのような条件が揃うと，遅刻しないかを明確にすることである．遅刻しない条件が明確になれば，既に出来ていることなので，遅刻の大幅な減少が期待できる．

人や組織の問題を解決する解決志向アプローチの質問技法には，4つの質問技法があり，2種類に分類できる．過去のリソースを引き出す質問と未来を創造する質問である．

過去のリソースを引き出す質問は，

・例外探しの質問

・サバイバル・クエスチョン

である．

未来を創造する質問は，

・スケーリング・クエスチョン

・ミラクル・クエスチョン

である．

例外探しの質問は，問題の中にある稀に上手くいった時に焦点をあて，上手くいった時の対処法を繰り返すことにより，問題の解決を行う．

サバイバル・クエスチョンは，問題を抱えている当事者を勇気づける質問である．

スケーリング・クエスチョンは，目標を10点満点としたときに，現状は何点で，10 点満点にするためには，何が必要かを考える質問である．

ミラクル・クエスチョンは，夜，寝て，朝起きるまでに，奇跡が起こって，目標が達成できていたら，どのような一日を過ごすかを聞く質問である．

4つの質問の中で，例外探しの質問が，人や組織の問題を迅速に解決するためには，一番有効である．

例外探しの質問は，上手く行っていることに注目する質問で，上手く行っていない状況の中でも，上手く行っている事はある．いつも上手く行かないと思っている場合（図10，図11）でも，気づかないだけで，上手く行ったときがある．気づかないのは，人は，問題であることを認識すると，問題が起こった時に関心が集中し，上手くできなかった時は，よく気づくようになる．ところが，稀に，上手く対処できた時があると，問題が起きないので，気づかないことになる．

例外探しの質問手順は，図12のようになる．

・1 回ぐらい上手く行った時は，ありませんか

・1 回毎に，上手く行った事があるか，確認し見つかるか

・上手く行ったことが見つからなかったら，1 回毎に点数化してみて，一番点数の高い時はいつか

・上手く行った時は，何が違ったのか

1 回毎に，上手く行った事がないか，確認する質問では，現在から過去の方向に，確認していくことが重要である．現在から過去の方向に，確認していくことにより，上手くいった時が見つからないときに，記憶が怪しくなったところで，質問をやめることができる．最近の方が，記憶が明確に残っているので，上手く対処するための行動が，明確になる．

例外探しの質問の特徴は，現状把握する時間を最小にして，短時間で問題解決できる可能性がある．解決津志向アプローチでは，いくら問題を詳しく聞いても，問題は解決できない．上手く対処できた時を，丁寧に探すことが重要となる．私の経験では，1 対 1 のコンサルティングでは，現状を詳しく聞くことなく，10分程度で，上手く対処できた時を見つけ，解決できている．

例外探しの質問は，短時間に人や組織の問題を解決できるよい方法であるが，ビジネスの現場で用いる場合には，大きな問題を抱えている．

解決志向アプローチの問題解決法は，原因を究明しない問題解決法である．解決志向アプローチのワークショップをビジネス向けに行っているが，参加者の感想として，「解決志向アプローチの有効性は理解できるが，原因を究明しない問題解決法は，職場では理解されないので使えない」という感想を，いただくことがある．解決志向アプローチをビジネスで用いる場合は，ステークホルダーに解決策を説明する必要がある．日本では，幼いころから社会人にいたるまで，問題を解決するためには，原因を究明し，原因を取り除くことによって，問題は解決できると学んできた．したがって，原因究明しない問題解決法を，受け入れることが難しい．対策として，解決志向アプローチでは，解決策が先に出てくるので，解決策に相応しい原因を後から考えて，ステークホルダーに伝えることで，問題を解決できる（図 13）．解決策に相応しい原因は，多くの場合，解決策の否定を原因とそればよい．注意点として，原因は人ではなく，事にすることである．

当事者の認識(いつもだめ)						
日	月	火	水	木	金	土
×	×	×	×	×	×	×

図 10 当事者の認識

実際(出来ている時がある)						
日	月	火	水	木	金	土
×	×	×	○	×	×	×

図 11 実際

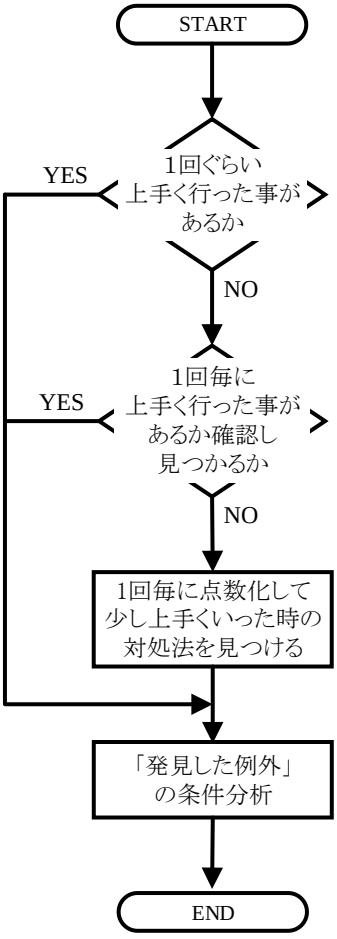


図 12 例外探しの質問手順

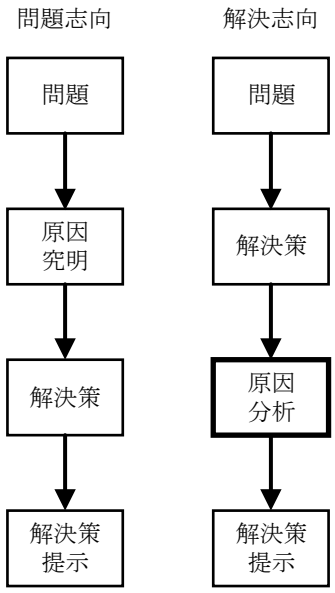


図 13 ビジネス向け解決志向アプローチ

3.2.2. Technical Problem(技術的問題)

技術的問題は, 原因究明が重要となるので, なぜなぜ分析をはじめとするロジカル・シンキングが, 有効である.

3.3. Try

Recycle, Organizational Problem, Technical Problem で行った分析結果から得られた対処方法を記述し, 良かったことも, 悪かったことも, 無駄なく, 次回に活かすことができる. 日本では, 特に, 良かったことを深堀して, 再利用するということが少ないように思う.

4. 結果と考察

アジャイル開発の振り返りのフレームワークである KPT の進化系として, ROTT をビジネス向けワークショップで, 紹介してきた. 参加者から, ワークショップの満足度に関するアンケートで, 高い評価をいただいている.

日本のビジネスは, 弱点克服に偏っていると感じている. 弱点克服と同様に, 強味も活かすことが重要だと考えている. 特に, ソフトウェア業界では, 問題といえば, バグ対処である. バグは, 原因究明が必要な直線的因果律で, 原因を取り除かない限り, 問題は解決できない. そのため,

人や組織の問題においても，真の原因は何かという方向に，なりがちである．失敗から学ぶと同様に，成功からも学ぶことが重要だと考える．

今回，問題を 2 種類に分類したが，両方の問題解決法が必要となるようなバグもある．例えば，ウォーターフォールモデルのシステムテストで，発生したバグの原因分析をした結果，上流工程のシステム要件定義に，問題があることが判明した．システム要件定義からの対応となり，バグ修正の工数が大きくなり，開発体制の強化が必要となった場合である．ステークホルダーに働きかけて，組織を動かすことが必要となる．この場合，原因究明型のロジカル・シンキングでバグ分析を行い，次に，組織を動かすために，システム・シンキングを基本とする解決志向アプローチを用いると効果的である．

解決志向アプローチは，人や組織の問題であれば，振り返り以外にも，幅広く使うことができる．問題を起こしている関係者自身が，解決志向アプローチを使って問題解決することが，容易ではない場合がある．問題の関係者は，問題の悪循環に取り込まれているので，客観的に悪循環を分析して，悪循環から抜け出す事は容易ではない．第3者である上司，同僚，コンサルタントからの，解決志向アプローチを用いたサポートが効果的である．

人をまとめることが仕事であるリーダーやマネジャーは，システム・シンキングは必須のスキルだと考える．ところが，日本では，ロジカル・シンキングと比較するとシステム・シンキングは，あまり知られていない．google 検索エンジンを使って，2018 年 4 月 18 日現在の知名度を比較した．システム・シンキングは，システム思考と呼ばれることも多いため，「システムシンキングとシステム思考」，「ロジカルシンキングと論理思考」で検索した．検索結果は，ロジカル・シンキングが，463,000 件となり，システム・シンキングが，94,200 件となった．百分率で表すと，ロジカル・シンキングは，83.1%，システム・シンキングは，16.9%となった．

日本のビジネスに，ロジカル・シンキングと同様に，システム・シンキングや解決志向アプローチが，広まることを願っている．

参考文献

[1] 天野勝，これだけ! KPT，すばる舎
[2] 中原淳，長岡健,ダイアローグ 対話する組織，ダイヤモンド社
[3] 西村行功，システム・シンキング入門 (日経文庫) ，日本経済新聞社
[4] 森俊夫，黒沢幸子，森・黒沢のワークショップで学ぶ解決志向ブリーフセラピー，ほんの森出版

結合・総合テストフェーズにおける継続的テスト設計の取り組み

山口　真
SCSK 株式会社
Makoto.Yamaguchi@scsk.jp

豊田　圭一郎
SCSK 株式会社
Keiichirou.Toyoda@scsk.jp

田辺　紘明
SCSK 株式会社
Hi.Tanabe@scsk.jp

要旨

1. はじめに

システム開発プロジェクトにおける有識者のセンサー・感覚、そしてプロジェクトのサブシステム担当者（以下、サブシステム担当者）の懸念・疑念という 2 つの「暗黙知」を短いサイクルで抽出し、テストケースという「形式知」に昇華させ、効果的・効率的かつ継続的にテスト品質を高める取り組みを紹介する.

2. 背景・課題

結合・総合テストフェーズにおいて、従来の記述式(スクリプト)テストと合わせて、業務・システム有識者の知見・経験による探索的テスト[1][2]がプロジェクトの品質を左右している現状がある.

しかし、近年プロジェクトの短期間化やプロジェクト並行度・複雑度の高まりが進んでおり、その状況下において下記 3 つの課題が顕在化し、品質確保が困難となっていった.

① プロジェクトにアサインできる有識者の不足
② 有識者の知見の共有やノウハウの蓄積がされない
③ サブシステム担当者の懸念・疑念が解決されないままプロジェクトが進み結合・総合テストフェーズに突入する

上記課題に対し、いかに効果的・効率的に結合・総合テストを実施し品質を確保するかが求められてきている.

3. 事例概要

3.1 対応した課題

顕在化した課題①は有識者を育成し増やすことにより解決するが、長期間を要することになる.そのため、比較的短期に行える課題②③にフォーカスし、暗黙知を引き出すため「モヤモヤ会」と称する会議体を設定した.

3.2 「モヤモヤ会」の概要・ポイント

モヤモヤ会は有識者・サブシステム担当者の頭の中にある言語化されていない暗黙知の抽出（表出化）を行い、それらをテストケース化(形式知化)する、「チームで行う継続的なテスト設計」である.モヤモヤ会のポイントは以下の通りである.

・短時間(30 分/回)で毎日行う短期反復型
・どのような意見・アイデアをも否定・批判しないブレイン・ストーミング（意見・アイデアは出すだけで称賛）
・誰が・どのように実施するかは後々決める（抽出を重視）
・一つの意見・アイデアに対し他メンバーにも感想・意見を求めアイデアの相乗効果・チーム意識を促す

3.3 「モヤモヤ会」の成果

(1)開発規模あたりの UAT 不具合数・本番障害数の減少

結合・総合テストフェーズでの不具合検出率に対し、ユーザー受け入れテスト(UAT)や本番稼働後に発覚する不具合率は「モヤモヤ会」導入前のプロジェクトと比較して 90%減となった.

意見・アイデアそのものをプロジェクトとして明確にテスト化しただけでなく、意見・アイデア同士が新たなテスト観点の創出にも寄与していた.

(2)属人的なテストからの脱却、有識者のテスト観点の蓄積

テストケース化(形式化)することで有識者のみが行えていたテストを他メンバーにて代替して実行を可能にすることで、有識者不足で「実行不可能」とあきらめていたテストを「実行可能」な状態にすることができた.また、形式化されたケースは再利用可能な資産となり、後発のプロジェクトで活かせるものとなった.

4. 今後の課題

暗黙知の具体化・言語化するにあたり、意見の要約、意見を促すといったファシリテーション能力が求められるが、現状ファシリテーションを行えるメンバーも限られている.今後ファシリテーションを行えるメンバーを増やし、テストフェーズに限らないプロジェクトの随所でモヤモヤを解決する自発的な動きに繋がればと考える.

参考文献

[1] James Bach, General Functionality and Stability Test Procedure, http://www.satisfice.com/tools/procedure.pdf
[2] 高橋 寿一, 知識ゼロから学ぶソフトウェアテスト, 翔泳社

バグ修正時間を考慮したソフトウェア最適リリース問題についての一考察

岡村 寛之
広島大学大学院工学研究科
okamu @ hiroshima-u.ac.jp

住田 大亮
広島大学大学院工学研究科

土肥 正
広島大学大学院工学研究科
dohi @ hiroshima-u.ac.jp

要旨

本稿では，バグ修正難易度を表す指標としてバグ修正時間を用いたソフトウェア最適リリース問題の再考を行う．特に，従来のバグ修正時間を考慮したソフトウェア信頼性モデルにおいて，バグ発見時刻と修正時間が独立であるという仮定を修正し，バグ発見時刻と修正時間の相関を表現することができる一般的なモデルフレームワークを取り上げ，そのモデルフレームワークにおけるソフトウェア最適リリース問題について考察する．

1. はじめに

ソフトウェア信頼性はソフトウェア品質における最も重要な指標の一つである．定量的なソフトウェア信頼性はある期間にバグが発見されない（システム障害が発生しない）確率として定義される．これまでに，定量的なソフトウェア信頼性を評価するための確率モデルとしてソフトウェア信頼性モデル（ソフトウェア信頼度成長モデル）が提案され，ソフトウェア開発管理において様々な側面で利用されている [6, 4, 5, 9]．

従来のソフトウェア信頼性モデルではテスト工程において発見されるバグ数に着目したモデリングとなっている．一方，現在のソフトウェア開発ではオープンソースソフトウェア（OSS: Open Source Software）で見られるように，継続的インテグレーションにより常にソフトウェアが利用できる状態を保持するスタイルが主流となっている．そのような開発形態では，各リリースでそこまでに発見したバグがすべて修正されているとは限らない．つまり，バグ発見だけではなく，バグ発見，バグ特定，バグ修正などのバグに対するライフサイクルを考慮した信頼性評価が必要となる．

バグ修正を考慮したソフトウェア信頼性モデルは，これまでにもいくつも議論されている．Schneidewind [10, 11], Xie ら [13, 12], Gokhale ら [1, 2] は累積修正バグ数を表現する確率モデルを構築している．Xie ら [12] はバグ発見とバグ修正を統合するモデルの構築をしている．これら，累積バグ発見と累積バグ修正がともに非同次ポアソン過程に従うという特徴をもつ．また，Okamura and Dohi [7] は上述のすべてのモデルを含むバグ修正モデルフレームワークを提案している．特に，文献 [7] では，バグ発見時刻とバグ修正時間の相関も考慮した一般的なモデルフレームワークを提案している．ここで言うバグ発見時刻と修正時間時間の相関は，テストの早い段階で見つかったバグほど修正時間が短い（あるいは長い），または，テスト終盤で見つかるバグほど修正時間が短い（あるいは長い）と言った関係を定量的に表しており，これは，どのモジュールを先にテストするかなどのテスト戦略と密接に関係している．文献 [7] では，現在までに見つかっていないバグ数だけではなく，現在までに見つかっているが修正されていないバグ数を考慮した信頼性指標の提案を行っている．

本稿では，上述のバグ発見時刻とバグ修正時間の相関を考慮したモデル上でのソフトウェアリリース問題を扱う．ソフトウェア開発において，経済的な観点からソフトウェアテストを終了しソフトウェアのリリースを行う適切なタイミングを決定することは非常に重要である．これはソフトウェアリリース問題と呼ばれ，古くからソフトウェア信頼性モデルに基づいた数理的なアプローチが数多く行われている．Okumoto and Goel [8] はバグ発見時刻のみを考慮した指数形ソフトウェア信頼性モデルを利用して，テスト費用，テスト期間中のバグ修正費用，リリース後のバグ修正費用の期待値を算出し，その総費用を最小にする最適なリリース時間の決定を行って

いる．Okumoto and Goel [8] 以降，様々な要因を取り入れたリリース問題の定式化がなされているが，そのほとんどがバグ発見時刻だけに着目したものである．つまり，バグ発見後，即座にバグが修正される仮定を考えている．しかしながら，現実のソフトウェア開発にでは様々なバグが存在し，その修正難易度も大きく異なる．一般的にバグ修正では，バグの原因の特定，プログラムの修正，修正箇所のテスト（回帰テスト）が行われ，再現性の低いバグでは原因特定に時間を要することがある．バグ発見だけに着目したソフトウェアリリース問題ではこのようなバグの修正難易度にかかわらず一律のバグ修正費用を課すことが多く，バグ修正がリリース時間に与える影響について明らかになっていない．そこで，本稿では，バグ修正時間を考慮したモデルにおけるソフトウェア最適リリース問題を再考する．特に，一般的なバグ修正時間を考慮したモデルフレームワーク [7] におけるソフトウェアリリース問題を再定義し，バグ修正時間ならびにバグ発見時刻とバグ修正時間の相関が最適なリリース時刻に与える影響について調べ，経済的な観点からどのようなテスト戦略が良いかについて議論する．

2. ソフトウェア信頼性モデル

2.1. バグ発見時刻によるソフトウェア信頼性モデル

従来のソフトウェア信頼性モデル（ソフトウェア信頼度成長モデル）は，開発工程における累積バグ発見数を確率過程で表現し，現時点での残存バグ数や将来のバグ発見に関する評価・予測を行うモデルである．非同次ポアソン過程（NHPP: non-homogeneous Poisson process）に基づくソフトウェア信頼性モデルは以下の仮定から構築される．

- テスト前にソフトウェア内に潜在するバグ数 M は，平均 ω のポアソン分布に従う．
- M 個のバグの発見時刻は独立かつ同一に分布する確率変数 $\{X_1, \cdots, X_M\}$ で表され，その累積分布関数を $F(t)$ とする．

いま，時刻 $t = 0$ でテストを開始したとき，時刻 t までの累積バグ発見数を $\{N(t), t \geq 0\}$ とすると，上記の仮定から以下の確率関数が得られる．

$$\begin{aligned} P(N(t) = n) &= \sum_{m=n}^{\infty} P(N(t) = n | M = m) P(M = m) \\ &= \sum_{m=n}^{\infty} \binom{m}{n} F(t)^n (1 - F(t))^{m-n} \frac{\omega^m}{m!} e^{-\omega} \\ &= \frac{(\omega F(t))^n}{n!} e^{-\omega F(t)}. \end{aligned} \tag{1}$$

上記の確率関数より，累積バグ数の確率過程 $N(t)$ は平均値関数 $E[N(t)] = \omega F(t)$ の NHPP となる．ソフトウェア信頼性モデルが与えられた時，テスト時刻 t から $t+u$ でバグが発見される確率（ソフトウェア信頼度）は

$$R(u|t) = \exp\left(-\omega(F(t+u) - F(t))\right) \tag{2}$$

で与えられる．

2.2. バグ発見時刻と修正時間を考慮したモデル

文献 [7] では，次で紹介するバグ発見だけではなく修正時間を考慮したモデルを提案している．まず，修正時間を扱うために以下の仮定を設定する．

- テスト前にソフトウェア内に潜在するバグ数 M は，平均 ω のポアソン分布に従う．
- M 個のバグの発見時刻 X および修正時間（発見から修正完了までの時間）S は独立かつ同一に分布する二変量確率変数 $\{(X_1, S_1) \cdots, (X_M, S_M)\}$ で表され，その同時分布関数を $F(x, s)$ とする．

先のバグ発見時刻に対するモデルとの本質的な違いは，バグ発見時刻と修正時間が二変量分布で表現されている点にある．つまり，バグ修正時間がバグが発見される時刻に依存することを許容している．これは，テストの早い段階で見つかったバグほど修正時間が短い（あるいは長い），または，テスト終盤で見つかるバグほど修正時間が短い（あるいは長い）と言った相関関係を表すことになる．

文献 [7] では二変量分布 $F(x, s)$ から次の分布を考えている．

- $F_{UC}(t)$: あるバグが時刻 t までにバグが発見されたが修正されていない確率
- $F_C(t)$: あるバグが時間 t までにバグが発見・修正される確率

- $F_D(t) = F_{UC}(t) + F_C(t)$: あるバグが時間 t までに発見される確率

これらは同時密度関数 $f(x,s) = \partial^2 F(x,s)/\partial x \partial s$ を用いると

$$F_{UC}(t) = \int_0^t \int_0^{t-x} f(x,s) ds dx, \tag{3}$$

$$F_C(t) = \int_0^t \int_{t-x}^{\infty} f(x,s) ds dx, \tag{4}$$

$$F_D(t) = \int_0^t \int_0^{\infty} f(x,s) ds dx \tag{5}$$

となる．いま，$N_D(t)$ ならびに $N_C(t)$ を時刻 t までに発見されたバグの総数ならびに修正されたバグの総数とすると，初期バグ数 M がポアソン分布に従うため $N_D(t)$ と $N_C(t)$ の同時確率は

$$\begin{aligned} &P(N_D(t) = d, N_C(t) = c) \\ &= \frac{(\omega F_{UC}(t))^{d-c} (\omega F_C(t))^c}{(d-c)!c!} e^{-\omega F_D(t)} \end{aligned} \tag{6}$$

として記述できる．上述の周辺分布 $P(N_D(t) = d)$, $P(N_C(t) = c)$ はともに平均 $\mathrm{E}[N_D(t)] = \omega F_D(t)$, $\mathrm{E}[N_C(t)] = \omega F_C(t)$ のポアソン分布となる．つまり，文献 [7] のモデルフレームワークはバグ発見過程，バグ修正過程がともに NHPP となるモデルをすべて包括している．実際，既存の文献 [10, 11, 13, 12] のモデルがこのモデルフレームワークのサブクラスとなっている．

文献 [7] では，ソフトウェア信頼度（ある区間でバグが見つかる確率）以外の信頼性尺度として，ある時刻までに発見されたバグがすべて修正されている確率を算出している．これは，例えば OSS を流用したソフトウェアを開発する場合に従来のソフトウェア信頼度よりも有益な尺度となることが示唆されている．また，文献 [7] ではさらに，効率的なモデルパラメータの推定アルゴリズムについて言及している．

3. ソフトウェア最適リリース問題

3.1. 従来のソフトウェア最適リリース問題

Okumoto and Goel [8] は次に示すテスト費用を考慮したソフトウェア最適リリース問題を定義している．

- c_1： テスト工程における単位時間当たりのテスト費用
- c_2： テスト工程において発見されたバグを修正するための単位個数当たりの費用
- c_3： リリース後の運用工程において発見されたバグを修正するための単位個数当たりの費用

一般的に，バグ修正費用はテスト工程よりもリリース後の運用工程の方が高いため $c_3 > c_2$ を仮定する．現在時刻 t までに発見されたバグの個数を $N(t)$ とし，ソフトウェアのサポート打ち切り時刻を T_{LC} とする．このとき，時刻 T でソフトウェアをリリースしたときの総期待費用を $C(T)$ とすると，

$$C(T) = c_1 T + c_2 E[N(T)] + c_3 (E[N(T_{LC}) - E[N(T)]). \tag{7}$$

となる． $E[N(t)]$ は時刻 t までに発見されたバグ総数の期待値であり，バグ発見時刻分布 $F(t)$ および初期バグ数の期待値 ω を用いると，式 (7) は

$$C(T) = c_1 T + c_2 \omega F(T) + c_3 \omega \{F(T_{LC}) - F_D(T)\}. \tag{8}$$

となる．このとき，ソフトウェア最適リリース問題は総期待費用 $C(T)$ を最小化するリリース時刻 T^* を求める問題として，次のように定式化される．

$$\min_{0 \le T \le T_{LC}} C(T). \tag{9}$$

3.2. バグ修正時間を考慮したソフトウェア最適リリース問題

本稿では，2.2 で紹介したバグ修正時間を考慮したモデルに基づいてソフトウェア最適リリース問題を考える．いま，Okumoto and Goel [8] と同様に以下の費用を仮定する．ただし，テスト工程ならびに運用段階におけるバグ修正費用が単一バグ当たりの費用ではなく，修正時間に比例したコストに変更している．

- c_1: テスト工程に要する単位時間当たりの費用.
- c_2': テスト工程において発見されたバグを修正するために必要な単位時間当たりの費用.
- c_3': 運用段階において発見されたバグを修正するために必要な単位時間当たりの費用.

いま，ソフトウェアの利用期間を T_{LC} とすると，時刻 T でソフトウェアをリリースするときの総期待費用 $C(T)$ は以下のように定式化される．

$$C(T) = c_1 T + c_2' W(T) + c_3' \left(W(T_L) - W(T)\right). \tag{10}$$

ここで，$W(T)$ は時刻 T までに発見されたすべてのバグを修正するのに必要な期待累積修正時間を表す．いま，$\{(X_1, S_1) \ldots, (X_{N_D(T)}, S_{N_D(T)})\}$ を時刻 T までに発見されたバグの発見時刻と修正時間の列とすると，時刻 T までに発見された $N_D(T)$ 個のバグに対する修正時間 $S_1, \ldots, S_{N_D(T)}$ の和となるため

$$
\begin{aligned}
W(T) &= \mathrm{E}\left[\sum_{i=1}^{N_D(T)} S_i\right] \\
&= \sum_{k=0}^{\infty} P(N_D(T) = k) k \int_0^T \mathrm{E}[S|X = x] f_D(x) dx \\
&= \omega \int_0^T \int_0^{\infty} s f(x, s) ds dx. \qquad (11)
\end{aligned}
$$

となる．ここで $f_D(x)$ はバグ発見時刻に対する周辺密度関数であり，さらに，

$$
\mathrm{E}[S|X = x] = \frac{\int_0^{\infty} s f(x, s) ds}{f_D(x)} \qquad (12)
$$

である．

特別な場合として，バグ発見時刻 X とバグ修正時間 S が独立な場合，つまり，テストの初期と終盤で発見されたバグの修正時間が同じ場合は

$$
W(T) = \mathrm{E}[N_D(T)]\mathrm{E}[S] = \omega F_D(T)\mathrm{E}[S] \qquad (13)
$$

となる．つまり，単一のバグ修正費用を $c_2'\mathrm{E}[S]$ ならびに $c_3'\mathrm{E}[S]$ とした Okumoto and Goel [8] によるソフトウェアリリース問題に帰着される．換言すると，Okumoto and Goel [8] によるリリース問題の定式化では「テストの初期と終盤で発見されたバグの修正時間が同じ」であることを暗に仮定している．また，ここで提案した上記の定式化は Okumoto and Goel [8] による問題の数理的な一般化となっている．

以上の準備のもと，バグ修正時間を考慮したソフトウェア最適リリース問題は総期待費用 $C(T)$ を最小化する時刻 T を見つける問題として定式化される．

$$
\min_{0 \le T \le T_{LC}} C(T). \qquad (14)
$$

4. 数値例

ここでは，バグ発見時刻と修正時間を相関を表現するために以下に示す FGM コピュラ [3] を用いる．

$$
F(x, s) = F_D(x) F_C(x) \left(1 + \alpha \overline{F}_D(x) \overline{F}_C(s)\right). \qquad (15)
$$

表 1. 最適リリース時刻と最小総期待費用.

α	T^*	$C(T^*)$
1.0	495.4	1095.6
0.5	477.3	1077.4
0.0	455.3	1055.3
-0.5	427.5	1027.0
-1.0	389.9	988.0

ここで，$F_D(x) = P(X \le x)$, $F_C(x) = P(S \le s)$ はバグ発見時刻ならびに修正時間の周辺分布関数，$F(x, s) = P(X \le x, S \le s)$ は同時分布関数を表す．また，$\overline{F}_D(t) = 1 - F_D(t)$, $\overline{F}_C(t) = 1 - F_C(t)$ である．さらに $-1 \le \alpha \le 1$ は相関の強さを表すパラメータであり，$\alpha < 0$ の時は負の相関（早い段階で見つかったバグほど修正時間が長い），$\alpha = 0$ の時は無相関（テストの初期と終盤で発見されたバグの修正時間が同じ），$\alpha > 0$ の時は正の相関（早い段階で見つかったバグほど修正時間が短い）となる．FGM コピュラによりバグ発見時刻と修正時間の相関を表現するとき期待累積修正時間は

$$
\begin{aligned}
W(T) &= \omega F_D(T)\mathrm{E}[S] \\
&\quad \times \left(1 - \frac{\alpha \overline{F}_D(T)}{\mathrm{E}[S]} \int_0^{\infty} F_C(s) \overline{F}_C(s) ds\right) \qquad (16)
\end{aligned}
$$

となる．さらに，簡単のためバグ発見時刻と修正時間をともにパラメータ β_D, β_C の指数分布とすると

$$
W(T) = \frac{\omega}{\beta_C}(1 - e^{-\beta_D T}) \left(1 - \frac{\alpha}{2} e^{-\beta_D T}\right) \qquad (17)
$$

となる．

表 1 は $c_1 = 1.0$, $c_2' = 0.01$, $c_3' = 0.2$, $\omega = 100.0$, $\beta_d = 0.01$, $\beta_c = 0.002$, $T_{LC} = 720.0$ とし，パラメータ α を -1 から 1 まで変化させた時の最適リリース時刻 T^* と最小総期待費用 $C(T^*)$ を示している．パラメータ c_1, c_2', c_3' の設定では基本的なテスト費用 c_1 に対する比率が重要であるため，$c_1 = 1$ と設定し，c_2', c_3' は基本的なテスト費用の 1%, 20% として相対的に設定している．

この表からバグ発見時刻と修正時間に正の相関がある場合，独立な場合と比較して最適なリリース時刻が長くなり総費用も高くなることがわかる．反対に負の相関がある場合，最適リリース時刻が短くなり総費用が低くなる．この結果は，バグ発見時刻と修正時間に正の相関がある場合，テスト終盤で修正時間の長いバグが発見され

る傾向があるため，リリース後のバグ修正費用を抑えるためにリリースの延長を考慮する必要があることを示唆している．また，正の相関がある場合はリリースを延長したとしても全体の費用を抑えることにはつながらないこともわかる．つまり，全体のテスト関係の費用を抑えるためにはテスト初期において修正に時間のかかるバグを発見する方が良いと言うこともわかる．ここでは，c_1, c_2', c_3' に相対的な費用を設定しているため，$\alpha = 1$ と $\alpha = -1$ の場合の費用の差は 1095.6 - 988.0 = 107.6 であり，全体費用の 10 % 程度ではあるが，プロジェクトの規模が大きければなるほど基本的なテスト費用として設定した単位時間あたりのテスト費用（1 日当たりのテスト費用） c_1 が大きくなるため，負の相関を示すようなテスト戦略を行うことによって，絶対的な費用としてはかなり多くの費用が低減できることもわかる．

5. まとめと今後の課題

本稿では，バグ修正時間を考慮したソフトウェア信頼性モデルの一般的なモデルフレームワークを利用し，ソフトウェアリリース問題の定式化ならびに，バグ発見時刻とバグ修正時間の相関が費用とリリースに与える影響について分析した．文献 [7] ではいくつかのオープンソースプロジェクトのデータからバグ発見時刻とバグ修正時間における相関を分析し，いくつかのプロジェクトで（強い相関ではないが）相関があることを確認している．

本稿の数値例における結果では，テスト初期において修正時間のかかるバグを発見することが経済的な観点から重要であることがわかった．また，リリースに対する意思決定においてはバグ発見時刻とバグ修正時間の傾向を見ることも重要であることがあることも示された．この指標は恐らくこれまでリリースに対して，あまり意識されていない指標であると考えられる．

今後は，ソフトウェア品質評価やテスト進捗などから得られる指標に加えて，ここで議論したバグ発見時刻とバグ修正時間の相関も考慮したリリース判定に対する意思決定スキームについて議論する予定である．

参考文献

[1] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi. Analysis of software fault removal policies using a non-homogeneous continuous time markov chain. *Software Quality Journal*, Vol. 12, No. 3, pp. 211–230, 2004.

[2] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi. Incorporating fault debugging activities into software reliability models: A simulation approach. *IEEE Transactions on Reliability*, Vol. 55, pp. 281–292, 2006.

[3] H. Joe. *Dependence Modeling with Copulas*. CRC Press, 2014.

[4] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1996.

[5] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, 1999.

[6] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability, Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[7] H. Okamura and T. Dohi. A generalized bivariate modeling framework of fault detection and correction processes. In *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE 2017)*, pp. 35–45. IEEE CPS, 2017.

[8] K. Okumoto and L. Goel. Optimum release time for software systems based on reliability and cost criteria. *Journal of Systems and Software*, Vol. 1, pp. 315–318, 1980.

[9] H. Pham. *Software Reliability*. Springer, Singapore, 2000.

[10] N. F. Schneidewind. Analysis of error processes in computer software. *Proceedings of the International Conference on Reliable Software*, Vol. 10, pp. 337–346, 1975.

[11] N. F. Schneidewind. Modelling the fault correction process. In *Proceedings of The 12th International Symposium on Software Reliability Engineering*, pp. 185–190. IEEE Computer Society Press, 2001.

[12] M. Xie, Q. P. Hu, Y. P. Wu, and S. H. Ng. A study of the modeling and analysis of software fault-detection and fault-correction processes. *Quality and Reliability Engineering International*, Vol. 23, pp. 459–470, 2007.

[13] M. Xie and M. Zhao. The schneidewind software reliability model revisited. In *Proceedngs of the 3rd International Symposium on Software Reliability Engineering*, pp. 184–192. IEEE Computer Society Press, 1992.

トピックモデリングに基づく開発者検索手法の構築へ向けて

福井 克法
和歌山大学 システム工学部
fukui.katsunori@g.wakayama-u.jp

大平 雅雄
和歌山大学 システム工学部
masao@sys.wakayama-u.ac.jp

川辺 義勝
株式会社 SRA
kawabe@sra.co.jp

要旨

効率的なソフトウェア開発には適切な人材配置が重要となるため，ソフトウェア開発組織は各開発者がこれまで携わった業務に関する経験をできる限り詳細に把握し管理できることが望ましい．しかしながら，組織の規模がある程度大きくなると在籍するすべての開発者の業務経験を人為的に管理することは現実的に困難な問題となる．本研究の目的は，ソフトウェア開発の全工程を対象として開発者を広く検索できる手法を構築し効率的な人材配置を支援することである．特に本論文では，メーリングリストアーカイブへ潜在的ディリクレ配分法 *(LDA: Latent Dirichlet Allocation)* を適用した結果に基づいて，検索語と開発者の関係性を定量的に計測することにより開発者を検索する手法を提案する．*4* つの *OSS* プロジェクトのメーリングリストアーカイブを対象として提案手法の評価実験を行った結果，プロジェクトによってばらつきが見られるものの，提案手法により *61%*〜*100%*の精度で開発者を検索できることが分かった．

1. はじめに

効率的なソフトウェア開発には適切な人材配置が重要となるため，ソフトウェア開発組織は各開発者[1]がこれまで携わった業務に関する経験をできる限り詳細に把握し管理できることが望ましい．しかしながら，組織の規模がある程度大きくなると在籍するすべての開発者の業務経験を人為的に管理することは現実的に困難な問題となる．そのため，構成管理システムに記録された開発者の過去の活動履歴に基づいて開発者を検索する手法が研究されている．例えば，Mockus らは，ソースファイルの変更履歴に基づいてソフトウェアシステムのモジュール毎に開発者の専門性を計測する手法を提案しており，問い合わせ先として適任の開発者を検索できるよう支援している [1]．同様に，Robbes らは，ソースファイルの変更履歴に加えて変更時期（直近の変更であれば専門性を高くする）を考慮した開発者検索手法を提案している [2]．

先行研究は，多数の開発者が在籍する大規模組織における開発者検索手法としてはある程度の有用性を期待できるものの，構成管理システムに記録される活動と紐付く開発者，すなわち，製造工程や保守工程において構成管理の対象となるファイルを変更したことのある開発者のみしか検索の対象にできないという点で適用範囲に限界がある．実際の開発組織においては，要件定義や基本・詳細設計などの上流工程を主に担当する場合もあり，業務の中でソースファイルの変更に直接的に関与しない開発者も少なくない．したがって先行研究は，大規模な開発組織内に在籍する開発者を広く検索する手法としては実用性の観点で問題があると言える．

本研究の目的は，ソフトウェア開発の全工程を対象として開発者を広く検索できる手法を構築し効率的な人材配置を支援することを最終的な目的とする．特に本論文では，メーリングリストアーカイブへ潜在的ディリクレ配分法 (LDA: Latent Dirichlet Allocation) を適用した結果に基づいて，検索語と開発者の関係性を定量的に計測することにより開発者を検索する手法を提案する．

本論文の構成は以下の通りである．続く 2 章では，提案手法の詳細について記述する．3 章では，提案手法を評価するための実験について述べる．4 章で評価実験の結果を示し，5 章において結果を考察する．6 章で関連研究を紹介し，最後に 7 章において，まとめと今後の課題について述べ本論文を結ぶ．

[1]本研究における開発者とは，ソフトウェア開発工程に関与するすべての実務者を指す．

2. 提案手法

2.1. 概要

本研究の目的は，ソフトウェア開発の全工程を対象として開発者を広く検索できる手法を構築し効率的な人材配置を支援することである．特に本論文では，メーリングリストアーカイブへ潜在的ディリクレ配分法 (LDA: Latent Dirichlet Allocation) を適用した結果に基づいて，検索語と開発者の関係性を定量的に計測することにより開発者を検索する手法を提案する．メーリングリストアーカイブを用いる理由は，開発工程の違いや部門の違いを問わずメーリングリスト[2]が開発組織において現状最も広く利用されているコミュニケーションツールであるため，開発者を広く検索するためのデータソースとして適していると考えたためである．

メーリングリストアーカイブを用いた最も単純な開発者検索手法には，メーリングリスト内で各開発者が使用した単語の回数に基づいて検索語毎に開発者を順位付けする方法が考えられる．しかしながら，このような単純な方法では，開発者が特定の投稿で多数同じ単語を使用した場合や，様々な分脈や話題で広く利用される一般的な単語を使用した場合には，開発者の専門性をその単語から想起することが困難になる．検索語となる単語がメーリングリスト内のどのような議論で特徴的に利用されてきたのかについての意味的な分析を踏まえて，検索語と開発者の関連性が定量化される必要がある．そこで本研究では，メーリングリストに存在するトピック（潜在的な意味）を把握する手段として潜在的ディリクレ配分法（LDA : Latent Dirichlet Allocation）[3] を用いることとした．

提案手法は主に，(1) メーリングリストアーカイブに LDA を適用しトピックを抽出する処理，(2) メーリングリストへの各投稿と検索語との関係性を定量化するための処理，(3) 検索語との関係性が高い投稿を行った開発者を順位付けするための処理から成る．以降では，それぞれの処理を詳しく説明する．

2.2. LDA によるトピック抽出

本研究では，メーリングリスト全体に含まれるトピックと各投稿を構成するトピックを把握するために LDA を用いる．図 1 は，メーリングリストアーカイブへ LDA を適用する一連の流れを示したものである．本節では，LDA のアルゴリズムを簡潔に説明するとともに，メーリングリストアーカイブを LDA に適用する際の前処理について説明する．

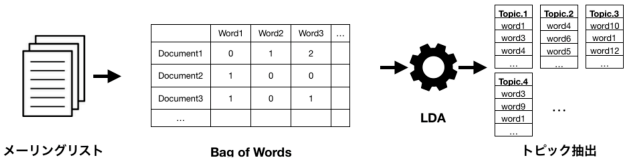


図 1. メーリングリストアーカイブへの LDA の適用

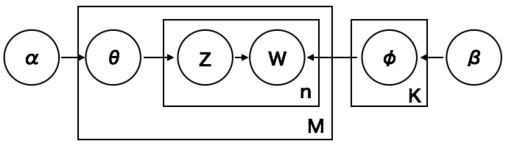


図 2. LDA のグラフィカルモデル

2.2.1. 潜在的ディリクレ配分法（LDA ： Latent Dirichlet Allocation）

LDA は文章の生成過程を確率的にモデル化するトピックモデルの 1 つである．トピックモデルとは，文章集合に含まれる潜在的なトピック（潜在的な意味）を推定するモデルである．LDA を用いることで，文章集合に含まれる複数のトピックを推定できるだけではなく，各トピックを構成する単語のトピック内での生起確率である**単語分布**や，1 つの文書に含まれるトピックの構成割合である**トピック分布**を計算することができる．

図 2 に，潜在的ディリクレ配分法のグラフィカルモデルを示す．LDA では 1 つの文書は複数のトピックから構成されており，各トピックで生起する単語の生起確率に応じて文書が生成されるという考えに基づいたモデルであるため，各文書はトピックの構成比であるトピック分布 θ を持っており，θ に応じて生起する単語のトピック Z が決定する．トピック Z の単語分布より生起する単語 W が決定する．n は 1 つの文書上の単語の出現回数を表し，M は文書数，K はトピック数を表す．α は θ を決定するための，β は ϕ を決定するためのハイパーパラメータである．

[2]個人間のやり取りを含むメールは本研究では対象外とする．

2.2.2. メーリングリストアーカイブへの LDA の適用

メーリングリストアーカイブに LDA を適用するためには，メーリングリストアーカイブから抽出したテキストデータ（本文）を数値で扱う必要がある．本研究では，図 1 左部分に示したように，Bag-of-Words を用いてテキストデータをベクトル化する．具体的には，メーリングリストへの各投稿に含まれるテキストデータ（本文）を Document1，Document2 のように行として，投稿中に出現する単語を Word1，Word2 のように列とし，各投稿に含まれる単語の頻度を行列で表したものが本研究で扱う Bag-of-Words ベクトルである．本研究では，30% 以上の文書に出現する一般的な単語（頻繁に使われる特徴的ではない単語）を除外してから Bag-of-Words ベクトルを作成している．また，文書全体で 2 回以下しか出現しないような単語も専門性を表す際のノイズとなる可能性が高いためあらかじめ除去した．

2.3. 検索語と各投稿との関係性の定量化

本研究では，開発者の人材配置を意図して大規模な開発組織に所属する開発者を検索する際には，開発者の業務経験や技術スキルを表す専門的な用語が検索語として用いられると想定している．本研究ではまず，LDA を適用した際に算出されるトピック分布 θ および単語分布 ϕ を用いて検索語と各投稿の関係を定量化し，2.4 節で述べる開発者のスコアリングのための前処理を行う．

まず，LDA の適用により抽出されるトピックから検索語と関連性の高い上位 5 つのトピックを抽出する．図 3 に示すように，具体的には各トピックの単語分布から検索語の生起確率が高い順に 5 つのトピックを抽出する．この処理によって，検索語とは一致しなくても検索語を含むトピックに含まれる単語を用いた開発者を検索する手がかりとなる．すなわち，検索語を含むトピックを何らかの専門性を表す文脈であると仮定し，その文脈に多く現れる開発者を検索語に関連性の深いエキスパートと考え検索結果の上位に位置付けるための前処理である．

次に，検索語と関連性の高い上位 5 つのトピックが各投稿にどの程度含まれているかをトピック分布（図 4）から算出し，検索語と各投稿との関係性を $DocScore_{w_i,d_j}$ として定量化する．

まず，各検索語 w_i に対して生起確率の高い上位 5 つのトピック T_{w_i} として特定する．

図 3. トピック特定

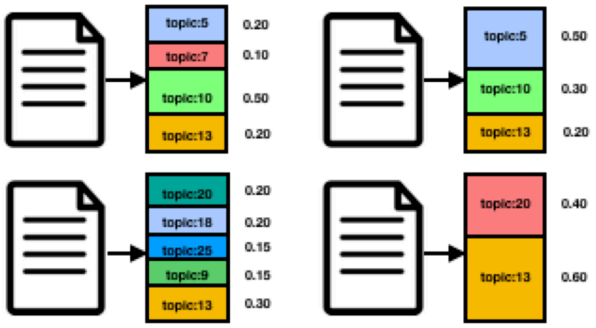


図 4. トピック分布の抽出

$$T_{w_i} = (t_1, t_2, t_3, t_4, t_5) \tag{1}$$

これらの 5 つのトピックを利用して式（2）よりに各投稿ごとに $DocScore_{w_i,d_j}$ を算出する．

$$DocScore_{w_i,d_j} = \sum_{k=1}^{5} \theta_{t_k,w_i} \times \phi_{t_k,d_j} \tag{2}$$

ここで，θ_{t_k,w_i} は検索語 w_i のトピック t_k における検索語の生起確率を表し，ϕ_{t_k,d_j} は投稿 d_j のトピック t_k のトピック分布の比率を表す．θ_{t_k,w_i} は，検索語 w_i の生起確率が高いトピックほど検索語 w_i に関する内容を含むトピックであると考え $DocScore_{w_i,d_j}$ の値が高くなる．また，ϕ_{t_k,d_j} についても，検索語と関連性の高いトピック t_k の比率が高いほどトピック t_k の内容を投稿 d_j が多く含むと考え $DocScore_{w_i,d_j}$ の値が高くなる．

2.4. 開発者の順位付け

次に，2.3 節で求めた $DocScore_{w_i,d_j}$ を利用して検索語と開発者の関係性を $AuthorScore_{dev_n}$ として算出し

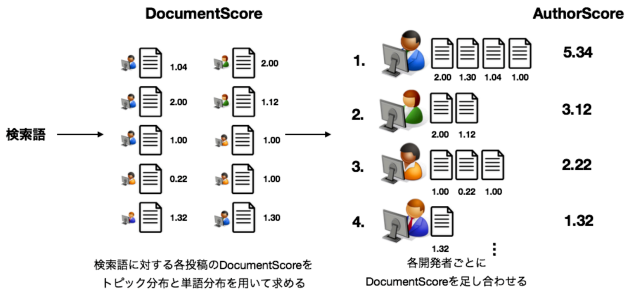


図 5. 開発者の順位付け

開発者を順位付けする（図 5）.

$$AuthorScore_{dev_n} = \sum_{m=1}^{NumOfMail} DocScore_{w_i,d_m} \quad (3)$$

$NumOfMail$ は各開発者が投稿した件数を表しており，$AuthorScore_{dev_n}$ は各開発者の各投稿の $DocScore$ を足し合わせることで算出する．$AuthorScore$ を開発者間で比較して，$AuthorScore_{dev_n}$ が高い開発者ほど検索語に関連したトピックを含む投稿を多く行っていると考えられる，すなわち，検索語と関連性の高い開発者と考えられるため，$AuthorScore$ の高い順に検索結果として出力する．

3. 評価実験

本章では，提案手法を評価することを目的に行った評価実験について述べる．

3.1. 目的

本実験の目的は，検索語に対してふさわしい開発者を本手法により検索できているかを確かめることである．実験の対象として複数のプロジェクトが混在しており様々な工程の開発者が使用している開発組織のメーリングリストアーカイブを用いることが望ましい．しかし，企業などの開発組織から大規模なデータを入手することが困難なため，本実験では 4 つの OSS プロジェクトのメーリングリストアーカイブを合成し，複数プロジェクトが混在したような合成データを作成する．各プロジェクトのドメイン用語を検索語とした時，対象としたドメイン用語を用いるプロジェクトに所属する開発者を検索できるかどうか調べることで提案手法を評価する．

表 1. 評価実験に使用するデータセット

プロジェクト	期間	件数	開発者数
Apache Hive	2008/11/11~2010/10/07	1,881	28
Apache Pig	2007/10/31~2010/09/28	1,662	44
Apache ZooKeeper	2007/10/31~2010/09/28	1,506	23
Apache Chukwa	2009/03/10~2013/10/08	877	25
合計		4,209	118

3.2. 対象データセット

本実験では，Apache Hadoop プロジェクトのサブプロジェクトである Apache Chukwa，Apache ZooKeeper，Apache Hive，Apache Pig のメーリングリストアーカイブを評価対象とする．本手法は大規模組織での人材配置に役立てることをの最終的な目標としているため，1 つの組織を再現するためにプロジェクトの内容が離れすぎていないプロジェクトを選択した．また，正解集合を作成しやすいように，Hadoop のサブプロジェクトであるがサブプロジェクト間で開発者の重複が少なく，データが取得しやすいという理由で上記 4 プロジェクトを選択した．

3.3. データセットの作成

実験のデータセットとして使用する対象プロジェクトの開発者向けメーリングリストのメールデータを mbox 形式で取得する．取得した mbox ファイルは月ごとにすべての投稿が 1 つのファイルに連結して保存されている．mbox ファイルから実験に必要なデータ（送信者情報，メッセージ ID，本文など）を取得することができる．ただし，開発者メーリングリストには，開発者同士のやりとり以外にプロジェクトで利用されているツール (Bugzilla, JIRA, Jenkins 等) から通知される投稿も含まれてるため，それらの投稿はあらかじめ除去した．また，本文に含まれる URL や開発者の署名は LDA を利用してトピック抽出する際のノイズとなるため，正規表現を利用してできる限り除去した．

上記の処理を施した後の本評価実験で使用したデータセットの内訳を表 1 に示す．メーリングリストへの投稿は総計 4,209 件，開発者（投稿者）は延べ 118 人であった．この内，複数のプロジェクトに参加している開発者が 2 人存在する．取得したこれら 4 つのサブプロジェクトの mbox ファイルを 1 つの mbox ファイルにまとめることによって，複数プロジェクトが混在した合成データ

表 2. tf-idf 値の高い上位 10 単語

Hive		Pig		Zookeeper		Chukwa	
単語	tf-idf 値	単語	tf-idf 値	単語	tf-idf 値	単語	tf-idf 値
hiveopensourc	29494.33	physicallay	6189.56	nioservercnxn	964.15	adaptor	818.90
srcpart	7406.33	pig	5193.71	zookeep	906.55	chukwa	683.94
metaexcept	5130.52	mapreducelay	4413.45	nioservercxn	842.17	hicc	420.35
jobconf	4256.35	cocoon	2088.30	todd	810.10	demux	398.50
thrift	3680.84	gld	1757.78	clientcnxn	756.99	collector	296.34
srcbucket	3274.15	inactiveaccount	1718.07	socket	662.31	seqfilewrit	281.23
protect	2116.89	piggybank	1497.06	peer	543.22	agent	267.72
numreducetask	2010.47	eq	1425.57	antlib	504.34	row	249.19
processnam	1842.93	acct	1389.78	hunt	488.12	depart	243.32
libjar	1746.07	foreach	1334.48	elect	483.75	scienc	243.26

を作成する．

3.4. 評価実験用の検索語の抽出

評価実験において，提案手法が検索語にふわさしい開発者を提示することができるかどうかを調べるために，実験で用いる検索語をあらかじめ用意しておく必要がある．評価実験で用いる検索語は，各プロジェクトのドメインの特徴を表すような用語（ドメイン用語）が望ましい．本論文におけるドメイン用語とは，特定の開発のみで使用される用語，または，特定の開発を表すような用語を指す．

評価実験では，文書集合の単語の重要度を評価する手法である tf-idf を用いて各プロジェクトの特徴語を抽出し検索語を決定する．また，プロジェクト内で頻繁に利用されている語もプロジェクトの特徴を表しているのではないかと考え頻出語を抽出して検索語とする．頻出語は tf 値を用いて抽出することが一般的であるが，tf 値を用いて抽出すると tf-idf 値を用いて抽出した場合と重複することが多いため，本研究では df 値を用いて抽出した．以降では，評価実験用の検索語を抽出するために行った処理について説明する．

3.4.1. 汎用語の削除

まず，図 6 のように，すべてのプロジェクトで共通して利用される汎用語を削除する．汎用語は，各プロジェクトから特徴語と頻出語を抽出する際のノイズになると考えられる．例えば，投稿メッセージの冒頭でのあいさつ（Hi や Dear など）や，実験で用いたプロジェクトは hadoop のサブプロジェクトであるため hadoop などの用語も全てのサブプロジェクトのメーリングリストで出現する汎用語に含まれる．

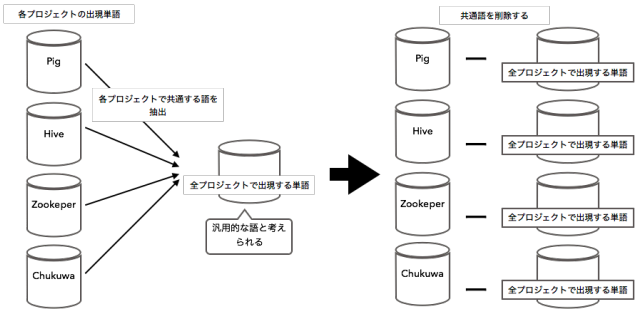


図 6. 汎用語の削除

3.4.2. 特徴語の抽出

文章中の単語の重要度を評価する手法の 1 つに tf-idf がある．tf-idf は文章 d における単語 t の出現頻度 $\mathrm{tf}(t,d)$(Term Frequency) と，単語 t が出現する文書数の全文書数に対する割合である $\mathrm{df}(t)$ の逆数の対数をとった逆文書頻度 $\mathrm{idf}(t)$(Inverse Document Frequency) に基づいて，式（7）で算出することができる．

$$\mathrm{tf}(t,d) = \frac{|\{t \in T_d\}|}{|T_d|} \tag{4}$$

$$\mathrm{df}(t) = \frac{|\{d \in D : t \in T_d\}|}{|D|} \tag{5}$$

$$\mathrm{idf}(t) = \log_2 \frac{1}{\mathrm{df}(t)} \tag{6}$$

$$\mathrm{tfidf}(t,d) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t) \tag{7}$$

ここで，T_d は文章 d で出現する単語の集合を，D は全文書の集合を表している．

本実験では各投稿 1 通を 1 つの文書として td-idf を適用し，各プロジェクト毎に出現単語の tf-idf 値を算出した．出現単語の tf-idf 値を目視で確認して，プロジェクトの特徴を表すと考えられる上位 10 単語を各プロジェクトの特徴語として抽出した．表 2 に抽出した各プロジェクト毎の特徴語（if-idf 値の高い単語）を降順に示す．

3.4.3. 頻出語の抽出

各プロジェクトの頻出語は式（5）の $df(t)$ の値の高い上位 10 単語を用いる．$tf(t,d)$ は単語の使用頻度である

表 3. df 値の高い上位 10 単語

Hive		Pig		Zookeeper		Chukwa	
単語	df 値	単語	df 値	単語	df 値	単語	df 値
hiveopensourc	7311.00	pig	18039.00	zookeep	2152.00	chukwa	1560.00
hive	4844.00	physicallay	1194.00	hunt	404.00	adaptor	347.00
srcpart	2041.00	mapreducelay	830.00	nioservercnxn	242.00	hicc	168.00
thrift	1357.00	schema	569.00	clientcnxn	239.00	demux	168.00
metaexcept	1231.00	foreach	539.00	todd	216.00	collector	123.00
jobconf	1102.00	gate	516.00	elect	201.00	scienc	115.00
srcbucket	836.00	cocoon	495.00	socket	198.00	depart	114.00
protect	590.00	piggybank	452.00	nioservercxn	193.00	jiaqi	113.00
numreducetask	456.00	gld	407.00	lock	165.00	agent	107.00
metastor	448.00	javacc	365.00	peer	148.00	seqfilewrit	79.00

ため特定の文書に偏って同じ単語が使用されていても高い値が示されるのに対して，$df(t)$ は単語 t が出現する文章が多いほど高い値を示すため，プロジェクト内で幅広く使われている単語も各プロジェクトの特徴を表すのではないかと考えた．表 3 に各プロジェクト毎の頻出語（df 値の高い単語）を降順に示す．

3.5. 実験手順

評価実験は手順で行った．

手順 1：LDA モデルの作成 3.3 節で述べた 4 つのプロジェクトを組み合わせた合成データに対して LDA を適用する．LDA ではトピック数を任意の数に設定できる．設定したトピック数が検索精度に与える影響を調べるため，本評価実験ではトピック数をそれぞれ 100，500，1000 の 3 つのモデルを作成した．各モデルに対して各投稿のトピック分布およびトピック毎の単語分布を計算する．

手順 2：特徴語・頻出語を用いた開発者の検索 3.4 節で述べた各プロジェクトの特徴語および頻出語を検索語として用いて提案手法を適用する．2.4 節で述べた AuthorScore の高い上位 10 名の開発者を検索結果とする．

手順 3：検索結果から正答数と正答率を算出 手順 2 の検索で得られた開発者がどの程度正しく検索できているか検証するために，検索して得られた開発者が特徴語および頻出語（検索語）が出現するプロジェクトに参加していれば正解，参加していなければ不正解としてトピック数とプロジェクト毎に正答数と正答率を算出する．

4. 実験結果

表 4 に，特徴語（tf-idf 値の高い上位 10 単語）を検索語とした場合の平均正答率をプロジェクト毎に示す．また，表 5 に，頻出語（df 値の高い上位 10 単語）を検索語とした場合の平均正答率をプロジェクト毎に示す．表 4 および表 5 ではそれぞれ，設定したトピック数別（100, 500, 1000）で平均正答率を示している．ここでの正答率とは，表 2 および表 3 の検索語を用いて検索を行なった結果得られる上位 10 名の開発者が，検索語が属するサブプロジェクトに何名参加していたかを表す．例えば，hiveopensource を検索語とした場合に Apache Hive に参加する開発者が 10 名中何名検索結果として提示されていたかで正答率を求める．したがって，ここでの平均正答率とは，検索語 10 単語を用いて検索を行った場合の正答率の平均値を表す．

表 4. 特徴語（tf-idf 値の高い上位 10 単語）を検索語とした場合のプロジェクト毎の平均正答率

プロジェクト	トピック数		
	100	500	1000
Apache Hive	81%	61%	62%
Apache Pig	91%	100%	99%
Apache ZooKeeper	61%	70%	73%
Apache Chukwa	71%	67%	78%

表 5. 頻出語（df 値の高い上位 10 単語）を検索語とした場合のプロジェクト毎の平均正答率

プロジェクト	トピック数		
	100	500	1000
Apache Hive	78%	65%	66%
Apache Pig	86%	99%	91%
Apache ZooKeeper	66%	75%	75%
Apache Chukwa	71%	69%	82%

以下ではサブプロジェクト毎に検索精度を議論する．

4.1. Apache Hive プロジェクトの実験結果

特徴語を用いた検索結果

すべての検索語の正答率の平均では，トピック数を 100 に設定した時に正答率が 81%で最も高い値を示し，トピック数を 500 に設定した時の正答率が 61%で最も低い値を示した．トピック数を 1000 に設定した時の正答率は 62%であった．

頻出語を用いた検索結果

すべての検索語の正答率の平均では，トピック数を 100 に設定した時に正答率が 78%で最も高い値を示し，トピック数を 1000 に設定した時の正答率が 65%で最も低い値を示した．トピック数を 500 に設定した時の正答率は 66%を示した．

4.2. Apache Pig プロジェクトの実験結果

特徴語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 500 に設定した時の正答率が 100%で最も高い値を示し，トピック数を 100 に設定した時の正答率が 91%で最も低い値を示した．トピック数を 1000 に設定した時の正答率は 99%であった．

頻出語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 500 に設定した時の正答率が 99%で最も高い値を示し，トピック数を 100 に設定した時の正答率が 86%で最も低い値を示した．トピック数を 1000 に設定した時の正答率は 91%であった．

4.3. Apache ZooKeeper プロジェクトの実験結果

特徴語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 1000 に設定した時の正答率が 73%で最も高い値を示し，トピック数を 100 に設定した時の正答率が 61%で最も低い値を示した．トピック数を 500 に設定した時の正答率は 70%であった．

頻出語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 1000 と 500 に設定した時の正答率が 75%で最も高い値を示し，トピック数を 100 に設定した時の正答率が 66%で最も低い値を示した．

4.4. Apache Chukwa プロジェクトの実験結果

特徴語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 1000 に設定した時の正答率が 78%で最も高い値を示し，トピック数を 500 に設定した時の正答率が 67%で最も低い値を示した．トピック数を 100 に設定した時の正答率は 69%であった．

頻出語を用いた検索結果

すべての検索語の正答率の平均は，トピック数を 1000 に設定した時の正答率が 82%で最も高い値を示し，トピック数 500 に設定した時の正答率が 69%で最も低い値を示した．トピック数 500 に設定した時の正答率は 71%であった．

4.5. 実験結果のまとめ

各プロジェクトとトピック数の設定によって正答率に差はあったが，正答率は 61%〜100%であった．Apache Pig の特徴語を用いてトピック数を 500 に設定した時の正答率が 100%で最も高く，Apache Hive の特徴語を用いてトピック数を 500 に設定した場合の正答率が 61%で最も低かった．Apache Hive ではトピック数を 100 に設定した時，Apache Pig ではトピック数を 500 に設定した時，Apache ZooKeeper と Apache Chukwa ではトピック数を 1000 に設定した時が最も正答率が高かった．

5. 考察

本章では，評価実験の結果を踏まえ，提案手法の有用性と今後の課題について考察する．

5.1. 正答率と提案手法の有用性

実験の結果，Apache Pig の検索語を用いて実験を行った場合が最も正答率が高い結果となった．表 1 で示したように Apache Pig の開発者数は 44 人と最も多く，他の 3 サブプロジェクトは多くとも 28 人である．これらの結果を鑑みると，多くの開発者が議論に参加する活発なメーリングリストであったため，Apache Pig の開発者の経験やスキル，役割などに沿って LDA によるトピック抽出が他のサブプロジェクトに比べて上手く行えたのではないかと考える．結果として，検索語から得られる上位 10 人の開発者が，検索語毎に比較的分散して出力された結果，他のサブプロジェクトよりも高い平均正答率を示したのではないかと考えている．

一方で，特徴語を検索語とした場合の Apahce Zookeeper は，61%（トピック数）と最も平均正答率が低かった．トピック数を増やすことで平均正答率が 70%

表 6. 正答率が半分以下の検索語

検索語						
thrift	libjar	javacc	socket	antlib	row	scienc
hicc	peer	numreducetask	srcbucket	seqfilewrit	jobconf	

∼ 73%に向上するものの，平均正答率には改善の余地があると言える．また，Apache Hive のようにトピック数を増やすと平均正答率が低下するプロジェクトも存在するため，トピック数を増やすことが平均正答率の向上につながるとも一概には言えない．Apahce Zookeeper の開発者は 4 つのサブプロジェクトの中では最も少ない 23 人であり，メーリングリストへの参加者数が正答率に影響を与えている可能性も否定できない．これらのことから，プロジェクトの開発者数，投稿数，トピック数からは平均正答率との一貫した因果関係を読み取ることはできないため，検索精度の向上へ向けて今後は各開発者の投稿内容をより詳細に分析する必要がある．

評価実験の結果全体としては，すべてのプロジェクトにおいても平均正答率は 6 割以上を示しており提案手法は一定の有用性を示したと考えている．

5.2. 検索語と正答率

実験の結果，検索語によって正答率の違いがあることが分かった．各プロジェクトで使用した検索語のうちトピック数 100，500，1000 のいずれかに設定した際に正答率が半分以下であった検索語を表 6 に示す．

正答率が半分以下であった検索語のうち thrift, libjar, javacc, socket, antlib, row, science, peer については，ソフトウェア開発に関するプロジェクトでは頻繁に利用される単語であるため，各プロジェクトの正答率が半分以上であった検索語と比較すると各プロジェクトの特徴を表すことができず正答率が低かったと考えられる．

thrift は，Apache Thrift プロジェクトのことを指していると考えられる．Apache Thrift は RPC（リモートプロシジャーリコール）フレームワークのプロジェクトである．RPC はネットワークに接続された他の PC 上でプログラムを呼び出すためのプロトコルである．

libjar は，Java 言語において外部ライブラリをインポートする際に lib フォルダに jar ファイルを置くため，外部ライブリのインポートの一連の流れに関する内容がデータに含まれていたため出現した単語であると考えられる．

javacc は，Java Compiler Compiler の略であり Java 言語向けの構文解析器生成ツールである．

socket は，ネットワーク用語の一つであり通信プロトコルを利用する際に通信の出入り口を Socket のことを指していると考えられる．このようにこれらの単語は必ずしも各サブプロジェクトの開発に特化した内容で用いられる用語ではなく，サブプロジェクト内で比較的頻繁に利用される語であった．

さらに，numreducetask の正答率が低かった原因として，複数のプロジェクトで使用されていたため各プロジェクトの特徴を表すことができなかったと考えられる．chukwa プロジェクトでは tf-idf 値が 13.55，pig プロジェクトでは tf-idf 値が 52.56 で出現していた．3.4.1 節で述べたように，本研究では汎用語をあらかじめ除去して合成データを作成しているが，本評価実験では 4 つのプロジェクトで共通して出現する単語のみを除去している．2 つあるいは 3 つのプロジェクトで共通して出現する単語も汎用語として除去することで平均正答率の向上につながる可能性がある．しかしながら，汎用語として多くの用語を除去すれば，該当するトピック・開発者が紐付きにくくなる可能性もあるため，汎用語の削除の効用については実験条件を追加し比較する必要がある．

また，正答率が半分以下であった残りの hicc，srcbucket，seqfilewrit については，検索語として抽出したプロジェクトでのみ使用される語であったが正答率は半分以下であり，正答率の低さを説明することが現時点では困難なため，今後はより詳細に各検索語の利用される文脈を投稿内容から理解する必要がある．

5.3. トピック数

実験の結果から，トピック数の変化による正答率の変化はプロジェクトによって異なることが分かった今回の実験では，トピック数による正答率の大きな違いは見られなかったが正答率を上げるためにはデータに対して最適なトピック数を設定する必要があると考えられる．本

表 7. Apache Hive の特徴語：srcbucket

topic:100：正答率 (80%)					topic:500：正答率 (50%)				
#74	#97	#88	#75	#9	#225	#143	#2	#210	#244
ok	ok	org	junit	junit	timefram	junit	junit	org	junit
junit	org	type	info	org	timefram	tabl	data	type	info
exec	junit	hive	exec	info	maintain	ok	tabl	ok	org
tabl	data	problem	hiveopensourc	hive	hdf	data	ok	partit	data
info	tabl	java	usr	tabl	hive	partit	info	srcpart	ok
data	java	tabl	file	exec	zookeep	srcpart	srcpart	tabl	tabl
org	hive	data	org	ok	think	hr	partit	java	srcpart
partit	queri	class	build	file	client	usr	org	class	usr
hr	partit	refer	job	hiveopensourc	project	hiveopensourc	hr	hr	partit
build	srcpart	ok	data	usr	award	queri	exec	hive	hr

節では，特定のプロジェクトでのみ使用されているが，正答率が半分以下を示した hicc について設定したトピックのうち正答率が最も高かったトピック数と低かったトピック数で検索語が含まれるトピックとして提案手法により特定できた上位 5 つの構成語を比較することで，正答率とトピック数の違いを考察する．表 7 は，検索語として srcbucket を用いた時に特定できたトピックの主要な構成語を表している．正答率の最も高かったトピック数 100 の時と，低かったトピック数 500 の時に特定できたトピックの構成語を比較すると 3.4.3 節で特定できた Apache Hive の特徴語と頻出語がトピックの構成語として含まれている数は一緒であったが，正答率が高かったトピック数 100 の時の方が，hiveopensourc などの tf-idf 値がより高い語が多く含まれていた．正答率の高かったトピック数と低かったトピック数で特定できたトピックの構成語を確認することで正答率の高かったトピック数の方が，各プロジェクトの特徴語と頻出語が多く含まれることや，より tf-idf 値が高い語が含まれる場合が多いことが分かった．そのため，正答率が高い場合は正答率が低い場合よりも特定したトピックの内容が各プロジェクトに関連するトピックであったため正答率が高くなったと考えられる．

5.4. ソフトウェア開発組織への応用へ向けて

本実験では，OSS プロジェクトのメーリングリストを用いることで評価実験を行なった．しかし，OSS プロジェクトと企業などのソフトウェア開発組織ではコミュニケーションの取り方が異なると考えられる．OSS プロジェクトでは，メールでのコミュニケーションが中心であり議論の内容等がメールアーカイブに残りやすいが，ソフトウェア開発組織では会議や会話でのコミュニケーションが中心となるためメールアーカイブに開発に関する議論などが残りにくい．そのため，本手法をソフトウェア開発組織に応用するにあたり，メールアーカイブのみでなく会議の議事録や仕様書を利用することが有効であると考える．また，本実験では tf-idf 値と df 値を用いることで機械的に各プロジェクトの開発者を検索するための語を決定したが，ソフトウェア開発組織において利用する際の検索語は異なると考えられる．

6. 関連研究

本研究で提案したような開発者検索手法および推薦手法は広く研究されている．本節では，本研究に関する関連研究としてコードレビューや不具合修正などに着目した開発者推薦に関する研究を紹介する．

Rahman ら [4] は，開発者推薦において特にコードレビューを行うレビューアーについて着目した．コードレビューはテストやコーディングの初期段階で行われ，あらかじめバグとなる箇所を適切な開発者が目視でコードを確認しバグを発見することで開発の全体的なコストを削減することができることが知られている．レビュー待ちのプルリクエストに対して，過去に開発者がレビューしたプルリクエストとの類似度を測ることで推薦手法を提案している．

Alan ら [5] は，不具合修正を担当する開発者の推薦手法を提案した．開発者の推薦を行う際に，バージョン管理システムのデータと不具合管理システムのデータに着目した．双方のデータを用いた際の推薦手法の結果の比較を行い．双方のデータを用いた手法の利点について

述べた．より正確に目的の開発者を推薦したい場合は，バージョン管理システムをデータとして用い，正確さよりも関連する開発者を多く推薦したい場合は，不具合管理システムのデータを用いることが良いことを示した．

Bayati ら [6] は，特に情報セキュリティに関する分野に精通した開発者を推薦するための手法を提案した．ソフトウェアのセキュリティバグの深刻度について述べ，情報セキュリティに関する知識を持った開発者を推薦する手法を提案した．大規模 QA サイトである StackOverflow のデータから投稿に付与されたタグと，質問への回答数，信頼度スコアを利用することで開発者をスコアリングし開発者を推薦する手法を提案した．

これらの研究では，特定のタスク（レビュー，不具合修正など）における開発者の検索・推薦を支援することを目的としており，汎用性の高い開発者検索を目指す本研究とは立場が異なるが，検索結果の評価方法や検索精度の議論については本研究の参考になる．

7. まとめと今後の課題

本研究ではソフトウェア開発におけるコミュニケーションデータであるメーリングリストでの開発者のやりとりに着目し，大規模ソフトウェア開発組織における適切な人材配置を支援することを最終目的にして開発者検索手法を提案した．具体的には，メーリングリストに対してトピックモデルを適用し，メーリングリストからトピックと各トピックの単語分布，各投稿のトピック分布をそれぞれ算出し，検索語と各投稿との関係性の定量化するとともに検索語と開発者の順位付けして開発者を検索するための手法を提案した．

OSS プロジェクト 4 つのメーリングリストを用いて複数プロジェクトが混在した合成データを作成し，各プロジェクトの tf-idf 値の高い特徴語と df 値の高い頻出語を検索語として用いて評価実験を行なった．プロジェクトによって正答数にばらつきはあったが，すべてのプロジェクトで各プロジェクトの検索語を用いて検索を行なった結果，正答率は 61%〜100%であった．本研究における開発者の検索を行う際には，各プロジェクトでのみ使われるドメイン用語を検索語に用いると正答率が上がることが分かった．

今回はメーリングリストとメーリングリストに含まれる各投稿の内容把握のためにトピックモデルである LDA を用いた．しかし，LDA は事前にトピック数を設定する必要があり良い結果を得るためには最適なトピック数を推定する必要がある．今後は LDA 以外のアルゴリズムを用いてメーリングリストと各投稿から特徴量を抽出し，正答率を改善する予定である．

謝辞

本研究の成果は株式会社 SRA と国立大学法人和歌山大学との共同研究の一部である．また，本研究の一部は，文部科学省科学研究補助金（基盤 (A): 17H00731，基盤 (C): 18K11243）による助成を受けた．

参考文献

[1] Mockus, A. and Herbsleb, J. D.: Expertise Browser: A Quantitative Approach to Identifying Expertise, *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 503–512 (2002).

[2] Robbes, R. and Röthlisberger, D.: Using Developer Interaction Data to Compare Expertise Metrics, *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pp. 297–300 (2013).

[3] Blei, D. M., Ng, A. Y. and Jordan, M. I.: Latent Dirichlet Allocation, *J. Mach. Learn. Res.*, Vol. 3, pp. 993–1022 (2003).

[4] Rahman, M. M., Roy, C. K. and Collins, J. A.: CoRReCT: Code Reviewer Recommendation in GitHub Based on Cross-project and Technology Experience, *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 222–231 (2016).

[5] Anvik, J. and Murphy, G. C.: Determining Implementation Expertise from Bug Reports, *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pp. 2– (2007).

[6] Bayati, S.: Security Expert Recommender in Software Engineering, *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 719–721 (2016).

リスク構造化を用いたリスクマネジメント手法の提案と効果分析 ～「未来予想図」を用いたリスクマネジメント PDCA サイクル～

水野 昇幸
TOC/TOCfE 北海道
mizu-nori@e-mail.jp

安達 賢二
株式会社 HBA
adachi@hba.co.jp

要旨

ソフトウェア開発規模が大きくなり，プロジェクト型で開発することが一般的になっている．不確実性を伴うプロジェクト活動では，リスクを想定して管理することは重要である．しかし，リスクマネジメントは技術や経験が必要と言われ，現場でリスク対応が行われない場合もある．

本論文では，リスクマネジメントにて発生しやすい問題を解決するため，リスクを構造化した「未来予想図」を用いた手法を提案する．また，模擬プロジェクトとなる「折り紙」ミッションにて，本手法の効果があることを確認した．

1. はじめに

近年はソフトウェア開発の規模が大きくなり，プロジェクト型で開発することが一般的である．PMBOK（Project Management Body of Knowledge）においてプロジェクトとは「独自のプロダクト，サービス，所産を創造するために実施する有期性のある業務」と定義されている[1] ように，独自性を持つ．

プロジェクトのような独自性を持つ活動は，新たな活動を行うことを意味する．新たな活動は，先に何が発生するか分からない不確実性を伴う．この不確実性によって，定義されたタスクが予定通りに終了できない場合や，大きな遅れや損害に繋がってしまう場合もある．

これらの不確実性につながる事象を事前に把握して，状況をモニタリングしながら影響を最小化するために，PMBOK や ISO/IEC/JISQ31000 にて「リスクマネジメント」[1][2]が定義されている．

リスクを想定して管理することは重要である．しかし，このリスクマネジメントにて，担当者やマネージャとのリスク共有ができない場合，リスク管理シートの構築や更新の作業時間がかかってしまう．この作業負担によって，リスク管理シートが継続してアップデートされないことが多い．このような問題により，プロジェクトにおけるリスクマネジメントを効果的かつ継続的に行うことは難しい．

また，リスクマネジメントには実プロジェクト経験が必要とされるが，十分な数の担当者にリスクマネジメントが必要な規模のプロジェクト経験をさせることは困難である．

本論文では，一般的なプロジェクトにおけるリスクマネジメントにおいて発生しやすい問題を特定し，それらを解決するための手法とフレームワークを提案する．

2. リスクマネジメントにおける問題

本章では，一般的なプロジェクトにおけるリスクマネジメント手法と，その手順において発生しやすい問題を示す．また，その問題によって引き起こされやすい事象を示す．

2.1. 一般的なリスクマネジメント手法

一般的なリスクマネジメント手法としては，前述した PMBOK や ISO/IEC/JISQ31000 で定義されている．これらの手法について簡単に紹介する．

一般的なリスクマネジメントにおける全体構成を簡易に把握するため，ISO/IEC/JISQ31000 で定義されているリスクマネジメントのプロセスを図 1 に示す．本内容は PMBOK におけるプロセスとほぼ同義である．

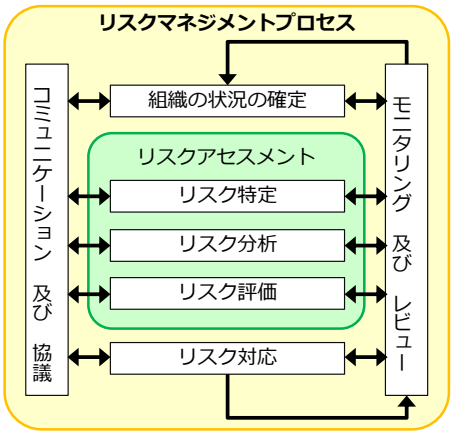


図 1. ISO/IEC/JISQ31000 のリスクマネジメントプロセス

ISO/IEC/JISQ31000 では，主要なプロセスとして図 1 における 7 つのプロセスが提示されている．PMBOK でも同様に「リスクの特定」，「定性的リスク分析」，「定量的リスク分析」，「リスク対応計画」，「リスクコントロール」という 5 つのプロセスが提示されている．

PMBOK におけるリスクマネジメントでは，一般的に表 1 で示される例のようなリスク管理シート（リスク登録簿）[1] やリスク・データ・シート[1]を用いた検討が行われる．

表 1. PMBOK にて紹介されるリスク管理シートの例

		定性的リスク分析				
ID	リスク記述	発生確率	影響度	リスク点数	対応	アクション
101	試験装置が他プロジェクトで使われており、必要時に投入できない	中	高	6	軽減	他プロジェクトのモニタ及び必要時に交渉
102	XXの影響で納期短縮	低	高	3	受容	

リスク管理シートでは表 1 の各行に対応する「リスク項目」単位でリスクを抽出し，それぞれ情報を付与する．例えば，定性的リスク分析向けとしては「ID」，「リスク記述」，「発生確率」，「影響度（対スコープ，品質，スケジュール，コスト）」，「リスク点数（優先順位を判断するための点数）」，「対応」，「アクション」というような情報を持つ．

上記に示したリスク管理シートが持つ情報はテーラリング対象であり，組織によって工夫が行われることが多い．しかしながら，共通的に「表」によって 1 つ 1 つの「リスク項目」単位で管理されている場合が非常に多い．

2.2. リスクマネジメントで発生しやすい問題点

前節で紹介したリスクマネジメントとリスク管理シートにおける問題点を整理するため，プロセスを「リスク特定」，「リスク分析・評価」「リスク対応」の 3 つに分けて，それぞれの問題点を示す．なお，分割したプロセスと PMBOK や ISO/IEC/JISQ31000 との対比を表 2 に記載する．

表 2. 分割したプロセスと各基準との対比

本論文 対象プロセス	PMBOK	ISO/IEC/ JISQ31000
① リスク特定	リスクの特定	リスク特定
② リスク分析・評価	定性的リスク分析 定量的リスク分析	リスク分析 リスク評価
③ リスク対応	リスク対応計画 リスクコントロール	リスク対応 モニタリング及びレビュー

それぞれのプロセスにて発生しやすい問題点について次に示す．

① リスク特定

本リスク特定プロセスにおいて発生しやすい問題点を次に記載する．

- 「決めつけ」的なリスクを特定してしまう
- 声の大きな人が決定してしまう
- 人によって感じている主要なリスクが異なる
- 具体的にリスクを特定する言語化が難しい

例えば，モデルベースで開発プロセスが厳格に決められている場合には，活動から逆算される「決めつけ」的なリスクが扱われてしまうことがある．また，衆知では無く一部の担当者の意見で決定されてしまう場合も多い．

② リスク分析・評価

本プロセスで発生しやすい問題は次のとおりである．

- （リスク管理シートで扱われることの多い）単一のリスク項目では，背景や他の影響が分かりづらい
- （全ての項目を管理する必要があるため）特定された多数のリスク項目を取り扱う必要がある

人により感じるリスクが異なる場合や，背景が分かりづらい場合に，「リスク項目や優先順位に実感ができない」，「解決すべきリスクが現場メンバー内やマネージャと共有ない」ケースが発生してしまうこともある．

③ リスク対応

本プロセスにて発生しやすい問題は次に示す．

- リスク対応が（優先順位があっても）しらみつぶし的になる
- 対応の効果がある部分を特定しづらい
- 初期検討では想定していないリスクが途中で発生してしまうことがある

多数のリスク項目を（優先順を用意しても）分析および対策をしてしまうことによって，多くの作業時間が必要となってしまう．この状況から「継続してリスク項目をアップデートすることができない」場合が発生してしまう．

以上の提示した問題により，リスクマネジメントを困難にする事象を引き起こしてしまう．

2.3. 問題構造と結果として発生してしまう事象

前節で提示したリスクマネジメントで発生しやすい問題は，それぞれ関連を持つ．これらの問題の関連性を矢印で結び，問題構造として整理したものを図 2 に示す．

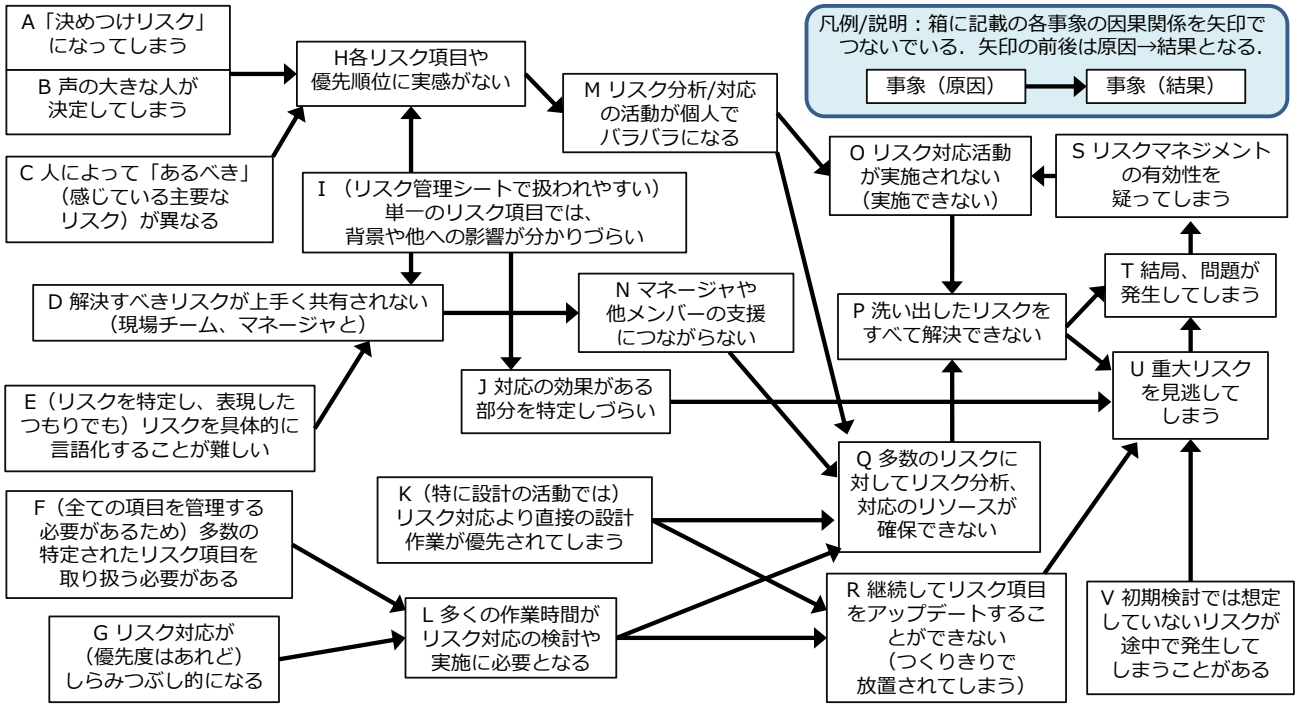


図 2. プロジェクトにてリスクマネジメントを実施している条件下にて発生しやすい問題構造

図 2 にまとめた問題構造から，リスクマネジメントを実施している条件下にて発生しやすい事象を次にまとめる．

リスク項目や優先順位に実感がない場合においては，リスク分析やリスク対応の活動が個人でバラバラになりやすい．解決すべきリスクがうまく共有できない場合には，マネージャの支援が行われづらい．加えて，リスク対応より直接の設計作業が優先されてしまうことから，多数のリスクに対してリソースが確保できない．この状況は特に設計担当者に発生しやすい．

結果として，「現在のリスク管理における最大の欺瞞は，リソースの裏付けがないリスク管理プランであると言える」[3]というように，リスク対応に十分なリソースをかけることができない状況が発生してしまう．

加えて，特定されたリスク項目すべてを分析し，対策をしらみつぶし的に実施する場合には，多数の作業時間が必要となってしまう．この作業の「重さ」から，継続してリスク項目をアップデートすることは難しくなる．

活動することによって判明するリスクも多く，初期検討では想定していないリスクが発生してしまうこともある．継続してリスク管理が行われない条件下では，重大リスクを見逃してしまい問題となる．結果として，リスクマネジメントの有効性を疑ってしまう場合すらある．

3. 解決策：リスク構造化を用いたリスクマネジメント手法

前章にて問題構造として示したリスクマネジメントにおける問題を解決するための方針とねらうべき問題点，提案する手法について示す．

3.1. 解決方針とねらうべき問題点

解決に向け，図 2 に示す各問題を引き起こす要因となる問題（主に図 2 左側）をねらう．要因側の問題を解決することにより，結果として図 2 記載の発生しやすい問題全体を解決する．

次のことが実現できる解決策を目指す．

- 特定されたリスク項目を共有できる
- リスク対応の効果がある部分や背景がわかる
- リソースが限られることを前提として活動を絞り込む
- 継続してリスク項目のアップデートができる

提示した「特定されたリスク項目を共有できる」，「リスク対応の効果がある部分や背景がわかる」を実現するため，「リスク同士の関係性の構造化」を行う．特定したリスクに対して，「リスク構造化」を実施することでリスク共有と効果のある部分が判断可能となる．

このリスク構造化には，TOC（Theory of Constraints）思考プロセス[4]における FRT（Future Reality Tree：未来構造ツリー），もしくはロジックブランチ[5]と呼ばれる図 2 で示される連関図のような表記法を活用する．

また，提示した残りの問題とリスク構造の表記法に適合するプロセスとして「SaPID（Systems analysis/Systems approach based Software Process Improvement methoD）」[6][7]のプロセスとフレームワークを適用する．SaPID では当事者自らが解決すべき問題点を絞り込んで特定し，現実的に解決（改善）しながら段階的・継続的にゴールを目指す．

この SaPID における問題の洗い出しと構造化，改善ターゲットの特定とふりかえりの各プロセスをリスクマネジメントに取り込む．これによって，リスクを共有し，対策を絞り込み，ふりかえりでアップデートを行う PDCA サイクルを繰り返すことができる．

以上で提示した TOC 思考プロセスにおけるツールと SaPID におけるプロセスを活用して，従来のリスクマネジメントで発生しやすい問題を解決する新しいリスクマネジメント手法を提案する．

3.2. リスク構造化を用いたリスクマネジメント

提案するリスクマネジメント手法では，リスク構造を表現する図，リスクマネジメントのプロセスをそれぞれ用意している．それぞれの特徴を次に記す．

① リスク構造を表現する図（未来予想図）

「リスクも関連性を持ち，構造を持つ」という考えにて，リスク同士の関係性の構造化する．この構造は，リスク項目を時系列もしくは因果関係でつなぐことで整理する．

本手法ではこの図を「未来予想図」と呼ぶ．図 3 に未来予想図の例を示す．

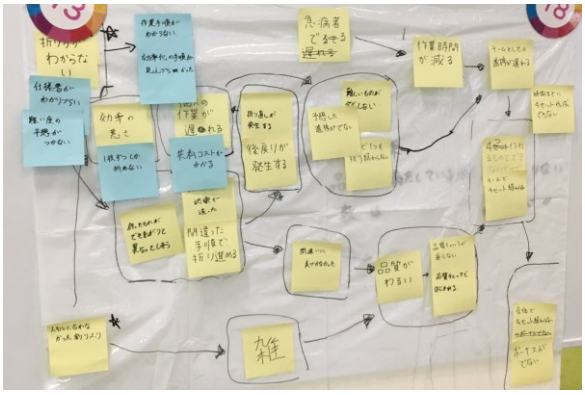


図 3．未来予想図の例

次に未来予想図の具体的な構造について説明する．

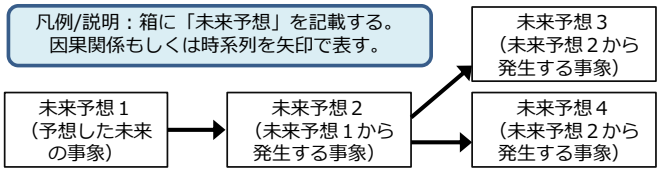


図 4．未来予想図の構造

未来予想図は，箱で表す「未来予想」から構成される連関図を表記法として用いる．ここで，未来予想という言葉は予想した未来の事象を表し，リスクと同義としている．「リスクとは，それが発生すれば少なくともスコープ，スケジュール，コスト，品質といったプロジェクト目標に影響を与える不確実な事象・状態」[1] という PMBOK 定義がある．現場の担当者が考える「未来予想」はこのリスク定義と大きな違いは無いと認識している．

現場でリスクマネジメントを経験した担当者が「リスク」という言葉にやらされ感の警戒心を抱くことは時折みられる．そのため，「リスク」という用語を使わず「未来予想」と呼ぶことで，より担当者の意見を引出しやすくしている．

未来予想（リスク）は因果関係もしくは時系列の関連性を示す矢印でつなぎ，構造化を行う．リスク間の原因と結果の関係を明らかにすることで，リスク背景や重要性の理解を促進することができる．加えて，将来発生するリスクにつながるリスクを特定することで，リスク対応の効果がある部分がわかる．限られたリソースですべての未来予想（リスク）の対策を行うことはできない．関係を示すことで効果があるリスク対応に絞り込んで活動を行うことができる．

現場で未来予想図を活用する際には図 3 のように付箋を用いることが多い．未来予想を書きだして整理するプロセスをチームで行うことで，リスク項目をチームで共有する副次的効果を得ることができる．

未来予想図を使用した検討例を図 5 に示す．

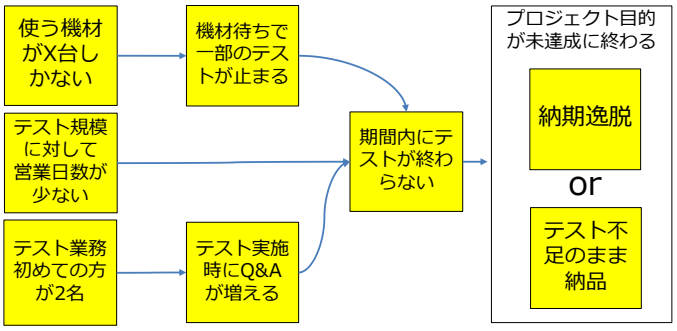


図 5．未来予想図を用いた検討例（テスト実施）

図 5 の例はテスト実施の前段階における未来予想図の検討例である．未来予想（リスク）の関連性を示すことで，各リスクの背景について理解が容易になる．また，リスク対応の効果がある部分を絞り込むことができる．

② リスクマネジメントプロセス

多数のリスク項目がある状況にて，解決策の絞り込みと継続したふりかえりによるリスク項目のアップデートを可能とするため，SaPID の手順を取り入れたプロセスを構築した．

● 提案するリスクマネジメントプロセス全体の進め方

提案するプロセス全体は，図 6 のように進める．各プロセスの詳細は後半にて記載する．

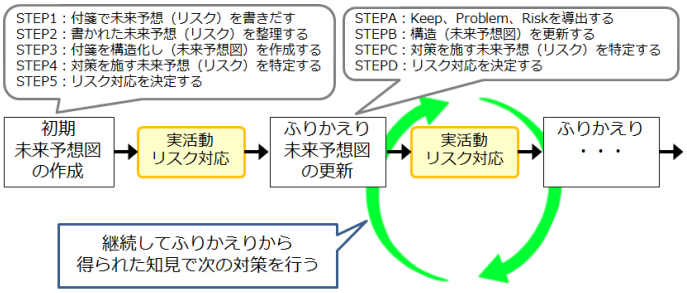


図 6. 提案するリスクマネジメントプロセス

まず，初期未来予想図の作成を行う．作成した未来予想図から対策を施す未来予想（リスク）を決定する．

ここから，リソース内で対応可能なように絞り込んだリスクへの対応を含めた実活動を行う．

次に，実活動の結果に応じてふりかえりを行う．このふりかえりにて不要なリスク，新たに見つかったリスクを発見して未来予想図を更新する．ふりかえりで更新した未来予想図から，リスク対応を決める．

その後，次の実活動を行うことで，継続的に PDCA サイクルを回し，新たなリスクを捉えることができる．

提案するプロセスにおける「初期未来予想図の作成」と「ふりかえりと未来予想図の更新」の詳細ステップを次に示す．

● 初期未来予想図の作成

本作業は，何もない初期状態から未来予想（リスク）の特定とその構造化，リスク分析と評価を行うために実施する．この作業は次の 5 つの手順からなる．

なお，未来予想（リスク）という表現をしているが，単にリスクという用語を用いても良い．

STEP1：付箋で未来予想（リスク）を書きだす
STEP2：書かれた未来予想（リスク）を整理する
STEP3：付箋を構造化し，未来予想図を作成する
STEP4：対策を施す未来予想（リスク）を特定する
STEP5：リスク対応を決定する

STEP1 では予想した未来予想（リスク）を付箋に書き込むことによって，全員の意見を引出すことができる．

STEP2 では付箋で書かれた未来予想（リスク）の内容を 1 枚ずつ確認する．付箋に書きだすことによって，各付箋における意味の曖昧さ（明瞭性[5]）や真実かどうか（実体の存在[5]）を明らかにして，想定される事象や不安内容を具体化できる．

未来予想（リスク）は漠然とした「不安」を感じているだけの場合がある．この際，不安を感じている本人もうまく未来予想（リスク）を言葉で表現できないことが多い．付箋に書き出し，各付箋を精査する活動にて，未来予想（リスク）を具体的に特定できる．具体化の活動は，チームメンバー間の未来予想（リスク）共有にもつながる．

STEP3 ではそれぞれの未来予想（リスク）を時系列もしくは因果関係でつなぐ．未来予想（リスク）も問題と同じように，関連性をつなぐことにより，背景や解決が必要な理由が明らかにできる．関連性を明らかにすることは，優先的に解決すべき未来予想（リスク）を決定するために役に立つ．本ステップにて未来予想（リスク）を構造化することで「未来予想図」が完成する．

STEP4 では未来予想図から対策を施す未来予想（リスク）を特定する．リスク対応を行うリソースは限られていることが多いため，未来予想図から最も効果的と考える対象を絞り込むことが重要である．対策を施す未来予想（リスク）が確定した後に，STEP5 でリスク対応を決める．

● ふりかえりと未来予想図の更新

未来予想図が構築された後に，実際に対策活動を行う．ここで示す手順は，対策活動の後，定期的（1 週間～1 ヶ月程度の周期）に状況を把握して新たなリスクを見極めるために実施する．次の 4 つの手順からなる．

STEPA：Keep，Problem，Risk を導出する
STEPB：構造（未来予想図）を更新する
STEPC：対策を施す未来予想（リスク）を特定する
STEPD：対策を決定する

未来予想図を用いて未来予想（リスク）を検討した場合でも，想定外の発生や，とある未来予想（リスク）が不要とわかることがある．そのため，実活動結果から学習した知見を用い，追加，不要なリスクを明らかにする．

STEPA では通常のふりかえりに新たな未来予想(リスク)項目の導出を追加することで，活動から得られた新たな情報を引出すことができる．

STEPB にて得られた情報を用いて未来予想図を更新することで，最新の未来予想(リスク)情報が明らかになる．加えて，次のリスク対応の対象を特定できる．

STEPC と STEPD は前述した STEP4 と STEP5 と同じ内容のため省略する．

③　プロジェクトへの適用案

提案するリスクマネジメントをプロジェクトへ適用する場合には，未来予想図のみ適用することもできる．この場合，PMBOK におけるリスク管理シート(リスク登録簿)を未来予想図と置き換えることで実現することができる．リスクの優先順位を判断するために未来予想図を活用することも可能である．

ただし，より適用効果を得るためには，未来予想図とリスクマネジメントプロセスの双方を適用することを推奨する．プロジェクト計画段階において「初期未来予想図の作成」を行い，1 週間～1ヶ月程度の周期にて「ふりかえりと未来予想図の更新」を実施することで，新たに見つかったリスクを取込み，重要なリスクを優先して解決する活動を行うことができる．

3.3. 各対策手順と提示した問題との対応

記載した STEP 単位の手順は，2 章で提示した問題を解決する対策となる．提示した問題と解決策となる各手順との対応について表 3 にまとめる．

表 3．提示した問題と解決策となる各対策との対応

2章で提示した問題点		解決策
リスク特定		
	A「決めつけ」リスクになってしまう B 声の大きな人が決定してしまう	STEP1：付箋で未来を予想する
	C 人によって感じている 主要なリスクが異なる	STEP1：付箋で未来を予想する STEP2：書かれた付箋を整理する STEP3：付箋を構造（未来予想図）化する
	E（リスクを特定し、表現したつもりでも） 問題を特定した具体的な表現が難しい	STEP2：書かれた付箋を整理する
リスク分析・評価		
	F（全ての項目を管理するため） 特定された多数のリスク項目を 取り扱う必要がある	STEP3：付箋を構造（未来予想図）化する STEP4：対策を施す要素を特定する STEPA：Keep、Problem、Riskを導出する STEPB：構造（未来予想図）を更新する
	I（リスク管理シートで扱われる） 単一のリスク項目では、 背景や他の影響が分かりづらい	STEP3：付箋を構造（未来予想図）化する
リスク対応		
	G リスク対策が（優先度はあれど） しらみつぶし的になる	STEP4：対策を施す要素を特定する STEP5：対策を決定する
	J 対策の効果がある部分を特定しづらい	STEP4：対策を施す要素を特定する STEP5：対策を決定する
	V 初期検討では想定していないリスクが 途中で発生してしまうことがある	STEPA：Keep、Problem、Riskを導出する STEPB：構造（未来予想図）を更新する

提案するリスクマネジメント手法の各手順によって，図 2 において提示した原因となる問題（主に図 2 の左側）を解決していることがわかる．これらの解決をすることで，リスクマネジメントで発生しやすい問題を解決することができると考える．

この際，「L 多くの作業時間がリスク対応の検討や実施に必要となる」という点に関しては，提案するリスクマネジメントプロセスの PDCA を細かくまわすことによる作業時間の増加が懸念される．

しかし，実際には「F 多数の特定されたリスク項目を取り扱う必要がある」と，「G リスク対応がしらみつぶしになる」という必要がなくなることによって，全体のリスク対応作業時間は減り，合計で作業時間を削減することができる．

なお，本対策を実行したとしても次のような問題点が発生することが多い．括弧は対象の手順である．

- 付箋で記述したとしても，一部の人の意見が多数を占めてしまう（STEP1）
- 1 枚の付箋において表現が複数含まれている場合や，表現が具体化されておらず，解決対象が特定できない（STEP2）
- 真実かどうか不明確な内容が含まれる（STEP2）
- 同じような表現が多数記載されて整理が難しい（STEP2）
- 関連性に繋がりが無いものが含まれる（STEP3）

これらの問題点はロジカルシンキングなどの論理思考技術やファシリテーション技術で解決できる問題である．つまり，本手法を用いることで，解決が困難な問題が，トレーニング可能な既存のスキルや知見で解決できる問題に置き換わるともいえる．

本手法では，トレーニングを受けたファシリテータが支援してこれらの問題を解決することを推奨している．

4. 手法の効果測定結果と分析

提示手法に対し，実際のプロジェクトを模擬したミッションを用いてその効果を確認した．その結果を記載する．

4.1. 効果測定方法：「折り紙」ミッション

効果測定を模擬プロジェクト上で行うため，「折り紙」を複数種類完成させるミッションを用意した（図 7 参照）．体験者は IT 系を中心とした様々な人たちでチーム構成される．4 名で 1 つのチームを作り，合計 4 チームでこの折り紙ミッションを実施した．

<折り紙成果物（例）>

項目	色
サンタクロース	ピンク
さんたぶーつ	赤
つりー	緑
いちまいぼし	黄
いちごのけーき	ピンク
ぶらうす	茶色

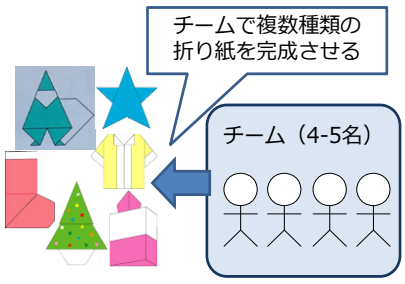


図 7. 効果測定のための折り紙ミッション

この折り紙ミッションは「CCPM 折り紙ワークショップ」[8]というクリエイティブコモンズライセンスのオープンコンテンツを活用している.「CCPM 折り紙ワークショップ」は，実施するミッションと問題解決のプロセスの 2 つに分かれているが，このミッション部分のみを採用して効果測定を行った.

今回、問題解決プロセスは，「図 6. 提案するリスクマネジメントプロセス」の内容を採用している.

- 実作業との対比とプロジェクト作業の模擬度合い

参考として，今まで 69 名に対して「折り鶴」が完成するまでの時間を測定した結果を図 8 に示す.

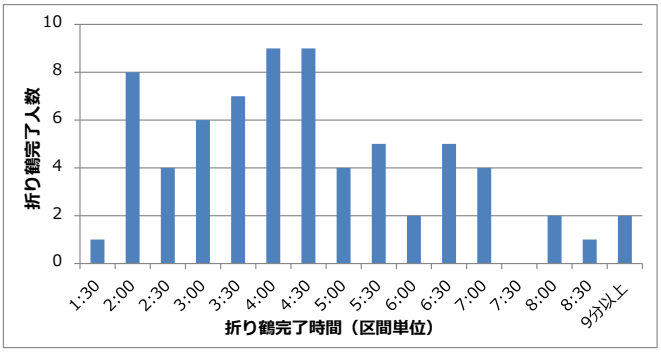


図 8. 折り鶴完成までの時間

この時間のばらつきは，不確実性を伴うタスクを実施する際に発生すると言われる確率分布「ベータ分布[9]に近い. スキルや経験が異なる人員が集まる状況では，折り紙作業でも図 8 のような不確実性によるばらつが発生する. この性質を活用することで，「新たな種類の折り紙を折る」作業において，実作業に近い状況を生み出すことができる.

しかし，折り紙ミッションでは同一種の折り紙を複数セット作成する. つまり，生産現場における作業に近い. 表 4 に生産現場における作業と折り紙作業の対比を示す.

表 4. 生産現場における作業と折り紙作業の対比

No	生産現場における作業	折り紙作業
①	開発方法の確立	新たな種類の折り紙を折る
②	生産の仕組み構築	一度作成した折り紙の 効率的作成方法を構築
③	生産の効率化	構築した方法の効率化

このうち，プロジェクトの実作業に近い不確実性を持つ作業は「①開発方法の確立」である. ①の作業はミッションの半分程度の時間を占めているため，折り紙ミッションは 50%程度はプロジェクトを模擬できると考えている.

4.2. 効果測定対象ミッションの進め方

次に，効果測定対象ミッションの進め方を示す.

最初に「折り鶴」を 1 度作成することで，体験者が自分のスキル（折り紙作成速度）を把握する. 体験者は折り鶴での結果を考慮して作業（指定時間 15 分で完成するセット数）を見積る. その上で，折り紙ミッションを開始する.

新しい種類の折り紙を折る作業は体験者にとって新規の活動であり，不確実性が存在する. この不確実性に対してチームとしてリスクを特定する. そのリスクを整理することで未来予想図を構築する. 未来予想図を用いて対策を検討し，折り紙ミッションに取り掛かる. 折り紙ミッションは 5 分間隔でインターバルを入れ，定期的にふりかえりの活動を行う（図 9 参照）.

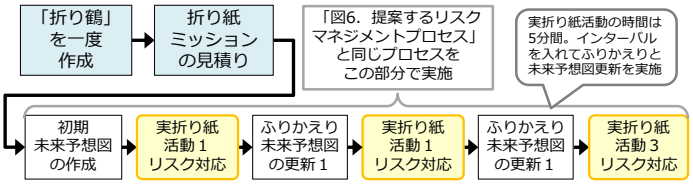


図 9. 効果測定ミッションの進め方

この折り紙ミッションを通じて，提案するリスクマネジメント手法の効果を測定した.

4.3. 効果を測定する対象とその理由

効果測定として，今回の「折り紙」ミッション自体の見積りに対する成果物の実際の完成数を比較した. また，測定項目を用いて，ねらうべき問題が解決したかを確認した. 表 5 に確認した測定項目とその理由を示す.

記載した測定項目にて，本手法でねらう問題解決の効果があることを確認する. あわせて，活動成果が想定よりも向上することも確認する.

表 5. 効果測定項目とその確認理由

効果測定項目	確認理由
見積りでの予想完成数と実際の完成数	実際の活動成果が向上したことの確認を行う
付箋を出した人数比率	一部の人では無く、全員が意見を出して多様さが得られていることを確認する
ふりかえり後での付箋更新枚数	活動から学習し、新たなリスクの追加、不要なリスクを除去した状況を確認する
定性意見	主に次の4点の効果をアンケート意見から確認する。 ①構造化によって背景や関係性を特定しリスクを共有できる ②納得したリスク対策の決定 ③ふりかえりでの学習効果、新たなリスク発見の効果 ④理解度、有用性、満足度

4.4. 効果測定結果とその分析

表 5 で示した項目単位で効果を測定した結果を次に記載する.

① 見積り時の予想完成数と実際の完成数

図 10 に,「見積り時の予想完成数と実際の完成数」の結果を次に記す. 見積り完成数は体験者 16 名それぞれの想定, 結果はチーム単位の完成セット数で示す.

見積り時予想完成数			実際の完成数	
完成見積り数	予想した人数		チーム	完成セット数
1セット完成	4人		チームA	4セット
2セット完成	6人		チームB	3セット
3セット完成	5人		チームC	4セット
4セット完成	1人		チームD	3セット
平均	**2.2セット**		平均	**3.5セット**

図 10. 見積り時の予想完成数と実際の完成数

初期は主に 1～3 セット(平均 2.2 セット)しか完成出来ないと見込まれていたが, 平均 3.5 セット完成した状況が確認できる. 見積りより成果が出ていることが確認でき, 活動成果が予想より向上したと考えることができる.

② 付箋を出した人数比率

こちらは, STEP1 の手引きとして「ひとり 5 枚作成」としていたため, 必ず全員の意見が出ることになる. 付箋を出した人数比率は, 平均した数となった. 本件は, 適切なファシリテーションを行うことで効果が出る部分と考え, 効果の判断から除外する.

③ ふりかえり後での付箋更新枚数

今回の「折り紙」ミッションでは, 図 9 で示すように初期の「未来予想図」作成後, 2 回ふりかえりを実施する. このふりかえり内で「未来予想図」を更新する. 図 11 が更新前後における未来予想図の実例である.

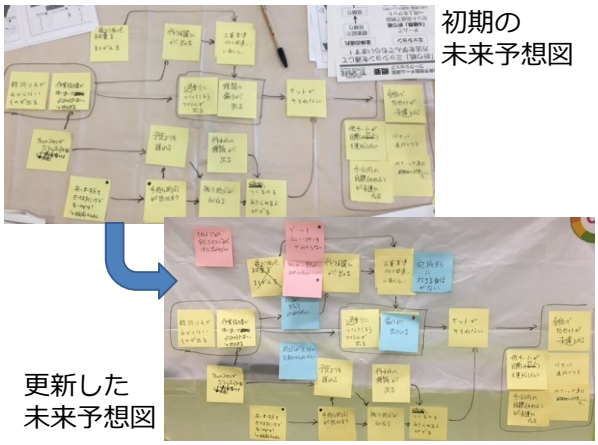


図 11. 未来予想図の更新前後における実例

「折り紙」ミッションを体験した 4 つのチームにおいて, ふりかえり毎の「追加された付箋」と「削除された付箋」について数を測定した結果を表 6 にまとめる.

表 6. ふりかえり単位での付箋更新結果

	ふりかえり1		ふりかえり2	
チーム	追加数	削除数	追加数	削除数
チームA	6	0	2	0
チームB	4	2	3	0
チームC	2	3	1	1
チームD	7	0	3	2

すべてのチームにおいて, ふりかえりで付箋の追加が行われている. つまり, 活動による学習によりリスクが更新されている状況が判断できる. また, 不要なリスクが削除されていることもわかる.

以上から, 本手法によってふりかえり単位で活動から学習した情報を基に新たなリスクの追加や, 不要と判断したリスクを除去する効果が確認できる.

④ 定性意見

最後に「定性意見」として, 体験者からのアンケート結果(回答は 15 名)の一部を示す. まず, アンケートの自由記述において, 3 点のねらいの効果に触れていた体験者の数を表 7 に記載する.

表 7. アンケート自由記述でのねらいの効果への言及

アンケート自由記述での対応項目	人数（15名中）
構造化によって背景や関係性を特定しリスクを共有できる	4名
納得したリスク対策の決定	2名
ふりかえりの学習効果、新たなリスク発見の効果	5名

15 名の体験者においてそれぞれ数名が，想定した効果に対して言及していた．

本アンケートは自由記述のため，特にねらった部分のみを確認することはできていない．しかし，アンケート結果により，「リスク共有」，「ふりかえりによる新たなリスク発見」の効果が出ている状況について確認できた．

また，体験者に「理解度」，「有用度」，「満足度」の 3 点に関して 5 点満点にて評価して頂いた結果を次の表 8 に記載する．（回答は 15 名）

表 8. 理解度，有用度，満足度確認結果

項目	5点	4点	3点	100点換算
理解度	11人	3人	1人	93点
有用度	11人	3人	1人	93点
満足度	14人	1人	0人	99点
※2点、1点はなかったため省略				

体験者には満足度や理解度のみではなく，有用度に関しても非常に高い評価を頂いており，実際に現場で適用もできるという意見も獲得している．

以上の定性意見を含めた測定結果により，提案するリスクマネジメント手法にて，従来のリスクマネジメントにおいて発生しやすい問題を解決することができると考えている．また，実際に活動成果を向上する効果もあるとみなしている．

なお，より効果を明確に示すために，今後の活動時において追加の測定を行う予定である．

5. まとめと今後の課題

不確実性を伴うプロジェクト活動では，リスクを想定して管理することは重要だが，このリスクマネジメントは技術や経験が必要と言われ，現場でリスクを解決する活動が実施されない場合もある．

本論文では，このリスクマネジメントにて発生しやすい問題点を提示し，解決するための手法を提案した．

提案した手法は，「未来予想図」と呼ぶ特定した未来予想（リスク）を構造化するための表記法を用いる．また，SaPID における継続的な改善プロセスを取り入れることで，リスクを共有し，対策を絞り込み，ふりかえりでアップデートを行う PDCA サイクルを繰り返す．

プロジェクトと同様の不確実性が発生する「折り紙」ミッションを用いて，この手法の効果の確認を行った．結果として，リスクマネジメントで発生しやすい問題を解決していること，ならびに実際の活動成果が向上していることを確認した．

今後の課題として，より効果を明確に示すための測定方法を強化することを想定している．また，実プロジェクトでは非常に多数のリスク項目が特定される場合がある．この多数のリスク項目を効率的に構造化するための方法を今後取入れていく予定である．

参考文献

[1] Project Management Institute (著), プロジェクトマネジメント知識体系ガイド（PMBOK ガイド）第 5 版 (A Guide to the Project Management Body of Knowledge), 鹿島出版会, 2014

[2] ISO/IEC/JISQ31000 , Risk management-Principles and guidelines リスクマネジメント－原則及び指針, 2009

[3] 長尾 清一著, 先制型プロジェクト・マネジメント―なぜ，あなたのプロジェクトは失敗するのか，ダイヤモンドセールス編集企画, 2003

[4] エリヤフ・ゴールドラット（著），三本木 亮（翻訳），ザ・ゴール 2－思考プロセス, ダイヤモンド社, 2002

[5] 岸良 祐司, きしら まゆこ（著），考える力をつける 3 つの道具, ダイヤモンド社, 2014

[6] 安達賢二，自分事化影響要因に着目した中期経営計画立案・展開への共創アプローチ[現状分析～計画立案編], ソフトウェア・シンポジウム 2017 in 宮崎, 2017

[7] 安達賢二，自律型プロジェクトチームへの変革アプローチ事例 チームの価値観変容を重視し，問題モデリングを活用した SaPID 流プロセス改善アプローチ，ソフトウェアプロセス改善カンファレンス 2015, 2015

[8] 水野昇幸，CCPM 折り紙ワークショップ（共有版），https://www.slideshare.net/NoriyukiMizuno/ccpm-52215583

[9] エリヤフ・ゴールドラット（著），三本木 亮（翻訳），クリティカルチェーン―なぜ，プロジェクトは予定どおりに進まないのか?, ダイヤモンド社, 2003

システム理論に基づくモデリングと質的研究を併用したソフトウェアプロセス教育の動機づけシナリオ改善

日下部 茂
長崎県立大学
kusakabe@sun.ac.jp

梅田 政信
九州工業大学
umerin@ci.kyutech.ac.jp

片峯 恵一
九州工業大
katamine@ai.kyutech.ac.jp

石橋 慶一
福岡工業大学
ishibashi@fit.ac.jp

要旨

パーソナルソフトウェアプロセス(PSP)のトレーニングコースを大学で実施する際の改善の取り組みについて述べる. これまでの取り組みで, 受講生の動機づけに着目し, 動機付けプロセスの標準状態遷移モデル(後に実践的状態遷移モデルに改称)を用いていた. そのモデルは, 組織論的期待モデルをベースに定義した基準状態遷移モデルの拡張したもので, PSP 受講者を状態機械とみなし, 動機づけに関わる状態と操作により動機づけプロセスを定式化している. このモデルにより, PSP に関する知識やスキルの導入から定着の成功や失敗に至るまでの過程の形式的な表現が可能となった. 次の課題として, 状態遷移機械としてモデル化された学生の状態遷移が実際に望ましいものとなるような指導を行う必要がある. システム理論に基づく STAMP/STPA と質的研究アプローチにより, このような課題を解決することについて提案する.

1. はじめに

PSP (Personal Software Process)[1]は, ソフトウェア技術者のための自己改善のプロセスであり, 自身の開発プロセスを自律的に管理し, データに基づいて自身が開発するソフトウェアの品質改善を図るものである. このような PSP の有効性に着目し, 大学の教育においても PSP のトレーニングをとり入れる試みがなされている. 例えば, 九州工業大学では, カーネギーメロン大学ソフトウェアエンジニアリング研究所(CMU/SEI)と連携し, PSP および TSPi (Team Software Process Introduction)[2][3]を正規の大学院教育に取り入れ, 高度情報通信技術者の育成に取り組んでいる[4].

そのような事例において, PSP は開発するソフトウェアの品質向上に必要なスキルの修得に有効なことが示される一方, 全ての受講者が PSP コースを修了できている訳ではないといった問題も生じている. このような問題に対する取り組みの一つとして, 動機づけに着目し, 組織論的期待モデルをベースに, PSP の教育コース受講生の動機づけの状態遷移モデルを用いるアプローチが提唱されている[5] (図 1 ① ②). PSP 受講者を状態機械とみなし, 動機づけに関わる状態と操作により動機づけプロセスを定式化する. このような PSP 受講者の動機づけの観点のモデルと定式化をベースに, PSP に関する知識やスキルの導入の指導からその定着の成功や失敗に至る過程の形式的な表現が可能となる. そのような表現を用いることで, PSP 教育実施方法について, より客観的な改善策の検討ができるとされている[6]. 本提案は, このような動機付けプロセスの定式化を起点とした研究であり, その概要を図 1 に示す.

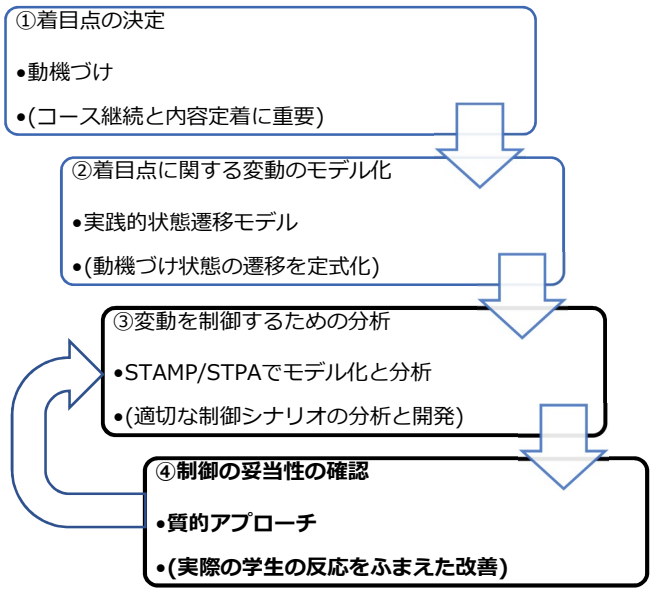


図 1 PSP 教育の改善の取り組み

動機づけの状態遷移モデルを用いて PSP 教育コースを実施・改善するには, インストラクタ役の教員が適切に受講者の動機づけ状態を把握し, 適切な状態遷移シナ

リオとなるように指導する必要がある. このような制御の問題を適切に表現して分析する観点から, システム理論に基づくハザード分析手法である STAMP/STPA (Systems-Theoretic Accident Model & Process /Systems Theoretic Process Analysis)[7]を活用する試みも行った[8] (図 1 ③). STAMP/STPA を用いることで, 動機づけのモデル上での制御の分析が可能になった. しかしながら, 実際の学生は人間であり, 指導に対する反応をコンピュータシステムのように考えることは適切ではない. そのため, 動機づけの状態制御の分析に, システム理論に基づく分析と質的研究アプローチによる分析を組み合わせることで, このような課題を解決すること提案する (図 1 ③④).

2. 動機づけプロセスの状態遷移モデル

2.1. 動機づけプロセスとその構造

動機づけ理論は, 動機づけに至るプロセスに関する過程理論と要因に関する内容理論の二つに大別でき, 本研究は過程理論の一つをベースにした既存研究に基づいている. Lawler の期待モデル[9]に環境や組織の要因を組み入れた組織論的期待モデル[10]を基礎として, 動機づけプロセスの構造を表現する.

図 2 は, 組織論的期待モデルをもとに, 新しい技術や手法の導入時の個人の実行プロセスと動機づけプロセスおよびその環境や組織による監視制御プロセスとの関係を表現したモデルである[5]. 図中, Bep は, 努力 E によって意図したレベルのパフォーマンス P が達成できるという個人の期待の主観確率 (0~1) を表す. また, Bpoi は, 意図したレベルの Performance(P) が報酬 Oi をもたらす主観確率 (0 ~ 1) を表す. Vi は, パフォーマンス P に伴って生じる報酬 Oi に対する個人の情動志向や魅力の程度を表す誘意性 (-1 ~ +1) である. ここで, Bpoi * Vi の総和を求めるのは, 一般に遂行 P に対して複数の種類の報酬 Oi があり得るためである. Bep, Bpoi, Vi の積は, 努力 Effort(E)がパフォーマンス P に結び付く可能性が高く(Bep >> 0), その P が何らかの報酬 Oi をもたらす可能性が高く(Bpoi >> 0), かつ, その報酬 Oi が望ましいもの

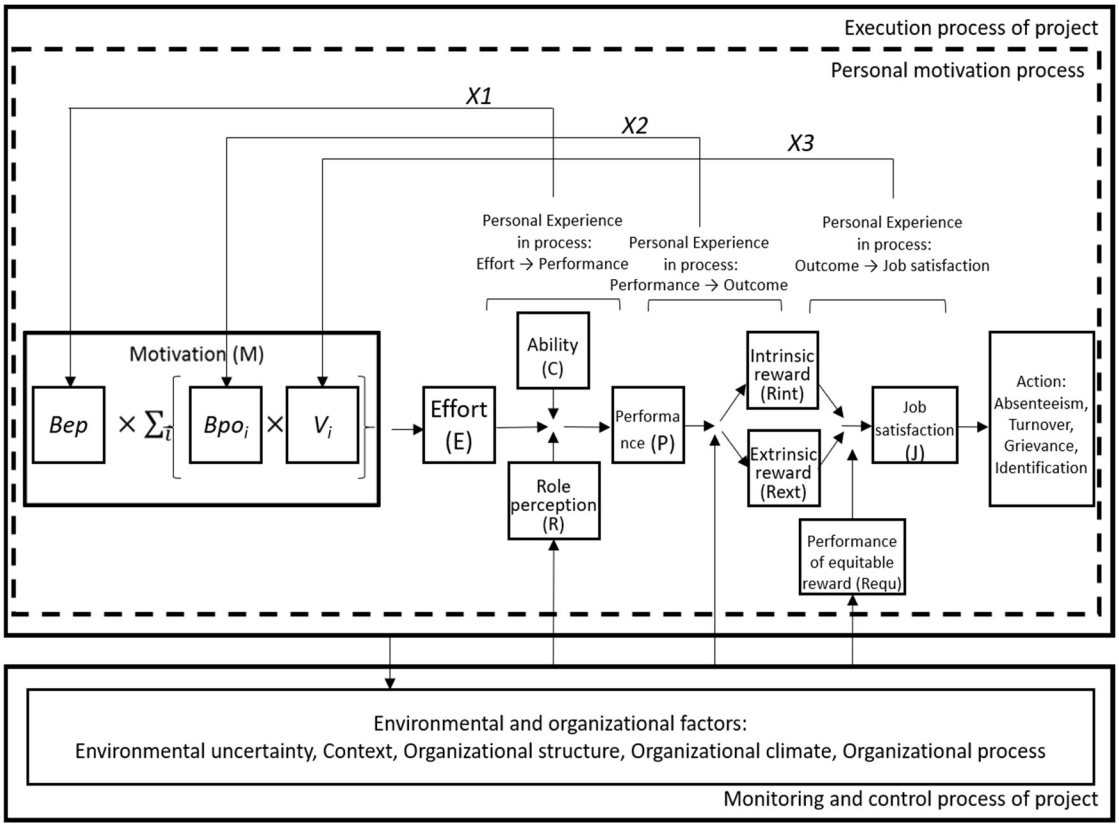


図 2 組織論的期待モデルに基づく動機付けプロセスの構造

(Vi >> 0) であれば, 高い動機づけ Motivation(M)が得られることを表している.

努力 E は, この動機づけ M により決まり, パフォーマンス P は, 努力 E と能力 Ability (C), 役割知覚 Role Perception (R)の積で決まる. パフォーマンス P は, 仕事の達成感等の内的報酬 Intrinsic Reward (Rint) と昇給や昇進等の外的報酬 Extrinsic Reward (Rext) のいずれか, またはその両方をもたらす. 職務満足 Job satisfaction (J) は, 内的報酬 Rint, 外的報酬 Rext, および, 報酬公平度の認知 Performance of equitable reward (Requ) の積によって決まり, 欠勤や苦情, 組織や部門への同一化といった行動を引き起こす. また, X1 ~ X3 の矢印は, 努力→パフォーマンス, パフォーマンス→報酬, および報酬→職務満足の各プロセスにおける個人経験が, 動機づけに関わる Bep, Bpoi, および Vi に, それぞれ反映されることを表す.

2.2. 状態遷移モデル

動機づけプロセスの状態遷移モデルは個人や組織を一つの状態機械と見なし, プロセスの状態とそれに対する操作により動機づけプロセスを定式化する. このようなモデルを用いて, PSP トレーニングコースの受講者の動機づけ状態を, インストラクタによって観測, 制御することを目指す.

公式な PSP トレーニングコースには複数のバージョンがあり, ここでは PSP-Planning と PSP- Quality の 2 コースからなる, PSP for Engineers を前提に説明する. 各コースは, 講義と演習の対を 1 日の実施単位とした 4 日間の後に, 最終日に総括的なレポート課題を課す. 九州工業大学の大学院教育での実施例では, この PSP for Engineers を, 時間割構成に合わせて提供している. PSP コースの基準状態遷移モデルは, そのような PSP コースで導入される実践内容とその目的とをそれぞれ遂行 P と役割知覚 R とし, 講義と課題の組を一つの操作 OP として定義したモデルである. その後, 実際の PSP トレーニング講義の指導経験をもとに, よりきめ細かなモニタリングと指導を反映できる実践的状態遷移モデル(提案時は標準状態遷移モデルで後に改称)が提案されている.

このような動機づけを観点とする PSP 受講者のモデルにより, PSP に関する知識やスキルの導入からその定着の成功や失敗までの動機づけ状態の遷移過程の形式的な表現が可能で, そのような過程の表現を用いて, PSP コース実施方法の改善策の検討をより客観的に行うことを目指していた. 例えば, 動機づけプロセスの実践的状態遷移モデルにおいて, それぞれの状態が取りうる値について表 1 のように想定し, 定着の成功や失敗に至る受講者の動機づけの状態遷移のシナリオが議論されている.

表 1 要因の状態集合例

要因	State value set
Bep	{VeryHigh, High, Low, Unknown}
Bpo	{High, Low, Unknown}
V	{High, Low, Unknown}
努力 *E*	{VeryHigh, High, Low, Unknown}
能力 *C*	{VeryHigh, High, Low, Unknown}
役割知覚 *Ri* (i=1..87)	{Perceived, NotPerceived, Unknown}
パフォーマンス *Pj* (j=1..10)	{Accomplished, NotAccomplished}
課題 A*j* (j=1..10)	{NotGiven,Given,PlanningCompleted, Completed}
内的報酬	{Given, NotGiven}
外的報酬	{Given, NotGiven}
職務満足	{HighLevel, LowLevel}

2.3. 対象事例

今回分析の対象とする, 九州工業大学での PSP トレーニングコースの講義では, 次のような学生の望ましくない状況と対応方針が用いられていた.

(1) SEI による PSP 修了基準(全課題完了)を満たさずにコースをやめる
- TSP/PSP 報告会や産業界講師による講演といった産学連携を通じ, 就職後も含めて PSP 修了の重要性と魅力を伝える

(2) 大学の単位修得要件(課題の 2/3 を完了)を満たさずにコースをやめる
- 単位修得要件を明示し, 単位修得を促す

(3) プロセスを改善できずに同じ誤りを繰り返す
- 改善できない原因の究明の手助けをすると同時に, 対応する指導を繰り返し行う

(4) プロセスの改善点の一般化ができない
- 回答を与えずに, 気づくまで繰り返し指導を行う

(5) 進捗が遅延し課題を予定通り完了できない
- 毎回の講義開始時や事前に遅れが察知された時点で進捗を確認し, 適宜(週単位で)講義日時の変更や追加説明を行う

(6) 分析が不十分で適切な改善提案ができない

- 適宜気づきを促す助言を与える

(7) Engineering のスキルの低さによりプロセス改善効果がでない
- Software Engineering 視点で助言を与える

前述のような状態遷移をもとに実際の PSP トレーニングコースの指導の実践と改善を行うには，望ましい状態遷移を実現し，望ましくない状態遷移を回避するよう，インストラクタ役の教員が適切に受講者の状態を把握し適切な制御を行う必要がある．そのような指導について，前述の動機づけプロセスモデルと関連付けた指導シナリオを系統的に導くために STAMP/STPA のモデル化と分析の試みを行った．

3. システム理論によるモデル化と分析

表 1 のような要因の状態集合を持つ動機づけの状態遷移の制御について，すべて要因の状態を展開して分析するのは困難である．制御の問題について，適切な抽象化を用いてモデル化と分析が可能な手法として，我々は STAMP/STPA に着目した．

3.1. STAMP

STAMP は，従来の解析的還元論や信頼性理論ではなくシステム理論に基づき，システムを構成するコンポーネント間の相互作用に着目したアクシデントの説明モデルである．STAMP のモデルは，次のような安全制約，階層的なコントロールストラクチャ，プロセスモデルという三つの基本要素で構成される．

- コントロールストラクチャ：システムの構成要素間の構造と，相互作用を表したもの
- プロセスモデル：コントロールする側がその対象プロセスをコントロールするアルゴリズムと対象プロセスを(抽象化して) 表現したもの

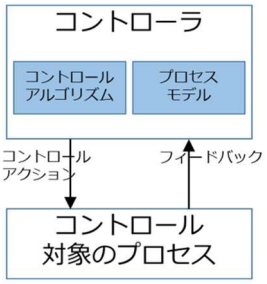


図 3 STAMP の基本的コントロールストラクチャ

- 安全制約：安全のために守るべき制約

STAMP での基本的なコントロールストラクチャとプロセスモデルの概略を図 3 に示す．例えば，PSP トレーニングの講義の場合，コントローラがインストラクタ役の教員で，コントロール対象が受講者と考える．一般には，そのような基本形を組み合わせて，システム内のコンポーネントの構造とそれらの間でやり取りされる制御の指示やフィードバックなどを表す．

STAMP では，このようなコントロールストラクチャとプロセスモデルに対して，システムの安全制約が正しく適用されているかどうかに着目する．STAMP では，安全に関するコントロールストラクチャがシステムの安全制約を守れず，システムがハザード状態になることでアクシデントが発生すると考える．STAMP での，アクシデント，ハザード，安全制約は以下のように定義されている[11]．

- アクシデント：望んでもないし計画もしてない損失につながるようなイベント．安全工学の技法を出来るだけ広く適用する意図で STAMP では広めの定義としており，損失は，人命喪失，けが，物損，環境汚染，ミッション喪失，経済的損失といったものもアクシデントとして考えることができる．
- ハザード：環境のある最悪な条件の集合と重なることでアクシデントにつながるような，システムの状態もしくは条件の集合．ハザードは対象システムの境界内のものであり，システムの範囲，範囲外の環境との境界が明確になっている必要がある．また，ハザードが実際に損失につながるのは，組み合わさる，最悪の環境条件の存在が必要．
- 安全制約：ハザードが識別されると，それらからシステムを安全に保つための要件もしくは制約を導く．トップダウンに考える場合，まず高レベルの安全制約が導かれるがその段階ですべて確定するとは限らず，分析の途中でも導出，修正され得る．

STAMP 自身はアクシデントを説明するモデルであり分析技法ではないが，STAMP をベースとして，解析の道具立てやプロセスが複数提案されている．STAMP を用いた安全制約の分析は，人や組織の制御の問題にも適用できる．例えば，ソフトウェアプロジェクトの場合，開発成果物に対してだけでなく，そのような成果物の開発プロセスや運用プロセスに対しても行うことができる．

3.2. STAMP/STPA による指導の分析

STPA では STAMP モデルの前述の三つの要素を用い，コントロールを行う側とその対象プロセスとの間の相互作用において，安全制約が不適切であったり，守られ

ない状態になったりするシナリオを中心に分析する．STPA は典型的にはトップダウンに実施する方法である．今回の場合も個々の要因の状態値の可能な組み合わせを列挙するようなボトムアップ的なアプローチはとらず，まず回避したい状態をハザードとしてトップダウンに定義した上で分析を開始する．

ハザード分析を行う目的や段階の違いも含め，STAMP/STPA の具体的な適用法には様々なバリエーションがあり得るが，以下に典型的な手順の例を示す．

- Step0 準備 1：アクシデント，ハザード，安全制約の識別
- Step0 準備 2：コントロールストラクチャの構築
- Step1：安全でないコントロールアクション（Unsafe Control Action：UCA）の抽出
- Step2：非安全なコントロールの原因の特定

このようなモデル化と分析を，PSP の指導に対して適用することを試みた．

3.3. STPA 準備

PSP トレーニングの講義の場合，コントローラがインストラクタ役の教員で，コントロール対象が受講者であるコントロールストラクチャを考える．動機づけプロセスモデルにおける状態遷移を考慮した指導シナリオを考えるために，被コントロール側に，図 1 の中の個人の動機づけプロセスが内在することとする．

コントローラである教員は，被コントロール側の受講生の状態をプロセスモデル変数として持ち，その値も参照した上で指導アルゴリズムにもとづき指導アクションを出す．指導アクションは，被コントロール側の動機づけ状態に直接的もしくは間接的な影響を与えると考え，受講生の状態の変化を把握し，コントローラ側のプロセスモデルの変数として保持している，受講者の動機づけ要因の状態変数の値に反映させる．(図 4 参照)

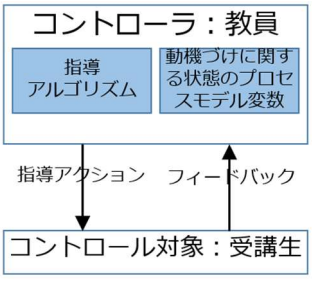


図 4　PSP 指導のコントロールストラクチャ

次に，アクシデント，ハザード，安全制約を決める．STAMP/STPA は安全工学の技法を広く適用できるよう意図されており，例えばアクシデントをある種のミッションの失敗とすることで，多様な問題に適用することが可能である．ここで，アクシデントは，受講者が受講を中止することとする．これは動機づけプロセスモデルでの欠勤や離職に相当する．ハザードは，パフォーマンス P に問題があり課題を実施できていない状態とし，何か最悪の条件が重なると履修中止になりかねないと考える．安全制約は，指導に関するコントロールループで課題に関するパフォーマンス P にそのような問題が生じないこととする．

アクシデントに相当する受講中止のような行動と関係がある職務満足 J は，内的報酬 Rint，外的報酬 Rext，および，報酬公平度の認知 Requ の積によって決まるものの，同時に，内的報酬や外的報酬が間接的に依存している努力 E を決定づけるものでもある，動機づけ M の要因にもフィードバックされるという，循環的な依存関係がある．状態の要因間の関係は単純な線形でも木構造といったものでもなく，まずどの要因に着目するか構造によって決めるのは容易ではない．STAMP/STPA は，アクシデントやハザードを決めトップダウンにモデル化と分析を行うため，動機づけプロセスの状態遷移における要因もトップダウンに分析できる．ハザードとの関連を考慮し，パフォーマンス P に問題がある状態をハザードとして系統的に分析を試みた[8]

このように考えることで指導アクションの検討に際し，考慮すべき事項の再構築につながる．前述の望ましくない状況と対応方針の中で，以下をアクシデントに相当と考える．便宜上それぞれ A1，A2 とラベルをつける．

(1) SEI による PSP 修了基準(全課題完了)を満たさずにコースをやめる (A1)
(2) 大学の単位修得要件(課題の 2/3 を完了)を満たさずにコースをやめる (A2)

また，以下がハザードとその回避対策案に相当すると考える．便宜上，ハザードに H1 とラベルをつける．

(5) 進捗が遅延し課題を予定通り完了できない (H1)
- 毎回の講義開始時や事前に遅れが察知された時点で進捗を確認し，適宜(週単位で)講義日時の変更や追加説明を行う

3.4. STPA/Step1 非安全なコントロールアクション

前述のようにアクシデント，ハザード，安全制約，コントロールストラクチャを定めた後，コントローラからのアクションのうち次の 4 タイプの非安全なコントロールアクションを識別する．

1. 与えられないとハザード (Not Providing causes hazard) : 安全のために必要とされるコントロールアクションが与えられないことがハザードにつながる.
2. 与えられるとハザード (Providing causes hazard) : ハザードにつながる非安全なコントロールアクションが与えられる.
3. 早過ぎ, 遅過ぎ, 誤順序でハザード (Too early/too late, wrong order causes hazard) : 安全のためのものであり得るコントロールアクションが, 遅すぎて, 早すぎて, または順序通りに与えられないことでハザードにつながる.
4. 早過ぎる停止, 長過ぎる適用でハザード (Stopping too soon/applying too long causes hazard) : 連続的, または非離散的なコントロールアクションにおいて, 安全のためのコントロールアクションの停止が早すぎる, もしくは適用が長すぎることがハザードにつながる.

PSP のトレーニングコースの公式の教材に沿った指導の場合でも, 誤った前提にもとづく不適切な説明を伴った指示や, 受講生や作業環境などの準備が整っていない状況での指示は非安全なコントロールアクションになり得る. 前述の H1 に相当する状況での対処について, ハザードになり得る指導について検討する. 以下を最初の対象アクションとして検討を開始する.

- 遅れを察知する
- 進捗を確認する
- 講義スケジュールを変更する
- 追加の説明を行う

例えば, 上記の「追加説明を行う」でも以下のように非安全なものとなり得る. このような分析は表形式で行うことが多い(表 2 参照).

1. 与えられないとハザード: 努力をパフォーマンスに結びつけるには追加の説明が必要なのに与えられずハザードに.
2. 与えられるとハザード: 望ましい状態遷移が起こさないような不適切な方法で追加説明を行いハザードに.
3. 早過ぎ, 遅過ぎ, 誤順序でハザード: 不適切なタイミングや, 誤順序での追加説明でハザード
4. 早過ぎる停止, 長過ぎる適用でハザード: 追加説明指導だと該当はない. (例えば, プロセスデータを提出させる, といった進捗状況の把握のための継続的な指導を早くやめるなどする場合にハザード)

3.5. STPA/Step2 非安全なコントロールの原因の分析

非安全なコントロールを識別したのち, それにつながるシナリオを考え原因を識別する. 安全制約を保つためのコントロールループやその構成要素を確認し, 以下のような点を分析する. (例は図 5 参照)

1. プロセスモデルが正しくなくなってしまう原因も含め, どのようにして非安全なコントロール, ひいてはハザードにつながるかを, 分析する.
2. 必要なコントロールアクションが与えられたにもかかわらず, 適切に実行されない原因を分析する.

例えば必要な追加説明が与えられない場合, 教員がその必要性を認識できない状況に至った原因, 例えば受講生や作業環境などの準備が整っていない状況を把握できなくなるなどの原因を分析する. また, 前述の四つのパターンに加え, 一見妥当に見える指導アクションでも, 受講生の振る舞いが期待と異なる場合, その原因を探るなどする. STPA/Step2 は一般に対象問題領域の専門知識を必要とし, Step0 や Step1 とくらべて手順を定型化するのが困難とされている.

さらに, 今回の検討事例の場合, 人である受講生の動機づけの問題であり, 動機づけプロセスを状態遷移機械でモデル化したとしても, 受講生の振る舞いは機械のものでなく, 決定的なものとしては扱えない. そのような問題の分析にあたり, 質的研究アプローチ, 特に GTA (Grounded Theory Approach)を用いることとした[12].

4. 質的アプローチの併用

今回の事例では, 指導アクションの対象が人であり, その振る舞いについて理解する必要がある. そのため, STAMP/STPA の Step2 において, 質的研究アプローチを併用した. 質的研究アプローチは, 現象が起こるプロセスや文脈を重視し, 質的な理解や説明, 解釈を求めるもので, 質的研究でもデータを重視するが, データは当事者の会話や観察などである. そのような質的研究アプローチの中でも, 単なる記述や印象の議論でなく, インタビューや観察などにより得られたデータの分析法を提唱している GTA を用いた.

4.1. Grounded Theory Approach

大学での PSP コース受講者は, 講義担当以外の教員に加え, クラスメート, 研究室の先輩や指導教員などからなる小さな社会に属していると考えることができる. そのため, 社会的な現象が生じる仮説や理論の構築も想定した

GTA が適していると考えた. GTA は, 最初の提唱の後, 様々な異なる方法が提唱され, 唯一絶対の GTA と言えるものはない. しかしながら, 基本的に, データ収集で得られた結果をまずテキスト化した上で, 特徴的な単語などをコード化し, コード化されたデータに基づいてカテゴリやその間の関係を分析し, 現象が生じる仮説や理論を構築する. その手順はおおよそ次のようなものである.

- リサーチクエスチョンの設定
- フィールドデータの収集
- データに対するコーディング
- カテゴリ抽出および関連の分析
- 適宜, フィールドデータ収集から繰り返す

4.2. 適用

分析対象プロセスの変遷の違いが生じる状況を分析する意図の下, 協力が得られた受講経験者八名を二群に分け, データの収集以降の分析を二度繰り返した.

リサーチクエスチョン

リサーチクエスチョンは,「コースの進行に従ってどのように受講者の動機づけが変遷するのか」とした.

フィールドデータの収集

受講経験者に,「受講のきっかけと, 受講時の状況, 完了の見通し」について半構造化インタビューを行った.

データに対するコーディング

データの分析に対する考え方の違いにより具体的な GTA に様々なバリエーションがある. 今回は, プロセスやその変化に着目したものを用いた. 特徴的な単語やフレーズとして, 開発プロセス体験, 時間的高コスト, チームプロセスによる開眼, 指導の教員による差, などがあった.

カテゴリの分析

高レベルのカテゴリとして, 想定済みの肯定事項, 想定済みの否定事項, 追加的な肯定事項, 追加的な否定事項, を抽出した.

これらの結果を用い, 動機づけの観点から, 非安全な指導シナリオの分析を行った. その結果, それまで気づいてなかった指導の問題, 例えば, 課題の評価や再提出に関する指導法の一部が不公平感を生じさせ, 受講者の動機づけを低下させる問題を発見することができた.

5. おわりに

本論文では, 組織論的期待モデルをベースに定義, 拡張されてきた, PSP の教育コースを対象にした動機づけプロセスの実践的状態遷移モデルを, 実際の教育の改善に活用するための方法を提案した. 動機づけの状態遷移モデルを用いることで, 指導の指針を立てやすくなるものの, 状態遷移機械とみなした学生が望ましい状態遷移を実際に起こすような指導を実現することは必ずしも容易ではない. 本発表では, システム理論に基づく STAMP/STPA と質的研究アプローチの一つである GTA を組み合わせて用いることで, このような課題を解決すること提案した. 提案手法の試行の結果, これまで見落としていた指導方法の問題を発見できた. 今後も GTA での理論的な飽和を目指し, データの収集と分析を繰り返す.

参考文献

[1] Humphrey, W. S. (秋山義博監訳 JASPIC TSP 研究会訳). PSP ガイドブック:ソフトウェアエンジニア自己改善. 翔泳社. 2007.

[2] Humphrey, W. S., Introduction to the Team Software Process. Addison-Wesley. 1999.

[3] Humphrey, W.S., TSP: Leading a Development Team. Addison-Wesley Professional. 2005.

[4] 梅田政信, 片峯恵一, PSP/TSP による実践的な ICT 人材育成の取り組み, 情報処理学会誌, 53(10), 1084−1087. 2012.

[5] 石橋慶一, 動機づけプロセスの状態遷移モデルによる人的資源マネジメントに関する研究, 博士論文, 九州工業大学. 2013.

[6] 梅田政信, 片峯恵一, 石橋慶一, 橋本正明, 吉田隆一, ソフトウェアプロセス教育における動機づけプロセスの定式化と教育改善への応用, 信学技報, KBSE2013-10, 55-60. 2013.

[7] Leveson, N.. Engineering a Safer World. MIT press, 2012.

[8] 日下部 茂, 梅田 政信, 片峯 恵一, 石橋 慶一, ソフトウェアプロセス教育向け動機づけモデルをシステム理論に基づく STAMP/STPA により効果的に活用する手法の提案, プロジェクトマネジメント学会 2017 年度秋季研究発表大会, 2017

[9] Lawler, E. E. (安藤端夫訳), 給与と組織効率,ダイヤモンド社, 1972.

[10] 坂下昭宣, 組織行動研究, 白桃書房, 1985

[11] Thomas, J. and Leveson, N. STPA Primer ver.1 http://psas.scripts.mit.edu/home/home/stpa-primer, 2015.

[12] Willig, C. Chapter 7 Grounded Theory Methodology in Introducing Qualitative Research in Psychology, Open University Press, 2013

表 2 非安全なコントロール(指導)アクション分析

アクション	与えられないとハザード	与えられるとハザード	早すぎ,遅すぎ,誤順序でハザード	早すぎる停止, 長すぎる適用でハザード
追加説明を行う	努力を遂行に結び付けるために必要な追加の指導が与えられずハザード H1	望ましい状態遷移が起きないような不適切な方法での追加説明でハザード H1	不適切なタイミングや, 誤順序での追加説明でハザード H1	…
…	…	…	…	…
指導アクション_i	努力を遂行に結び付けるために必要な追加の指導が与えられずハザード	不適切な方法での指導のため望ましい状態遷移が起きずにハザード	準備ができていない時期の指導, 時期を逸した指導, 誤順序の指導などでハザード	継続的実施の必要がある遂行の指導を早くやめすぎるなどでハザード
…	…	…	…	…

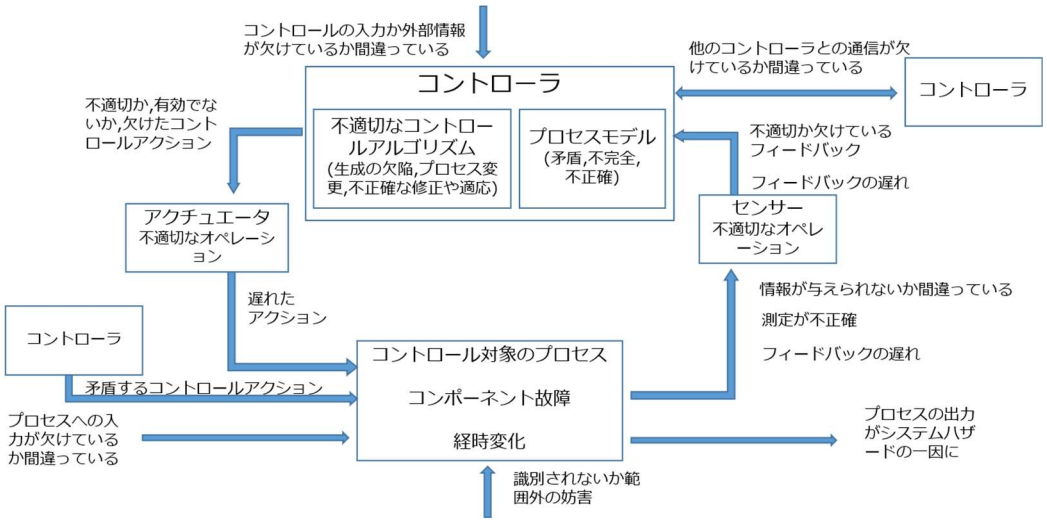


図 5 コントロールループの齟齬の原因例

現場に寄り添った教育が品質を支える ～ディスカッション教育に込めた想い～

渡辺　聡美
富士通エフ・アイ・ピー株式会社
watanabe.satomi@jp.fujitsu.com

要旨

運用部門のヒューマンエラー防止の取り組みは他業界のナレッジを取り入れ，成熟度向上を図ってきた．しかしヒューマンエラー撲滅は難題であり，時折，品質問題は発生する．また，件数こそ減少しているものの1件辺りの対応負荷は増加しており，負のコストは横ばい状態ともいえる．根底にはコミュニケーション不足やOJTが有効に機能していないという問題があった．今回の事例報告では，この状況を打破するために立ち上げた「ディスカッション教育」を紹介する．

1. 背景

安定稼働のため，当社運用部門では各種“標準“の導入，ヒューマンエラー防止プロセスの成熟度向上に関する医療，航空業界等のナレッジ活用を進めてきた．それでもヒューマンエラーは撲滅には至らなかった．また，エラー件数こそ減少しているものの，1件辺りの対応負荷は増加．負のコストの圧縮が課題となっていた．

2. 問題点

現状を深堀し，以下の問題を抽出した．「エラーには認識のズレが影響している」「障害対応力不足」「OJTが有効に機能していない」の3点である．

3. 対策（ディスカッション教育）

これらの問題はいずれも「力量」に関連する．そこで対策として，多忙な現場とは別の，品質マネジメント分野のプロフェッショナルによる，OJTを補完する教育を立ち上げた．この教育は「障害対応力向上プログラム」「コミュニケーション向上プログラム」の2点で構成している．

3.1. 工夫点

教育の質向上には以下の構成要素毎に熟慮が必要だ．カリキュラム，教育技法，受講者の意欲，講師の力量，教材の5点である．今回は特に教育技法と受講者の意欲を重視した．ディスカッションにより自分以外の多様な捉え方に気付き，視野を広げる機会とすること．そして，ただ楽しかったという一時的な満足で終わらせず，知識を使いながら，試行錯誤するという段階に進む契機とするため，演習素材は日常業務を中心に構成した．

3.2. 教育の成果

成果として以下の3点を捉えている．

1） 認識のズレの早期検出
相手を尊重した肯定的な会話が増え，認識のズレを問題発生前に検出．認識のズレに起因するエラーが減少した．

2） 障害対応力向上
教育で得た認識を各自が日常で実践することで知識が定着．分析の質向上，対応スピード改善，負のコスト削減がもたらされた．

3） プロアクティブ対応気質の醸成【想定外効果】
改善活動におけるスピーディな採否判断と対応完了．さらには改善活動での障害対応ノウハウ活用により改善提案の質向上，活性化に繋がった．

3.3. 今後の課題

残された課題は2点．OJTの質改善，ディスカッション教育をさらに定着させるための指導者の育成である．引き続き，更なる成果を創出できるよう努めたい．

参考文献

[1] 日本教育工学会，教育工学事典，実教出版，2000
[2] 浦山昌志，企業と人材 vol51 学びの近未来デザイン第 6 回現場でどう教えるか，産労総合研究所

勉強会を活用した組織成長モデル ～活動 2 年目の成果と課題～

伊藤 修司
SCSK 株式会社
Shuuji.Itou@scsk.jp

山口 真
SCSK 株式会社
Makoto.Yamaguchi@scsk.jp

豊田 圭一郎
SCSK 株式会社
Keiichirou.Toyoda@scsk.jp

要旨

1. はじめに

ソフトウェアシンポジウム 2017 にて紹介した,「参加型勉強会を活用した組織成長モデルの事例報告」について, 活動 2 年目で組織はどう変わったのか, 継続的な活動による成果と課題を事例として報告する.

2. 事例概要

2. 1「参加型勉強会」について

座学によるスキルや知識の習得を目的とした「通常の勉強会」とは異なり, グループワーク中心でアウトプット重視の勉強会である. 物事の捉え方や考え方をバージョンアップする気づきやヒントを得るための場で, 参加者全員が主役となる演出がポイントである.

2. 2「参加型勉強会」運営上の改善点

1 年目の課題を受けて, 2 つの観点で運営を改善した.

(1)オープンな体制

1 年目の会は一定の成果を収めた半面, 企画するには, 相応の知識と覚悟が必要との思い込みが生まれ, 活動そのものについて敷居が高くなりつつあった.

そこで, 今年度は, 運営メンバーを常時募集とした. この活動に興味のあるメンバーは希望すればいつでも運営に携わることができるようにした. 結果, 入社 5 年未満の若手を中心にとする 13 名で勉強会の企画・運営を進めることができた.

(2)勉強会のアウトプットを明示的にプロジェクトで活用

1 年目の活動を通じて, ふりかえり [1]だけでは, 想いやアイデアをプロジェクトにつなぐのは難しいというのがわかった. そこで, 予め勉強会のアウトプットを, プロジェクトで活用する前提でテーマを設定した.

例)A プロジェクトで考慮したいテスト観点を考えよう！

学びを応用し実行する場 [2]を予め設定することで, 2 つのプロジェクトのテストシナリオ作成にこのアウトプットを活用することができた.

2. 3. 「参加型勉強会」の成果

(1)人材育成との連動

年間 4 回開催のうち, 前半の 2 回は若手メンバーが企画からプレゼンまでを担当した. 若手メンバーにとっては, 自らが主体となって周囲を巻き込みながら仕事を進める予行演習となった. また, 中堅以上のメンバーは, 若手のプレゼン能力の高さを目の当たりにして, よい意味での脅威を感じる機会となった.

(2)プロジェクトの品質向上に寄与

プロジェクトにおけるテスト観点をワークのテーマとして, そのアウトプットをテストシナリオ作成の観点に活用した. シナリオテストの不具合検出率が従来プロジェクトよりも向上した.

(3)参加率の向上と学ぶ機会の増加

(1)人材育成との連動や(2)プロジェクトとの連動を強化することで, 勉強会への参加率が1年目よりも向上した. また, メンバーの退職や異動の際は, 勉強会形式で引き継ぎを進めたり, ベテラン層が若手に勉強会形式で知見やノウハウを伝えたりする機会が従来よりも明らかに増加することとなった. 優秀な講師は社内にいる [3]ということを改めて再確認した.

2. 4 今後の課題

今後の課題として考えているのは 3 つ

①この活動を変化させながら継続するための工夫

②人材育成活動との更なる連携

③コミュニケーション量と質の可視化

2 年目を終えて, 活動を継続することの重要性を改めて実感した. 今後も活動を継続することで、業界全体に有益な成果、知見として発表できるように努めたい.

参考文献

[1] これだけ! KPT 天野 勝著 すばる舎 ISBN978-4-7991-0275-6

[2] ファシリテーション入門 堀 公俊著 日経文庫 ISBN: 978-4-532-11026-0

[3] WORK RULES！ ラズロ・ボック著 東洋経済新報社 ISBN: 978-145-553484-5

要求獲得のためのヒアリングにおけるゴール指向要求分析の活用 ～「ゴール指向 Lite」の提案～

菅原 扶
株式会社インテック
sugahara_mamoru@intec.co.jp

山崎 哲
テックスエンジソリューションズ株式会社
yamazaki.satoshi.e4@tex-sol.com

室井 義彦
DIC株式会社
yoshihiko_muroi@ma.dic.co.jp

石川 冬樹
国立情報学研究所
f-ishikawa@nii.ac.jp

山口 俊彦
テックスエンジソリューションズ株式会社
yamaguchi.toshihiko.k9@tex-sol.com

栗田 太郎
ソニー株式会社
taro.kurita@sony.com

要旨

我々は，ソフトウェアシステム開発プロジェクトにおける要求定義での課題解決のために，新たな方策「ゴール指向 Lite」を提言することにした．従来からある要求獲得手法の「ゴール指向要求分析」の本質だけに注力することで，迅速かつ簡易に実施できる方策として「ゴール指向 Lite」を創出した．

実験として仮想の業務システム開発プロジェクトにおける要求定義での「ゴール指向 Lite」適用有無を比較検証したところその適用優位性が確認できた．

1. はじめに

ソフトウェアシステム開発プロジェクト（以下，プロジェクト）においては，例えば要求の抜け漏れなど，要求定義における課題に対処することが重要である．要求工学知識体系 REBOK（Requirements Engineering Body Of Knowledge）によれば，共通知識カテゴリにおける 8 つの知識領域のうち，要求定義に直接必要な知識領域は要求獲得，要求分析，要求仕様化，要求の検証・妥当性確認・評価の 4 つのプロセスである[1]．

我々は，このうち要求獲得プロセスに着目した．なぜならば要求獲得とは「顧客を含むステークホルダを明らかにし，会議やインタビューなどを通して要求を引き出す技術に関する知識」と定義されており，前述の課題解決に効果があると考えたためである．

本研究では，要求の構造化と分析の手法として注目される「ゴール指向要求分析」[2][3]を要求獲得において活用することに着目し，要求定義における有効性について研究を行う．

以下本論文の構成を述べる．2 章でゴール指向要求分析の特徴とその課題を示す．3 章では我々の提案する手法の詳細について説明する．4 章ではその手法の有効性検証のために実施した実験詳細を示し，5 章で実験結果について考察する．6 章では，まとめとして本研究の考察と今後の課題について述べる．

2. ゴール指向要求分析における課題

2.1. ゴール指向要求分析

ゴール指向要求分析では，ゴールとはシステムが満足すべき状態であると定義されている．また，システム要求とはゴールを達成するための手段であると定義されている．プロジェクトにおいて達成すべき利用者のゴールにこそ最も着目すべきであり，ゴールを分解・詳細化（サブゴール化）して達成手段を明確に定義したものをシステム要求と見なすということである（システム要求化）．これにより，システム要求に関する「何のためにそれが必要なのか」が明確になり，要求分析における議論や妥当性確認，要求変更時の追跡がそれぞれ行いやすくなる．

ゴール指向要求分析の具体的な手法としては，KAOS[4]， i*[5]，NFR [6]などが知られている．それぞれ，コンポーネントへの責務割当，ステークホルダ間の依存

関係分析，非機能要求の分析というように，手法独自の記法と分析方法が定められている．また i* と NFR で用いた記法を基に， User Requirements Notation （URN）[7]記法が国際電気通信連合（ITU）にて標準化されている．また Goal Structuring Notation（GRN）記法は，astah*など多くのツールでサポートされている [1]．

本研究においては，これらの様々な手法や記法に共通する部分として，ツリー構造によりゴール間の依存関係をモデリングする点に着目する．ゴール間の依存関係とは，「あるゴールを達成するために，（いくつかの）サブゴールの達成が必要になる」という関係である．ゴールモデルの概念図を図 1 に，具体例を図 2 に示す．

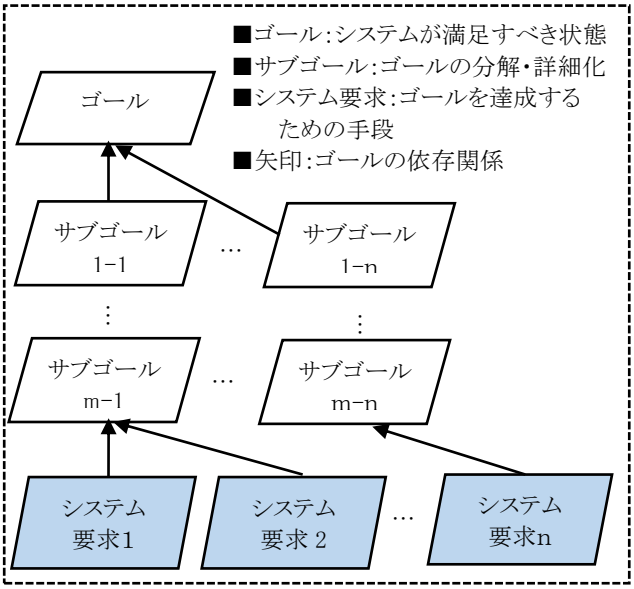


図 1. ゴールモデルの概念図

ゴールモデルにおいては，上位ゴールが下位ゴールに AND/OR 関係を用いて分解されている．上位ゴールになればなるほど抽象性が高くなり，ゴール分解の終了基準は，すべての下位ゴールに対してその達成手段，すなわちシステム要求が特定されることである．

ゴール指向要求分析の要求定義への適用，すなわちツリー構造によるゴール間の依存関係をモデリングすることで，すべてのゴールに対するすべての達成手段（システム要求）を特定でき，それらを明示的に可視化することができる．

[1] http://astah.change-vision.com/ja/product/why-gsn.html

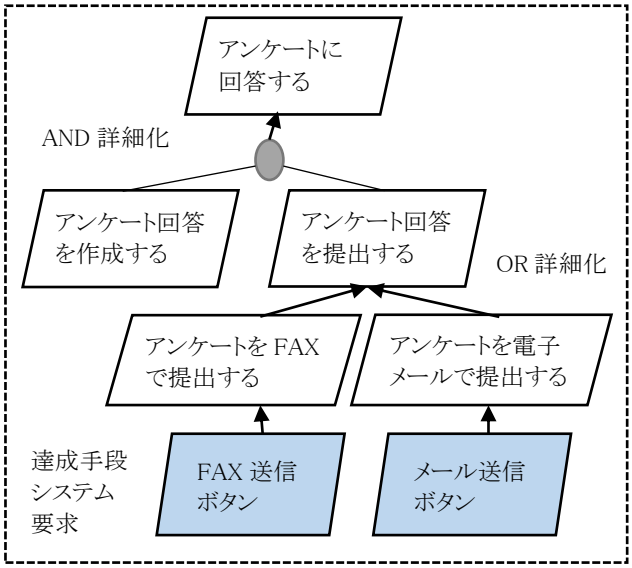


図 2. ゴールモデルの例

[2][3]における議論などを踏まえ，本論文においては，ゴール指向要求分析により期待される効果を，以下の6つの観点で論じる．

(1) 要求獲得における抜け漏れの防止：「上位ゴールに対して十分な下位ゴールが挙げられているか？」という問いに対応する情報が明示的に示されるため．

(2) 要求の必要理由の明確化： 各ゴールに対して，その上位ゴールが明示的に示されるため．

(3) 要求獲得における矛盾や誤りの排除： セキュリティと使用性など相反しがちな上位ゴールから得られた要求同士の矛盾を探すなど，ゴールモデルで明示された情報をきっかけに矛盾や誤りに気づくことがあるため．

(4) 要求の重要度・優先度の把握： 上位ゴールの重要度・優先度を明確にし，そのために特に貢献度が高く必要となるサブゴールを明確にするなど，重要度・優先度の伝搬をツリー構造上で明示的に検討できるため．

(5) ヒアリング時における暗黙のゴールに対する気付き： ヒアリング時に得られた情報において上位ゴールが不明確であった場合，ゴールモデル構築時に気づくため．

(6) ステークホルダ間での認識共有の促進： 得られたゴールモデルは，どうして何を作るのかの情報を表現した文書として利用できるため．

2.2. 課題

しかし，ソフトウェア開発の実務者である我々の経験では，ゴール指向要求分析の活用においては特に表 1 の課題があると考える．2.1 章で述べたように様々な手法においては，ツリー構造によるゴール依存関係の表現に加えて，多くの記法（モデリング要素）や分析の手法・観点が追加されている．それらは要求定義のあり方について一つの方向性を示したものであるが，多くのモデリング要素・分析観点を適切に理解し，使いこなしていくことは難しい．また現状の開発プロセスから工程のとり方が変わる点についても，後工程を短縮できるとしても，大きな変化を要する（表 1 No.1）．

一方で，抽象度が高いゴールを対象とするため，その原則や指針（例えばゴール分解の観点[4]など）も抽象的となってしまう．加えて URN や GSN は表記として標準的なものを定めたものであり，その上での分析の手法・観点は定義していない．このため表記として一通り埋めることに気が向いてしまったり，原則なく何となく埋めてしまったりすることが生じうる（表 1 No.2, 3）．

表 1. ゴール指向要求分析における課題

No	課題	内容	理由
1	時間制約	分析実施や手法の習熟に時間がかかる	・基本的にシステム全体のゴールモデルを書くことが前提となっており，ツリー内の記述に曖昧さを残せず，明示的に記述せざるを得ないため ・各手法の記述ルールの理解に時間がかかるため
2	属人性	分析結果が個人の経験や知識量に依存してしまう	・あくまでツリー構造による記述方法のみが定義され，記述内容は定義されておらず，結果，記述の自由度が高いため
3	本来目的の喪失	ツリーを完成させることに意識が働き，本質的な要求分析という目的を見失いがちとなる	・見た目の記述の枠組みに目が行きがちで，かつわかりやすい終了基準である記述の完成に目が向いてしまうため

3. ゴール指向 Lite

3.1. アプローチ

要求定義におけるゴール指向要求分析手法の有効性は認識するものの，実務者が取り組む上では 2.2 章で挙げた表 1 の課題が存在する．これに対し我々は，シンプル化・実用化することに主眼を置いた手法として「ゴール指向 Lite」を提案する．ゴール指向要求分析の様々な手法・記法に共通する部分のみを扱うことで，容易に習得・活用ができるようにする（表 1 課題 No.1）．またモデル全体を完成させるという意識よりも，要求獲得のための「問い」の観点に意識を向ける（表 1 課題 No.2, 3）．

ゴール指向要求分析については，一通り集めたゴールの情報を構造化，分析する際に用いる想定であることも多い．これに対して「ゴール指向 Lite」は，要求獲得のための問いを得ることを主眼においている．

3.2. 手法詳細

ゴール指向 Lite は極めてシンプルである．既に獲得済の要求に対し，2 つの手順を実施するだけである．図 3 にゴール指向 Lite の概念図を，表 2 に実施手順を，具体例を図 4 に示す．手順において重要なのは，表 2 の中の洗い出しの観点を自問することにより，導出対象を獲得するということである．実施にあたっては，思いつく限りの上位ゴールや他要求を複数件導出して構わない．上位ゴールとは，ゴールの目的や理由，必要性を問うたものである．導出観点の決定根拠は，複数あるゴール指向要求分析手法に共通する最も核となる観点だと考えたためである．また 1 段上位までとしたのは，手順をシンプルにすることで導出対象の獲得を容易にするためである．

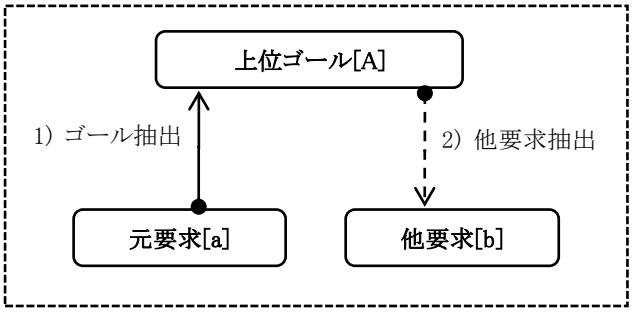


図 3. ゴール指向 Lite 概念図
（上位ゴールからのアプローチ）

表 2. ゴール指向 Lite 実施手順
(上位ゴールからのアプローチ)

No	手順	導出観点	導出対象
1	獲得済の要求から1段上位のゴールを導出する	なぜ要求[a]を実現する必要があるのか	上位ゴール[A]
2	導出した上位ゴールに紐付く他の下位要求を導出する	要求[a]を実現することだけで，上位ゴール[A]が実現するか	下位要求[b]
		上位ゴール[A]を実現するために必要なことは，要求[a]以外にないか	
		要求[a]以外で，上位ゴール[A]を実現することができないか	

表 3. ゴール指向 Lite 実施手順
(問題・リスクからのアプローチ)

No	手順	導出観点	導出対象
1	要求[a]が実現できない場合の問題・リスクを導出する	要求[a]を実現できない場合，どのような問題・リスクがあり得るか	問題・リスク[A’]
2	導出した問題・リスクに紐付く他の要求を導出する	要求[a]を実現することだけで，問題・リスク[A’]が起こらないか	他要求[b]
		問題・リスク[A’]を起こさないために必要なことは，要求[a]以外にないか	
		要求[a]以外で問題・，リスク[A’]を起こしてしまうことがあるか	

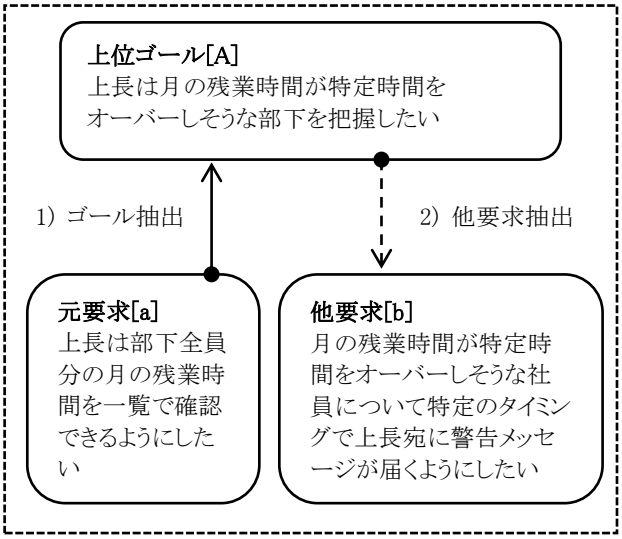


図 4. ゴール指向 Lite 活用例
(上位ゴールからのアプローチ)

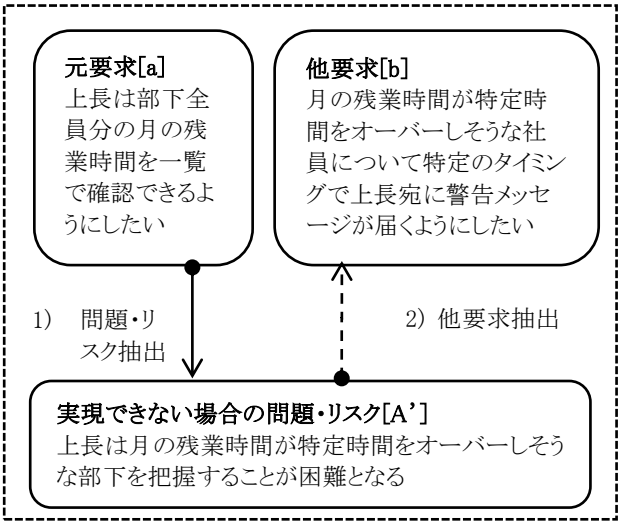


図 5. ゴール指向 Lite 活用例
(問題・リスクからのアプローチ)

また，上位ゴールがうまく導出できない場合の代替手順としては，問題・リスクからのアプローチを行う．表 3 にその手順を，図 5 に具体例を示す．

4. 実験

ゴール指向 Lite のヒアリングにおける有効性を検証するため，勤怠管理システム構築の仮想プロジェクトによる実証実験を試みた．利用者(ヒアリングされる者)と分析者(ヒアリングする者)に分かれ，さらに分析者は，ゴール指向 Lite の手法を用いなかった場合(分析者 A)と用いた場合(分析者 B)に分かれ，それぞれ要求獲得のためのヒアリングを実施した．最終的にヒアリング時の質問内容と獲得した要求を比較・分析することでゴール指向 Lite のヒアリングにおける有効性を検証した．

利用者より開示された要求リストを基に，分析者 A は，質問を思いつくままにリストアップしていくようなやり方をとり，分析者 B は，表 2，表 3 の実施手順に従いゴール指向 Lite を実施した．

4.1. 実験方法

今回の実証実験で初期開示した要求リストを表 4 に，実験の手順概要を表 5 に示した．

表 4. 初期開示した仮想プロジェクト
(勤怠管理システムの構築)の要求リスト

No	要求内容
1	各社員が毎日登録する勤怠入力画面が使いづらく，登録に時間がかかってしまっていることを改善したい
2	とはいえ，残業時間規制の管理目的より，一月分をまとめて登録するようなことはさせたくない(原則社員に毎日登録してもらうことを想定している)
3	入退室 ID などから自動登録するような厳密な管理は柔軟性を欠くためやりたくない
4	2018 年 4 月 15 日には全社員が登録可能とする
5	予算は 3,000 万円
6	現状，社内ネットワークに接続した PC からしか利用できないが，スマホからでも利用できるようにしたい
7	三六協定遵守を強化すべく，社員に注意を促したい
8	作業 Work や PJ，部門の間接費と作業時間を紐付けることで，当月発生した費用種別を分類したい
9	社員数 1,000 名，20 年分以上のデータ保持が可能であること
10	月末など複数ユーザーが同時に利用しても耐えられること

表 5. 要求獲得ヒアリング実証実験の手順概要

No	実験 手順	所要 時間
1	利用者(1 名)が作成した要求リストのうち 10 件の情報を分析者(各 2 名)に開示	30 分
2	開示された情報を基に，ゴール指向 Lite の分析を実施 ※分析者 B のみ	30 分
3	開示された要求リストを基に分析者 A，分析者 B がそれぞれヒアリング実施	60 分
4	最終的に分析者 A，B が獲得した要求リストとヒアリング時の質問内容を比較	30 分

4.2. 実験結果

実証実験にて，ゴール指向 Lite 適用による効果の分析結果を表 6 に，分析者 A，B がヒアリングして獲得できた要求のうち，分析者 B が獲得した要求リストの一部抜粋を表 7 に示す．

また実験時には，比較対象のために従来のゴール指向要求分析手法実施者も設定して分析を行った．十分な経験を持った分析者により実施しているが，30～60 分の時間制約内ではツリーを完成させられず，十分な効果をあげることはできなかった．これにより，表 6 の結果としても示せていない．

表 6. ゴール指向 Lite 実施による効果分析

ゴール指向 Lite 適用有無	総質問件数(件)	ゴール指向要求分析に期待する6つの効果												機能仕様または現状の確認	
		(1)抜け漏れの防止		(2)必要理由の明確化		(3)矛盾や誤りの排除		(4)重要度・優先度の把握		(5)暗黙のゴールに対する気付き		(6)認識共有促進			
A(無)	42	1	2.4%	4	9.5%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	**37**	**88.1%**
B(有)	42	0	0.0%	**13**	**31.0%**	1	2.4%	**2**	**4.8%**	**5**	**11.9%**	0	0.0%	21	50.0%

表 7. 実験にて分析者 B(ゴール指向 Lite 分析有)が獲得した要求リストの一部抜粋

No	獲得要求内容	表 6 との関連
2	プライベートのデバイス(携帯・PC)からは利用できないようにしたい	(5) 暗黙のゴールに対する気付き
8	承認差戻しなどが頻繁に起こらないことが望ましい	(5) 暗黙のゴールに対する気付き
15	三六遵守のために社員の残業状況を見える化したい	(2) 必要理由の明確化
16	三六協定違反者にはメールで通知する機能が欲しい	(2) 必要理由の明確化
17	三六協定に違反しそうな人には,アラートが上がる機能が欲しい	(2) 必要理由の明確化
18	三六遵守のアラート機能は該当社員だけでなく, 上長にもあがるようにしたい	(2) 必要理由の明確化
19	三六遵守のアラートは閾値設定で管理できるようにしたい	(2) 必要理由の明確化
20	さらに三六遵守のアラート前に, 後何時間のような情報が伝わるようにしたい	(2) 必要理由の明確化
23	(適度な柔軟性を欠くことがなければ) 自動登録機能があり, それを編集できる仕組みとしての検討の余地はある	代替案の提案
38	従業員の入力負荷削減を最優先で重視(1 人あたり 1 日 4, 5 分⇒1, 2 分にしたい)	(4) 重要度・優先度の把握

5. 考察

5.1. ゴール指向 Lite 適用の効果(ゴール指向要求分析の 6 つの期待効果に対する)

2.1 章であげたゴール指向要求分析の 6 つの期待効果に対し, ゴール指向 Lite での適用効果を 4.2 章の表 6 も踏まえて下記の通り考察した.

(1) 要求獲得における抜け漏れの防止

一定の効果があると期待される. なぜならば, 一般のゴール指向要求分析同様に, 各ゴール分割の十分性についての問いがなされるためである. ただし最上位のゴールまで多段階のゴールモデルを考えるわけではないため, 取り上げた部分の局所的な効果にとどまる.

表 6 における結果においては効果が現れていない. これは実験設定において当初のゴール集合が簡易的であり, これからヒアリングを繰り返す段階であったためだと考えられる. ヒアリングを数回繰り返すことにより効果が出てくるものと推測できる.

(2) 要求の必要理由の明確化

大きな効果があると考えられる. なぜならば, 一般のゴール指向要求分析同様に, 各ゴールの上位ゴールを問うためである.

例えば, 表 7 における要求 No.15 から 20(三六協定に関連する各要求)の獲得にその効果が表れている. 必要理由の明確化は, そもそもゴール指向 Lite の手順内容に包含されており, 期待通りの結果が出ているといえる.

(3) 要求獲得における矛盾や誤りの排除

効果はないと考えられる. 矛盾や誤りの排除は複数の要求を照らし合わせるなど, 全体を俯瞰して見ることで気が付く内容であるため, ゴール指向 Lite では難しい. 表 6 の実験結果においても特に効果は現れていない.

(4) 要求の重要度・優先度の把握

効果は直接的にはないと思われる. なぜならば, ゴール指向 Lite においては重要度・優先度を考えるような指示をしていないためである. ただし重要度・優先度を考える姿勢を分析者がとった場合, その助けとなりうる.

実験結果における具体例として，表 7 における要求 No.38（入力負荷軽減が最優先）の獲得があげられる．複数抽出した上位ゴールそれぞれに対して質問することで，優先度の高い上位ゴールが判断でき，関連する下位要求が明らかとなるためである．一定の期待効果が得られたと言える．

(5) ヒアリング時における暗黙のゴールに対する気付き

一定の効果があると考えられる．ゴール指向要求分析同様に問いかけの観点を提供しているためである．

表 7 における要求 No.2（個人のデバイスは使用不可という社内ルール）及び No.8（承認時の差し戻し頻発対策）の獲得などが該当する．ユーザーが当たり前と思っている内容は，要求として提示されないこともしばしば起こり得るが，上位ゴールから下位要求を抽出する段階で気づきを多少は促す効果があると考えられる．

(6) ステークホルダ間での認識共有の促進

利用方法に依存すると考えられる．ヒアリング時に一緒に付箋を貼りながら上位ゴールを導出するような双方協働での作業実施ができれば，より期待する効果があげられるのではないかと推測できる．

また，表 7 からは確認できないが，「代替案の提案」が可能となるという効果も実験の結果から見てとれた．これは，上位ゴールを分析した結果，それらから代替案を検討・提案するというアプローチが可能となる，という効果である．実験結果として，4.2 章の表 7 における要求 No.23（自動登録機能）の獲得にその効果が表れている．初期開示要求では，「自動登録機能は不要」であったが，「柔軟性担保」という上位ゴールを維持できれば，「自動登録ありかつ編集可能とすることならば検討の余地がある」という回答を獲得できている．

5.2. ゴール指向 Lite 適用の効果（ゴール指向要求分析の 3 つの課題に対する）

ゴール指向 Lite の適用に対し，2.1 章の表 1 の課題が解消できているかどうかを表 8 にまとめた．ただし，1 度の実証実験しか行えていないため確定的な結果とするためには，実験を重ねる必要があると考えている．

表 8. ゴール指向要求分析の課題への対応

No	課題点	ゴール指向 Lite を用いた結果
1	時間制約	実施手順が少ないので短時間で実施することができた
2	属人性	実施手順の観点が明確であるため，分析実施者が代わっても概ね同様の結果が期待できる
3	本来目的の喪失	記載レベル（分解化・詳細化）の観点を定めてあるので，重要で本質的な要求の獲得と分析という本来目的が実施できた

5.3. 実験結果から考察されたゴール指向 Lite の特徴

(1) ステークホルダとの協働分析作業

通常のゴール指向要求分析においては，獲得要求に対する分析をいつ行うのかは特に定められていない．しかしゴールモデルが大きくなることを考えると，ステークホルダとその場で一緒にツリーを作成するような方法は現実的には難しく，要求獲得後に分析者が個別分析し，質問事項を後で作成することが通常である．一方，ゴール指向 Lite の場合，1 つの要求から導出した複数の上位ゴールをヒアリング時に確認しながら協働で分析していくことで，ヒアリングと分析を短時間かつ同時並行で進めることができる．

これより，ステークホルダとの協働分析作業という点において，ゴール指向要求分析をすべて完了させるよりも，ゴール指向 Lite は，実施におけるハードルが低くなると考えられる．また，その場で実施完了を目指すので上位ゴールと要求に対するステークホルダとの認識共有促進が期待できる．

(2) ヒアリング傾向

ヒアリング時における質問全体の特徴としては，ゴール指向 Lite 非適用の分析者 A は，「機能詳細を明確にしようとする」傾向が見られた．一方，ゴール指向 Lite 適用の分析者 B は，「目的を明確にしようとする」傾向が見られた．この傾向から，質問範囲が狭くなってしまう分析者 A は分析者 B より早い段階で質問が枯渇することが予想され，ヒアリングを複数回繰り返していくことで，分析者 A，B 間の要求獲得・分析の差はさらに大きくなっていくものと推測できる．

6. 結論と今後の展望

我々は，仮想プロジェクトへの実証実験を通して，要求獲得のためヒアリング時における手法であるゴール指向 Lite を提案し，以下の有効性を確認した．

・ゴール指向要求分析の 6 つの効果を一部引き継いでいる

(5.1 の(1)～(6)参照)

・ゴール指向要求分析の 3 つの課題を解消できる

(5.2 参照)

また，今回の実証実験は時間制約上，1 つの仮想プロジェクトを1度しか実施することができなかったため，十分な検証が行えたとは言えない．特に 5.1 章の以下の 3 点に対しては実証実験を繰り返し行うことによりその効果検証ができると考える．

(1) 要求獲得における抜け漏れの防止

(5) ヒアリング時における暗黙のゴールに対する気付き

(6) ステークホルダ間での認識共有の促進

今後，実プロジェクトにおけるゴール指向 Lite 活用も含め，あらゆる検証を継続的に行いつつ，必要に応じてゴール指向 Lite をブラッシュアップしていくことで，より実用的な手法として確立していきたいと考えている．

参考文献

[1] 飯村結香子・斉藤忍，REBOK に基づく要求分析実践ガイド，近代科学社，2015

[2] 山本修一郎，ゴール指向による!!システム要求管理技法，ソフトリサーチセンター，2007

[3] Axel van Lamsweerde, Goal-Oriented Requirements Engineering: a Guided Tour, The 5th IEEE International Symposium on Requirements Engineering (RE 2001), pp.249-262, 2001

[4] Axel van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley, 2009

[5] Eric Yu, Paolo Giorgini, Neil Maiden John Mylopoulos (Editor), Social Modeling for Requirements Engineering, The MIT Press, 2010

[6] John Mylopoulos, Lawrence Chung, Eric Yu, From Object-Oriented to Goal-Oriented Requirements Analysis, Communications of the ACM, Vol. 42 No. 1, pp. 31-37, 1999

[7] User Requirements Notation (URN) – Language Requirements and Framework, ITU-T Z.150, 2003

Applying PReP Model to a Service Development Process

Koji Kinouchi
Weathernews Inc.
kinouchi@wni.com

Yasushi Tanaka
K-plus solutions Co. Ltd., NAIST
ytanaka@kplus-solutions.com

Yasushi Ishigai, Taichi Kawai
Mitsubishi Research Institute, Inc.
ishigai@mri.co.jp, t-kawai@mri.co.jp

1. Introduction

Before trying the PReP-based RD process, we only had a basic framework for process definition for service developing process as shown on the left side of Fig.1. The lack of a more comprehensive framework resulted in dependency upon the competence of the person in charge of the project, causing inconsistency in the quality of requirement definition and risking traceability between service contents and system requirements. In order to win an entry into a new business in the EU market, we decided to implement the PReP-based RD process to improve the issues in both the service requirement and the system development processes.

2. Applied method

The PReP model[1]-based RD process is a business process modeling method with a products-based process modeling approach. The model provides business process analyzing and designing procedures and problem-cause and risk analyses.

As shown on the right side of Fig.1, the RD process using this method enables the user to design its customer's business process. Furthermore, by using dedicated modeling tools, system requirements will be automatically generated from the designed business process model.

3. Target project

We decided to develop a new service for shipping business in Europe as a target project. We set the following three points as objectives:

- Improve our service development processes
- Develop a new service to solve the customer's shipping business issues and obtain a contract
- Ensure traceability between service contents and system requirements to improve the quality of service and service development process

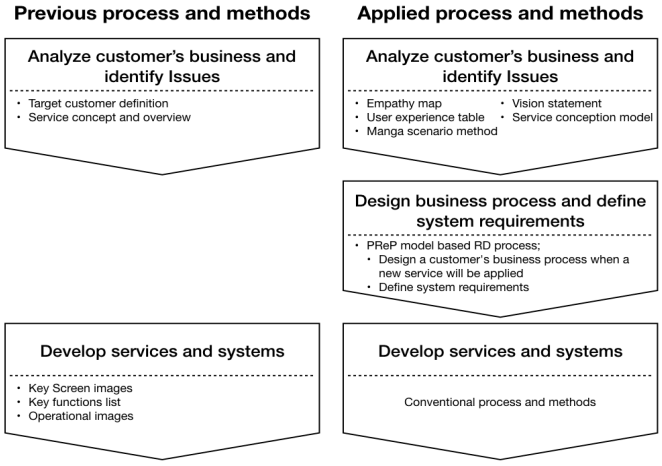


Fig.1 Difference between the previous method and the applied new method.

4. Result

By introducing the new PReP-based RD method, we saw the following improvements in the service development process:

- Comprehensiveness of requirements
- More reliable feasibility study for building the service
- Better communication with system developers
- Broader participation from relevant business personnel

As a result, we were able to win a contract with the new service we developed.

On the other hand, because our organization still relied on the competence of people in charge, managing RD process and applying the method after this case were difficult.

References

[1] Yasushi Tanaka, Yoshiki Goko, Kunio Mitsui: Appling PReP model to requirement development process, SS 2015, pp.162－171, 2015.

要求記述のスキル不足に対するSRS記述ガイドの有効性評価

不破　慎之介
(株)デンソークリエイト
fuwa_s@dcinc.co.jp

山田　ひかり
(株)デンソークリエイト
hikari@dcinc.co.jp

蛸島　昭之
(株)デンソー
akiyuki_takoshima@denso.co.jp

要旨

ソフトウェア開発の現場では人の経験や知識に頼った開発が通用しなくなりつつあり，人に依存しない開発の仕組みが求められている．ソフトウェアの開発経験が浅い担当者(以降，非熟練者と呼ぶ)であっても一定水準の品質で要求仕様書(SRS)を記述できるようにするための仕組みとして，SRSの記述ガイドに着目する．

本経験論文では，SRSの記述ガイドを参照する非熟練者と，ガイドを参照しない熟練者のSRSの品質の比較評価を行う．評価は，評価基準に基づいた定量的評価と，第三者の主観による定性的評価の両面で評価する．非熟練者のSRSの品質が熟練者のものと比べ遜色ない結果となっていることで記述ガイドの有効性を証明する．

1. はじめに

車載ソフトウェアの開発では，開発の短期化やコスト削減を背景として，業務委託比率の増加や人員の入れ替わり周期が短期化することで，これまでのような経験に頼った開発が通用しなくなってきている．そのため，熟練者の経験知(ノウハウ)を非熟練者が活用できるような仕組み作りが大きな課題となっている．特に，上流のソフトウェア要求分析工程においては，担当者に求められる経験知が多く，後工程への影響も大きい．本論文では，ソフトウェア要求分析工程において熟練者の経験知を活用する仕組みを策定し，その効果について評価した結果を述べる．

本論文は，次のような構成になっている．2章では本研究を行うに至った背景を述べる。3章では本研究に関連する研究について述べる． 4章ではSRS記述ガイドの内容についてまとめる．5章では記述ガイドの有効性を確認するための評価内容を説明し，評価の結果は6章，評価から得られた考察については7章にそれぞれ示す．最後に8章で本論文としてのまとめと今後の課題について述べる．

2. 研究の背景

筆者が以前属していた組織ではSRSの目次項目を定義したSRSテンプレートが利用されていた．しかしながら，このような目次項目のみが定義されたテンプレートは，目次の名称から意図が読み取れなかったり，具体的にどのようなことを記述すれば良いのか分からないという問題があった．特に非熟練者の場合はこのような問題を独力では解決できず，結局は質疑やレビュー等により熟練者から直接追加の情報を得ることで解決を図っていた．そのため，熟練者の経験知を仕組みに取り込めているとは言えない状態であった．このような問題を踏まえ，目次項目だけでなく，目次の意図や記述すべき内容までを解説したSRS記述ガイドが必要であるとの結論に至った．

3. 関連研究

自動車ソフトウェア要求仕様書のインスペクション方法に関する研究[1]の中で，SRSの品質の定量的な評価方法と参照SRSについて述べられている．参照SRSとは，SRSに含むべき要素とその構成を定めたものである．

4. 提案手法

本論文では，参照SRSに対して，非熟練者が熟練者の経験知を活用できるような情報を加えることでSRS記述ガイドを策定する．以降の各節では，本研究で有効性を評価したSRS記述ガイドの詳細を述べる．

4.1. 記述ガイドの解説項目

SRS記述ガイドは，参照SRSの目次項目毎に次の項目を解説している．

- 項目の説明
- 項目が必要な理由
- 書くべき内容
- 書くべきではない内容
- 項目がない場合のリスク

「項目の説明」では，目次項目の目的や，何を書くかを説明し，その目次項目の全体像が分かるようにする．「項目が必要な理由」では，その目次項目が SRS に記述されていることによる嬉しさを説明し，その目次項目の必要性，重要性を理解できるようにする．「書くべき内容」はその目次項目の目的を達成するために必要な記述項目を定義することで，記述の抜け漏れを防止する．「書くべきではない内容」では，非熟練者が起こしやすい誤りを説明することで，過去に発生した誤りの再発を防止する．

4.2. 記述ガイドの目次項目

本論文で使用した記述ガイドの目次項目を以下に示す．

1 はじめに
- 1.1 目的
- 1.2 適用範囲
- 1.3 用語・略称の定義
- 1.4 参考文献
- 1.5 概要

2 全体像
- 2.1 場所への適合の要求
  - 2.1.1 システムインタフェース
  - 2.1.2 ユーザインタフェース
  - 2.1.3 ハードウェアインタフェース
  - 2.1.4 ソフトウェアインタフェース
  - 2.1.5 通信インタフェース
  - 2.1.6 メモリ制限
  - 2.1.7 操作
  - 2.1.8 場所への適合の要求
- 2.2 ソフトウェアの機能
- 2.3 ユーザ特性
- 2.4 制約条件
- 2.5 前提と依存関係
- 2.6 要求の割り当て

3 要求の詳細
- 3.1 外部インタフェース
- 3.2 機能
- 3.3 パフォーマンス要求
- 3.4 論理データベース要求
- 3.5 設計上の制約
- 3.6 ソフトウェアシステムの属性

4 補足情報
- 4.1 目次と索引
- 4.2 付録

図1：参照 SRS の目次項目

4.3. 記述ガイドの例

例として，「ソフトウェアシステムの属性」の目次項目の信頼性に関する記述ガイドを以下に示す．

表1：ソフトウェアシステムの属性（信頼性）の記述ガイド

項目の説明	信頼性という用語は，元来は，故障等により製品が使用できなくなる状況に対応して，そうした事態が軽減される程度として考えられた．ソフトウェアでは，故障の原因は機械類のように摩耗等ではなく，主として開発過程での不具合，利用環境の変化，セキュリティ上の攻撃などから生じるので，特有の軽減策が必要となる． なお，信頼性と類似の言葉に「ディペンダビリティ」という考え方がある．
項目が必要な理由	車載ソフトウェアは，リリースされてから長時間稼動し続け，人命に関わる制御を行う場合もある．そのため，高い信頼性が求められる． 予めソフトウェアに求める信頼性を規定し，それを満たすように開発，テストを行うことで，運用後もソフトウェアが適切に稼動し続ける可能性を高めることができる．
書くべき内容	例として以下のものが挙げられる． ・許容しなければならない（発生してもソフトウェアが適切に稼動しなければならない）障害や環境 ・特定の悪条件下においてソフトウェアが正常に稼動しなければならない割合（耐ノイズ性など） ・テストのカバレッジ
書くべきではない内容	根拠が無い（FMEA や FTA 等の分析による裏付けが無い）要求は次のリスクがある． ・障害に対して適切に対応できないリスク ・実際には起こり得ない障害に対応してしまうリスク 製品の慣習のみに基づいた要求は次のリスクがある． ・ハードウェアの高信頼化やシステム構成の変化等により，既に必要なくなった要求となっているリスク ・ハードウェアの高機能化により，ハードウェアで対処した方が品質やコスト面が良くなっていることを見落とすリスク ・新たに発生する恐れのある障害に対処できないリスク
項目がない場合のリスク	・通常想定される障害が発生した場合や悪条件の下で利用された場合に，ソフトウェアが正常に動作しない． ・過剰な多重化やフェールセーフ設計により，ソフトウェアの性能や保守性が損なわれる．

上記の例から分かるように，SRS 記述ガイドの内容は，熟練者であれば知識や経験から既に理解していることがほとんどである．しかしながら，非熟練者はガイドが無ければ「ソフトウェアシステムの属性」や「信頼性」という名前

だけを見て内容を想像しなければならない. そのため, ソフトウェアに必要な属性を見逃したり, 要求を具体化できないリスクがある.

5. 評価内容

本章では SRS 記述ガイドの有効性の評価の内容を説明する. 評価は, SRS 記述ガイドを参照するグループ(実験群)と参照しないグループ(対照群)それぞれで SRS を作成し, SRS の品質を比較する方法で行う.

5.1. 被験者の情報

熟練者と非熟練者の経験の差が SRS 記述ガイドによって小さくなっていることを確認するため, ガイド有グループには開発経験年数 5 年以下の担当者 2 名, ガイド無グループには開発経験年数 10 年以上の担当者 1 名を割り当てている.

5.2. 評価の入力情報

SRS の入力となる上位要求は同一のものを使用する. 組み込みソフトウェアの特徴を持った小規模なソフトウェアを題材としている. また, 上位要求にはソフトウェアが搭載されるデバイスの情報とシステムの運用イメージが示されているのみであり, ソフトウェアへの具体的な要求事項については一切無い状態としている. これにより, 要求の網羅性や分析の深さが担当者の能力に大きく依存する(能力の差がより顕著に SRS に現れる)ようにしている.

ガイド有グループの入力となる SRS 記述ガイドは車載ソフトウェア向けに策定したものを用いる. 車載ソフトウェアは組み込みソフトウェアの特徴を包含しているため, 策定した SRS 記述ガイドをそのまま利用できる.

5.3. 評価の手順

ガイド有グループには上位要求と SRS 記述ガイドを渡し, ガイド無グループには上位要求のみを渡す. その後, 各担当者は独力で SRS を作成する. その際, 費やした時間が品質に影響を与えないよう, 両グループで同一の制限時間を設けている. なお, ガイド有グループには, SRS 作成時間とは別に SRS 記述ガイド読解のための時間を与えている.

5.4. 品質の評価方法

定量的な評価では, 第三者インスペクション方法[1]を用いて SRS のドキュメント品質を評価する.

定性的な評価は, いずれのグループにも属さない組み込みソフトウェアの熟練者 1 名が評価者となって行う. 評価者は, 作成者を伏せられた状態でパスアラウンド形式のレビューを行い, 品質特性毎に 0～100 点で採点する. 品質特性は, 第三者インスペクション方法[1]と同様の品質特性を採用する. ただし, こちらはドキュメント品質と要求品質の両方を評価する. 品質特性の対応付けと, 定性的な評価で使用する採点表を以下に示す.

表2:品質特性の対応付け

品質特性	ドキュメント品質特性	要求品質特性
正当性	責任追跡性	正当性
無曖昧性	明確性	N/A
検証可能性	N/A	検証可能性
完全性	記述網羅性	要求網羅性
無矛盾性	N/A	無矛盾性
変更容易性	変更容易性	N/A
追跡可能性	追跡可能性	N/A
達成可能性	N/A	達成可能性

表3:ドキュメント品質の採点表

品質特性	観点	スコア
責任追跡性	ソフトウェアの目的が記述されているか	
明確性	要求が一意に識別できる表現で記述されているか	
記述網羅性	SRS で記述すべき内容が全て記述されているか	
変更容易性	要求の変更がしやすい文書であるか	
追跡可能性	要求の根拠が記述されているか	

表4:要求品質の採点表

品質特性	観点	スコア
正当性	要求が目的を達成しているか	
検証可能性	要求に対して現実的に評価可能な手続きが存在して検証できるか	
要求網羅性	ソフトウェアが持つべき要求が過不足無く SRS に含まれているか	
無矛盾性	要求間で矛盾がないか	
達成可能性	要求が現実的に実現可能であるか	

6. 評価結果

本章では, 作成した SRS を評価基準に基づいて定量的に評価した結果と, 第三者の主観により定性的に評価した結果を示す.

6.1. 定量的な評価結果

各グループの担当者が作成した SRS に対して第三者インスペクションを行った結果を以下に示す.

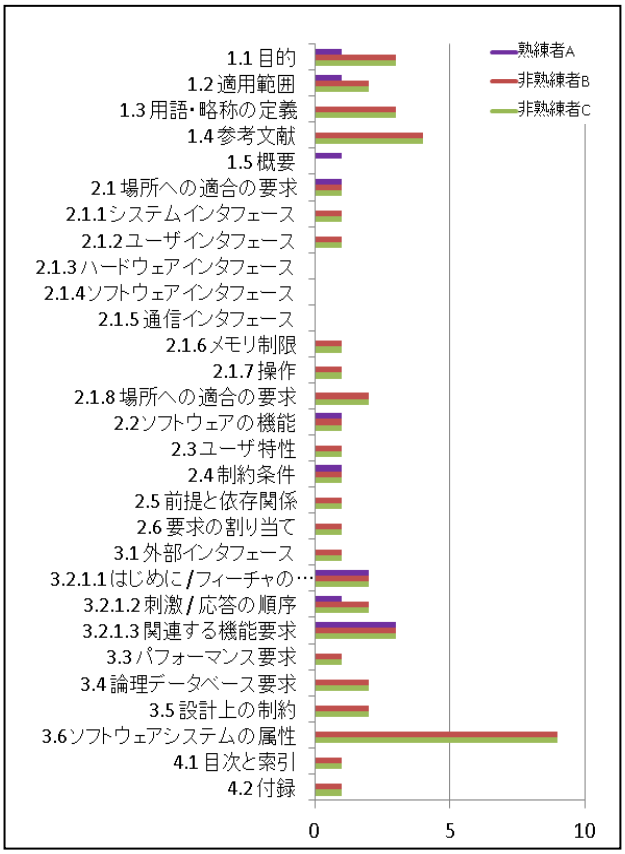


図2: 目次項目ヒストグラム

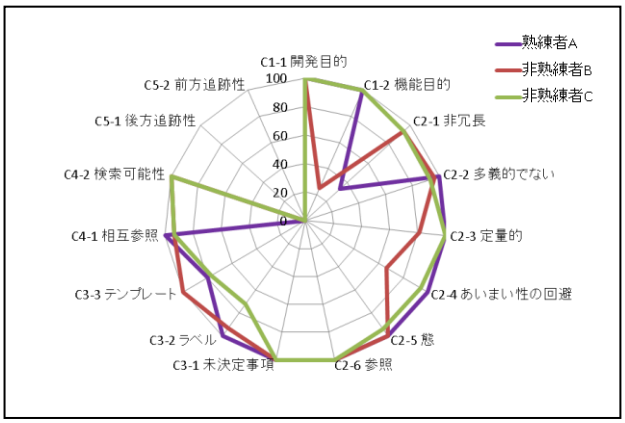


図3: 非正規化スコア

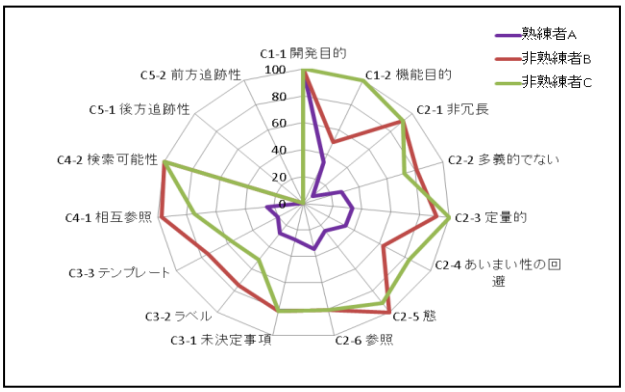


図4: 正規化スコア

各図の解説をする. 図2の目次項目ヒストグラムは, インスペクション対象の SRS が参照 SRS の目次項目をどれだけ網羅しているかを表したものである. 参照 SRS の各目次項目のスコアが 1 以上であれば対象の目次項目がインスペクション対象の SRS に存在することを表している. 図3および図4はドキュメント品質特性の満足度合いを評価したものである. 図3の非正規化スコアはインスペクション対象の SRS の目次項目を正とし, 各目次の特性の満足度合いを評価する. 一方, 図4の正規化スコアは参照 SRS の目次項目を正として評価する. つまり正規化スコアは, SRS が参照 SRS の目次項目全てを網羅しており, 且つ全ての目次項目が品質特性を満たしていなければ満点にならない.

図3の非正規化スコアについては担当者毎にばらつきがあり, 熟練者の方が高スコアとなっている. これは, SRS 記述ガイドを参照するだけでは非熟練者が品質特性を満たすように SRS を記述できないことを表している.

図4の正規化スコアについては非熟練者の方が熟練者よりも高スコアとなっている. これは, 非熟練者であっても SRS 記述ガイドを参照することで, SRS として必要な項目を不足無く記述できていることを表している.

図2の目次項目ヒストグラムからも図4と同様の結果を読み取ることができる. 熟練者が網羅している目次項目は全体の 30%程度であるが, 非熟練者は 85%程度を網羅している.

6.2. 定性的な評価結果

定性的な評価結果を以下に示す. なお, 定性的な評価は各グループにつき 1 つの SRS に対し評価している.

表5:ドキュメント品質の採点結果

品質特性	スコア	
	熟練者	非熟練者
責任追跡性	100	100
明確性	100	100
記述網羅性	70	70
変更容易性	80	90
追跡可能性	100	70
合計	450	430

表6:要求品質の採点結果

品質特性	スコア	
	熟練者	非熟練者
正当性	100	100
検証可能性	90	80
要求網羅性	70	70
無矛盾性	80	80
達成可能性	70	70
合計	410	400

総合的には非熟練は熟練者と遜色無いスコアとなっている. 特に, 品質特性の完全性に対応する記述網羅性や要求網羅性については同点となっている. このことから, SRS 記述ガイドを参照することによって, 非熟練者でも熟練者と同程度には要求を抜け漏れなく記述できるようになることが読み取れる. また, 変更容易性についても, 非熟練者の方が高いスコアとなっている. これは, 非熟練者の SRS は要求のタイプによって章が分けられており, 変更容易性が高いと判断されたためである.

また, 定性的な評価を行った評価者から, 良かった点として次のようなことが挙げられている.

表7:SRS の良かった点

SRS 作成者	良かった点
熟練者	・要求とその動機を対応付けたモチベーションビューによって, 要求の根拠が非常に分かりやすくまとめられている. ・ソフトウェアの動作が表でまとめられており, 振る舞いをイメージしやすい.
非熟練者	・要求の優先度に関する記述がある. ・性能要求がある. ・実装方針に関する制約や要求がある.

上記の評価からも, SRS 記述ガイドを参照することで, SRS として重要な項目を記述できるようになっていることが分かる. 一方で, 熟練者のように表現したいことに合わせて適切なビューやモデルを選定することは困難であることが分かる.

7. 考察

評価結果より, 非熟練者であっても, SRS 記述ガイドを参照することで SRS に必要な項目を抜け漏れなく記述できるようになっている. 一方で, 項目のような構造的な品質は向上しても, 品質特性のような記述そのものの品質の向上には至っていない. これは, SRS 記述ガイドが各項目の必要性や「何を書くか」については解説しているが, それらを「どう書くか」については言及していないためと考えられる. 従って, 更なる品質向上のためには, 品質特性を満たすように SRS を記述するためのガイドが必要である. このようなガイドの例としては, INCOSE の Guide for Writing Requirements[7]が挙げられる.

その他, SRS 記述ガイドを参照した非熟練者からの意見として, SRS 記述ガイドの解説文だけでは項目をどのように表現すれば良いか分からないとの声もあった. この問題については, 実際にその目次項目を記述したサンプルのようなものが有効であると考えられる.

8. まとめと今後の課題

SRS 記述ガイドは, 非熟練者であっても SRS に必要な項目を抜け漏れなく記述できるため SRS の品質向上に非常に有効である. その際, SRS 記述ガイドを補足するための記述サンプルを併せて作成することで, SRS 記述ガイドの理解を早められる. 今後は, SRS 記述サンプルと品質特性ガイドを策定し, 非熟練者が作成する SRS の品質向上への効果を評価する.

9. 謝辞

今回の評価において忙しい中協力をいただいた, 当社の林氏, 棚瀬氏に感謝いたします.

参考文献

[1] 蛸島昭之, 青山幹雄, 自動車ソフトウェア要求仕様書の第三者インスペクション方法の提案と適用評価, 情報処理学会論文誌, Vol.58 No.4 1–15 (Apr. 2017)

[2] IEEE, Std. 830-1998: IEEE Recommended Practice for Software Requirements Specifications, IEEE Computer Society (1998).

[3] ISO/IEC/IEEE 29148:2011, Systems and software

engineering —Life cycle processes — Requirements engineering(2011)
[4] Wiegers, K. and Beatty, J.: Software Requirements, 3rd ed., Microsoft Press (2013).
[5] JISA, 要求工学知識体系(REBOK) 第1版(2011)
[6] NASA: NASA-STD-2100-91, NASA Software Documentation Standard, NASA Headquarters Software Engineering Program (1991).
[7] INCOSE, Guide for Writing Requirements 2.1(2017)

付録

A.1. 記述ガイドの作成方法

本節では，記述ガイドのより具体的な作成方法について説明する．

最初に，あるドメインにとって標準的な SRS の目次項目を定義する．本論文で解説している記述ガイドの目次項目は[1]の参照 SRS を元に作成している．参照 SRS に対し，IEEE Std.830[2]の後継規格である ISO/IEC/IEEE 29148[3]への準拠や，車載ソフトウェアに必要な目次項目を追加した上で構成の再編を行っている．車載ソフトウェア以外の分野の場合は，これを該当分野の特徴に合わせた目次項目とする必要がある．

次に，各目次項目に対して本論文4.1に示した項目を記述する．これらを記述する際は外部の文献などを参照しながら記述することで，SRS 記述ガイド作成者の知識や経験のみに偏らないガイドとすることができる．本論文では，国際規格であるISO/IEC/IEEE 29148[3]をはじめ，複数の参考文献[4][5][6]を参照し，必要に応じて車載ソフトウェア向けに具体化したり表現の変更を行って記述している．それでも不足している部分については，熟練者の経験知により補完を行っている．

SRS 記述ガイドを作成する際の留意点として，SRS 記述ガイドを参照する組織の想定を予め決めておくことが挙げられる．理由は，組織によって解説の粒度を変える必要があるためである．例えば，性能に関する品質要求を記述する目次項目についての「書くべき内容」は，車載ソフトウェア横断の粒度では，「データ容量」や「ECUの起動時間」などの粒度で記述すべきであり，特定ドメインの粒度では具体的なデータ名称やタイミングを記述すべきである．具体化し過ぎた場合は無駄な項目ができてしまったり，逆に抽象化し過ぎた場合は非熟練者が記述内容をイメージできなかったりするため，適切な粒度での記述が必要である．

問題提起：提案依頼書（RFP）に含まれる「無理難題」を話題にして

神谷　芳樹
みたに先端研
ymitani@computer.org

門田　暁人
岡山大学
monden@okayama-u.ac.jp

要旨

ここ 10 年余，IPA/SEC や大学の情報科学研究科でソフトウェア・エンジニアリングに関する調査・研究に従事してきた．その中で，地方自治体など公的な機関による IT システム構築に関する提案依頼書（RFP）のいくつか（公開資料）に接し，詳細に評価する機会があった．そこには筆者らの経験してきた先端的なソフトウェア・エンジニアリング研究とはあまりにも乖離した驚きの世界があった．

そこではじめに一つの典型的な要求仕様書を紹介しながら問題提起の一文を Web サイトに寄稿し[1]，次いで，複数の提案依頼書を評価し，ジャーナル論文として寄稿した[2].

本論ではその経緯をあらためて整理し，その後の推移とあわせて若干の考察を示した．

1.　問題提起のきかっけ：最初の寄稿

ひとつの典型的な要求仕様書を紹介しながら，そこに含まれている「無理難題」をあぶり出し，問題提起とした[1].

(1) 要求仕様書の概要

紹介例は，市や町などの 5 つの医療機関の連合体による総合医療情報システムの開発に関する要求仕様書である（160 ページ）．タイトルは「○○連合総合医療情報システム要求仕様書」，表紙に大略次の文章がある（筆者が一部要約）．

（以下『 』内は当該要求仕様書からの引用．）

『本資料は，事業者が○○○医療情報システムの提案を行うにあたっての参考資料である．

本資料は，現時点で想定している○○○医療情報システムへの要求事項を，羅列したものであり，システム化にあたっては受託者による機能の整理・インテグレートが必要である．

受託者は，契約締結後，本資料および提案書をもとに，・・・・（5 つの医療機関・病院）とともに，○○○医療情報システム要求定義書を作成するものとする．』

つまり本資料は要求定義書作成作業の調達仕様で，要求定義書の骨格となる対象システムの要求条件を羅列した，と考えられる．この調達を受注しようとする者は，要求定義書作成作業に関する提案とともに，この調達仕様に羅列されている要求条件を整理し，対象システムの要求条件の大略を提案することが求められていると推定される．

そしてこの調達を受注できたら，顧客（つまり5つの医療機関・病院）と共同作業で，対象システムの要求定義書を作成し納めることになる．

そして実際にはこの要求定義書を納めた企業が，引き続き当該要求定義に基づいたシステム構築を実施する．

この調達仕様書には，対象システムの要求条件について基本的なことが，記述の粗密はあるものの，かなり詳細に示されている．一方，直接の調達対象である要求定義書作成作業の進め方については，冒頭の記述以外何も示していない．目次のおもな項目は下記である．但し，最後の機能要求項目一覧は実際には資料には含まれていない．

『1 はじめに
（背景，情報システム導入の目的，基本的な考え方）
2 基本要件
（業務範囲等，業務スケジュール，システム基本要件，ハードウェア／ソフトウェア／ネットワーク基本要件，データ移行要件，ファシリティ・インフラ／運用支援及び保守要件）
3 ネットワーク
（ネットワーク構築の基本方針）
4.機能要求項目一覧
（実際には含まれていない）』

(2) 要求条件中の「無理難題」

本仕様書には次のような驚きの表現がある．

『業務範囲：
本項は，本仕様書における業務の範囲の概要をまとめたものであるが，本情報システムの導入対象範囲は広く，構成する各サブシステムは複雑に連携することから，提案者側において，記載されている業務以外にも，必要とされる業務等がある場合は，発注者側に提案を行うものとする．
また，その他必要となる業務が生じた場合は，受託者の責任において，業務を行うものとする．』

『部門システム・外部機関との連携に係る仕様書の作成：
既存及び新規に導入予定の部門システムは現在，検討を進めていることから，決定後に既存及び新規に導入予定の部門システム，および外部機関との間で，連携に必要となる要件等の整理を行うこと．
※連携に係る部門システム側の経費については，別に見込むものとするが，本調達によって導入される情報システム側で発生する経費については，本調達に含むこと．』

『○○○○の構築に係るデータ連携等：
○○○○構築にあたり，既存の部門システムからデータを収集する必要があるものについては，この連携を構築することとするが，連携するシステムの選定は現在検討を進めていることから，決定後に連携を構築する．
※連携に係る一切経費を，本調達に含むこと．』

『○○○○支援システムに係るデータ連携等：
部門システムと同様に現在，検討を進めていることから，決定後に既存の○○○○システムと本調達により導入される情報システムとの間で，必要となる連携を構築すること．
※連携に係る一切経費を本調達に含むこと．』

『システム基本要件』
『(1)全般的事項』
『将来の機能拡張等におけるデータ移行時に特別な費用が発生しないこと．』
『法改正等のプログラム変更，パッケージのバージョンアップの際には，別途費用が発生しないこと．』

『(2)業務支援機能』
『同時処理件数の増加により，レスポンスに影響を与えないよう考慮されていること．
稼働年数の経過等によるデータ量の増加に伴って，レスポンスに影響が出ないように考慮されていること．』

『ハードウェア基本要件』
『(1)全般的事項』
『今回導入する情報システムは，マルチベンダ環境での利用を保証すること．』

『ソフトウェア基本要件』
『(1)全般的事項』
『医療情報システムとして，○○○病院以上の規模・機能の病院において，相当数の安定稼働実績のあるソフトウェアであること．』

『受注者として，相当数の導入実績と運用保守実績のあるソフトウェアであること．』

『(2)サーバ要件』
『データベースサーバ，アプリケーションサーバのOSは，オープン環境下のスタンダードなものを使用すること．また開発途中で陳腐化することがないよう十分な実績があり，かつ将来においてもその発展が見込まれるものであること．』

『データ移行要件』
『既存システムに蓄積された必要なデータを安全かつ確実に移行できること．』

『運用支援及び保守要件』
『(6)ソフトウェア保守』
『システムに関わる法令改正（診療報酬改正，薬価改定を含む．）が公示された場合は，速やかに対応し，施行前にシステム変更をし，運用に支障をきたさないこと．』

(3) 想定される課題

まず，このシステム構築はすでに医療情報システムとして相当数の実績のあるソフトウェアを使用し，かつその導入実績の豊富な企業でなければ応札できない．

次に，既存システムとの相互接続やデータ移行を求めている．一方，既存システムの仕様に関する情報がどの程度与えられるか不明である．本来はこれらの情報が要求定義の中に含まれなければ，見積もりはおろかシステムの実現性も見通せないだろう．

さらに，定義されていない条件によって，未来に発生する経費を見込んだり，あるいは現在は内容は分からないが将来確実に発生する作業を，費用が顕在化しない形で実施するように求めている．

そして「マルチベンダ環境での保証」や，「オープン環境下のスタンダードなもの」，さらには，「開発途中で陳腐化することがないよう十分な実績があり，かつ将来においてもその発展が見込まれるものであること」といった現実には不可能な条件を，それぞれの概念を定義することなく求めている．

2. 事例集積による論考の深化と論文寄稿

筆者らは上記の問題提起につづいて，典型的な 5 つの提案依頼書を評価し，「提案依頼書に含まれる無理難題の分類」として，事例紹介とともにジャーナル論文に寄稿した[2].

そこでは「無理難題」を(1)実績を求める要求，(2)技術的に

実現が難しい要求，(3)仕様の分からない既存システムの移行，連携を求める要求，(4)将来の課題への対応を求める要求，に分類した．

また事態改善に向けて，それぞれの項目に対する現実的な対策を例示した．

(1)実績を求める要求に関しては，「…○○の実績を有すること．○○に関する直接的な実績がない場合には，類似する実績や技術についてのエビデンスにより，○○の実施に支障がないことを示すこと．」

(2)技術的に実現が難しい要求に対しては，稼働率や稼働品質(応答時間，スループット，ターンアラウンド時間など)に関する要件については，具体的な数値や範囲を示す．

セキュリティに関する要件については，完全なセキュリティの実現は難しいため，最低限実現すべき要件を具体的に記述する，若しくは，順守すべきセキュリティ基準を明記すると共に，セキュリティが破られた際の対応に関する要件を記述する．

(3)仕様の分からない既存のシステムの移行，連携を求める要求に関しては，ユーザは既存システムの仕様を提供すること．

(4)将来の課題への対応を求める要求については，例えば「納入後のシステムの保守・運用については，別途保守契約を結ぶこととする．また，新たな要求によるシステムの変更，改良については，別途委託契約を締結し，それに基づいて実施するものとする．」など．

また，「システム改修などを施しても，改修など機能を含めて全機能が使用できること」といった要求は，「システム改修時の回帰テストに必要なドキュメントとツールを提供すること」といった要求に変更する．

3. その後の推移

こうした検討を進めているとき，ニュースの中で，本論で取り上げたような課題が含まれていると類推される訴訟事例が目に留まるようになった．例えば，

「旭川医大・NTT 東裁判の事例[3]」，
「京都市のシステム刷新失敗の事例[4]」，などがある．

いずれも著名な事例で，ここでは内容を省略するが，こうしたニュースを参照すると，本論で対象としたような提案依頼・調達に含まれる課題が，かなりの一般性をもったものであると類推できる．

また一方で公的機関による調達方式の改善努力も進められている．例えば国立大学での高額調達では，事前の仕様書案の公示と意見招請といった施策も試みられるようになった．

4. 一つの考察，調達と製品・サービス提供に関する組織活動の非対称性

類推ではあるが，こうした「無理難題」の背景の一つに組織活動に於ける調達(発注)と供給(受注)に関する非対称性があると考えられる．多くの企業・組織体では発注と受注，つまり調達と製品やサービス提供の活動は拮抗する．組織体はあるときは発注側，あるときは受注側として振る舞う．しかしながら，売り手市場の販売対象を持つ企業や医療，行政サービスなどの場合，受注側の活動は競争も少なく穏やかで，一方発注側の活動は買い手市場で，勢い強権的，いうなれば購買力を背景とした横柄な態度になることがある．こうした組織風土と，システム開発に関する技術未熟が重なると，本論で述べたような課題が現出する，ということがあるのではなかろうか．

本論の課題にはソフトウェア・エンジニアリングといった技術視点と合わせて，企業の組織風土のようなことにも踏み込んだ論考が必要なように感じられる．

5. まとめ

提案依頼書にこうした「無理難題」が含まれている場合，法務担当が完備し，リスク管理された一流企業は応札しないに違いない．リスク管理不在の企業が，「何でも出来ます，やります」方式で応札することになる．一般に受注側の提案書が発注側によって様々な視点から評価されるのに対し，発注側の調達仕様書・提案依頼書が評価される場は少ない．広く評価の視点に曝されることなく「無理難題」の羅列が提示され，それが「出来ます，やります」の姿勢で応札・受注され，調達行為が進んで行くのでは，双方にとって幸せな結果は得られないだろう．ソフトウェア・エンジニアリング以前の話，あるいは超・超上流工程の課題，ということになる．本論で示したような，これまであまり明るみに出して議論されることの少なかった課題に対して，幅広い認識とコンセンサスの醸成が期待される．

参考文献

[1] 神谷芳樹：RFP で垣間見たソフトウェア・エンジニアリングの現実，未来に発生する要求への対応要求など無理難題が，IT 記者会 Report（Web） 2014 年 6 月
[2] 門田暁人，住吉倫明，神谷芳樹：提案依頼書に含まれる無理難題の分類，SEC journal 51, 2017 年 12 月
[3] システム裁判は対岸の火事ではない，ユーザ企業が陥りがちな三つの罠，日経 XTECH，2017 年 10 月 4 日(Web)
[4] システム刷新に失敗した京都市，IT ベンダーと契約解除で訴訟の可能性も，日経 XTECH，2017 年 10 月 12 日(Web)

システム思考のモデリングはこれからのソフトウェアプロセスに有効か？

日下部 茂
長崎県立大学
kusakabe@sun.ac.jp

岡本 圭史
仙台高等専門学校
okamoto@sendai-nct.ac.jp

要旨

ソフトウェアの多様な利活用が進み，ソフトウェアのライフサイクルにおいて実世界や様々なシステムとのつながりを考慮することが必要となっている．筆者らはシステム思考のモデリングはこれからのソフトウェアプロセスに有効と考え，研究レベルのいくつかの取り組みを行っている．本フューチャープレゼンテーションでは，これまでの取り組みを紹介し，システム思考のモデリングはこれからのソフトウェアプロセスに本当に有効かの議論を行いたい．

1. はじめに

ソフトウェアは，計算だけに限らず，多様な目的での利活用が進み，そのライフサイクルにおいて，実世界や様々なシステムとのつながりを考慮することが必要となっている．安全性やセキュリティといった，様々な相互作用の分析を必要とする特性の重要性が高まっており，システム思考のモデリングはこれからのプロセスに有効と考えている．システムズエンジニアリングとそこで必要とされる分析を含むプロセスの関係はこれまでも議論も行われており，システム思考に関しても様々な枠組みが提案されている．ここではシステム思考の枠組みにはソフトウェア集約システムに有効とされている STAMP/STPA[3]を，プロセスについては例として ESPR をレファレンスとして議論を行う．有効性検討の事例として，アーキテクチャ記述言語 AADL[1]で記述したモデルを用いて STAMP/STPA, Fault Impact Analysis (FIA), Fault Tree Analysis (FTA)を実施して比較した例を紹介する．

2. ソフトウェア集約型システムのモデリング

STAMP は従来の解析的還元論や信頼性理論ではなくシステム理論に基づくハザードのモデル化と分析のために提唱されたもので，コンポーネント間の相互作用によって生じる創発的なものを含め，対象システムのハザードをコントロールするという観点に着目した事故モデルである(図 1 参照).

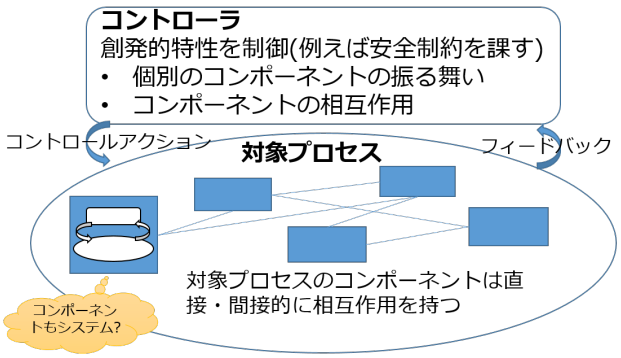


図1 対象プロセスに対するコントロール

STAMP/STPA はソフトウェア集約型のシステムが登場する以前に提唱された FTA や FMEA と異なり，ソフトウェア集約型システムを想定したものであるため，ソフトウェアのプロセスで効果的なモデル化と分析が可能と考える．

STAMP のモデルは，安全制約，階層的なコントロールストラクチャ，プロセスモデルという三つの基本要素で構成されており，コントロールストラクチャとプロセスモデルに対して，システムの安全制約が正しく適用されているかどうかに着目する．ここでコントロールストラクチャは，システムを制御する各機能の相互作用の構造を表すもので，コンポーネント間でやり取りされる制御の指示やフィードバックなどを表す．STAMP は安全工学の知見を広く活用することを念頭に提唱されており，モデリングの際の「事故」を「受容できない損失の発生」ととらえると，ソフトウェア開発での様々な関心事に適用可能とされている．

3. システム開発プロセスでの活用例

例えば組込みシステム開発のリファレンスプロセス ESPR[2]で考えると，SAP1：安全要求仕様書の作成や，

SYP2：システム・アーキテクチャ設計，SWP2:ソフトウェア・アーキテクチャ設計など FMEA や FTA といったハザード分析の実施が想定される部分では STAMP/STPA を活用可能と考える．

そのようなプロセスでの活用を念頭に，アーキテクチャ記述言語 AADL[1]で記述したモデルを用いて STAMP/STPA，Fault Impact Analysis (FIA)，Fault Tree Analysis (FTA)を実施した事例を紹介する．この事例の実施目的は，STAMP/STPA と他 2 手法の分析結果比較であり，特に STAMP/STPA の分析結果が他手法で得られるかを調査することを目的とした．

3.1. STAMP/STPA と AADL

STAMP/STPA によるハザード分析では，システム全体の振る舞いを示すモデルである，コントロールストラクチャを作成する．そして各コンポーネント間の相互作用，コントロールアクションの識別する．次にコントロールアクションの中からハザードを引き起こすものを非安全制御動作(Unsafe Control Action, UCA)として識別する．そして UCA がどのようにハザードを引き起こすのかの経緯を，必要に応じて更に詳細なモデルを作成するなどして分析し，誘発要因（Causal Factor）を見つける[3]．

システム開発プロセスの中では，仕様からシステムの構造や振る舞いをモデル化する．他方，STAMP/STPA ではハザード分析を行うためにシステムのモデルとしてコントロールストラクチャを構築する．これらの二種類のモデルを統合したモデルとして，アーキテクチャ記述言語 AADL とその安全分析用拡張である Error Model Annex を用いたモデルが提案されている[4]．

3.2. 分析

初めに，学習用に公開されている要求仕様を基に，AADL モデルを記述した．次に，作成した AADL モデルの抽象度を整理したものを STAMP/STPA のコントロールストラクチャ図とし，それを基に STAMP/STPA を実施した．最後に，この AADL モデルに，FIA に必要なエラー伝搬と，FTA に必要な状態遷移を Error Model Annex を用いて追記し，ツールによる FIA と FTA を実施した．なお，FIA，FTA に必要な情報を網羅的に記述し，分析を実施するのはコストがかかるため，STAMP/STPA で得られたハザードシナリオを選別しそのハザードシナリオに関連する情報のみを追記し，分析を実施した．

STAMP/STPA で識別したハザード H とそれを引き起こす UCA X の組のうち，単一コンポーネントの状態として定義されるハザード H に対し，FIA を用いて X が H を引き起こすことを追試できた．ただし，一般にハザードはシステムの状態であり，ハザード H が単一コンポーネントの状態として定義出来たために，UCA X から H へ至ることが識別できたことには注意が必要である．

FTA の分析結果から説明できない STAMP/STPA のハザードシナリオが存在した．今回，FTA に必要な状態遷移では，現在の状態のみに依存して次の状態を定義した．しかし，STAMP/STPA で識別されたハザードシナリオは，同一コンポーネントが状態 A から状態 B へ遷移した後にハザードへ至るという，過去の二状態の履歴に依存して引き起こされるハザードであった．FTA 用のモデルで適切な量の履歴に依存した状態遷移モデルを構築することで，FTA によってもこのハザードシナリオを識別できる可能性もあるが，分析前に適切な量の履歴を決定することは現実的ではない．

4. おわりに

筆者らが，これからのソフトウェアプロセスに有効と考える，システム思考のモデリングの導入について報告した．ソフトウェア集約システムに有効とされている STAMP/STPA を AADL と組み合わせて活用した事例では，FTA の分析結果から説明できない結果が得られており，ソフトウェアプロセスにはソフトウェア集約型システム向けのモデリング手法の効果が高い可能性がある．このような点も含め，システム思考のモデリングはこれからのソフトウェアプロセスに有効かについて議論を行いたい．

参考文献

[1] P. H. Feiler, D. P. Gluch, and J. J. Hudak, “The Architecture Analysis & Design Language (AADL): An introduction,” DTIC Document, Tech. Rep., 2006

[2] IPA SEC, “ESPR version2 組込みソフトウェア向け開発プロセスガイド改訂版”, 2007

[3] N. G. Leveson, “Engineering a Safer World”, MIT Press, 2011

[4] S. Procter and J. Hatcliff,, “An Architecturally-Integrated, Systems-Based Hazard Analysis for Medical Applications”, Formal Methods and Models for Codesign (MEMOCODE), 2014

ソフトウェア不具合予測への画像分類手法の適用

廣瀬　早都希
京都工芸繊維大学
大学院工芸科学研究科 情報工学専攻
s-hirose@se.is.kit.ac.jp

水野 修
京都工芸繊維大学
情報工学・人間科学系
o-mizuno@kit.ac.jp

要旨

ソフトウェアの品質を確保することはソフトウェア開発において重要である．そして，ソフトウェアの品質を保証するためには，ソフトウェアに含まれる不具合を早期に発見する必要がある．その課題に対し，本研究ではソフトウェアの不具合予測を行うシステムの試作を行なった．本研究では，ソースコードの変更点のテキストデータを画像化し，機械学習の *1* つであるニューラルネットワークを用いて画像分類を行うことで，ソースコードに不具合がふくまれるか否かを分類する手法を提案する．その手法において，全結合ニューラルネットワークと畳み込みニューラルネットワークの *2* 種類のモデルを作成し，学習・分類を行なった．その結果，全結合ニューラルネットワークでは，学習が正しく行われなかった．また，畳み込みニューラルネットワークでは，学習は行われたものの過学習となってしまったため，汎化性能がなく，画像分類ができなかった．今後，ネットワークの構成方法などを変更することで，モデルの改良を目指していく．

1　はじめに

現在，ソフトウェアは技術的及び経済的な発展において重要な役割を持っている．そのため，ソフトウェアには高い品質が求められる．しかしながら，ソフトウェア開発において高品質を保つことは，人的・時間的理由から難しい問題とされている．また，ソフトウェア開発過程における大部分を人間の手によって，直接的または間接的に行われるため，人的エラーを完全に排除することは難しく，ソフトウェアの品質保証においての問題の 1 つとされている．

こういった問題を解決する方法の 1 つとして，ソフトウェア不具合予測という手法がある．ソフトウェア不具合予測とは，過去のソフトウェア開発情報を利用することで，現在開発途中のソフトウェアに潜む不具合を予測することである．これにより，ソフトウェアに含まれる不具合を早期に発見できるため，後に重大な不具合を生じさせるリスクを減らすことができる．また，最終的にテストやメンテナンスにかけるコストを減らすことに繋がる．

ソフトウェア不具合予測に関する研究は古くから行われてきているが，Commit Guru[1] [1] というリポジトリの変更点 (commit) を解析してデータを与えるソフトウェア不具合予測用のツールが公開されたことからより進んできている分野である．近年では，ソフトウェア不具合予測に機械学習を用いた研究が Yang ら [2]，Wang ら [3]，Sharma ら [4] によって行われている．これらの研究では，ソフトウェア不具合予測に用いる特徴セットをより良いものとするために機械学習を用いて良い特徴を生成する手法が取られていた．つまり，機械学習の学習器部分は利用されているが，分類器としては重きを置いていない研究である．一方，機械学習をソフトウェア不具合予測の分類器として利用する手法が近藤ら [5] によって提案されている．この研究では，畳み込みニューラルネットワーク (CNN:Convolutional Neural Networks) を用いて，変更 (追加・修正) されたソースコードのテキスト情報から，変更後のソースコードに不具合が含まれているか否かを分類するという手法が取られている．この研究ではソフトウェアに潜む不具合を高精度で検出することができていた．

[1] Commit Guru:http://commit.guru

そこで，本研究では，近藤ら [5] の研究と同様に，機械学習を分類機として利用する．ただし，ソースコードの変更点における追加・修正情報をテキストとしてではなく，画像情報として取り出しソフトウェア不具合予測を行う手法を提案する．今回，テキスト情報を用いるのではなく，画像情報を取り扱うことにしたのは，人の目によってソースコードに含まれる不具合を発見できるのであれば，画像分類により不具合有無が判別できるのではないかと考えたからである．先行研究 [6] にてソースコードのインデントと複雑さには相関があるという結果が得られている．ソースコードの複雑さは不具合有無に関係しているため，ソースコードのインデントという画像で把握できる情報と複雑さに相関があるならば，画像分類によっても不具合を発見できるのではないかと考えた．

先行研究にて，ソースコードの変更点でのソフトウェア不具合予測モデルが有意であることが示されている [7–10]．変更点でのソフトウェア不具合予測では，変更が行われた段階で，その変更に対する不具合予測が行われるため，開発者に対するフィードバックが早く，また，開発者は不具合が検出された箇所のみレビューを行えば良いので，非常に簡単に不具合を解消することができる．これより，本研究でもソースコードの変更点でのソフトウェア不具合予測を行うこととする．

2 研究設問

本研究の目的はソフトウェア不具合予測をより簡易に行えるようにする方法を提案することである．先行研究 [5] ではテキスト情報に対して機械学習を行い，ソフトウェア不具合予測をすることに関して高い成果をおさめている．本研究ではソースコードの画像としての情報に着目し，ソースコードの変更点情報の画像を用いて機械学習を行い，変更点における不具合の有無を判定する手法を提案する．

よって，本研究における研究設問 (RQ:Research Question) は以下のように設定する．

- RQ1: ソースコードの変更点情報の画像で機械学習による学習は可能か?
- RQ2: 機械学習による画像分類でソフトウェア不具合検出を行った結果，精度はどの程度か?

3 準備

3.1 Commit Guru

Commit Guru [1] は Emad Shihab らによって公開されている，リポジトリを解析することで得られるデータをまとめて提供する Web サイトである．Commit Guru では，解析するリポジトリの各変更点に関して 13 項目の指標を記録する．ここで使用される 13 項目の指標は先行研究 [10] で設定されているものと同様に以下のものである．

- Number of Subsystems(NS:変更されたサブシステムの数)
- Number of Directories(ND:変更されたディレクトリの数)
- Number of Files(NF:変更されたファイルの数)
- Entropy(各ファイルにおける変更されたコードの分布)
- Line Added(LA: 追加されたコード行)
- Line Deleted(LD:削除されたコード行)
- Lines Total(LT:変更前ファイル内のコード行),
- Number Developers(NDEV:変更ファイルを編集した開発者の数)
- Age of changes (AGE:最後の変更からの時間)
- Number of unique changes(NUC:変更ファイルに含まれる固有の変更の数)
- Developer Experience(EXP:開発者の経験数 (総 commit 数))
- Recent Experience(REXP:開発者の最近の経験数 (日付で重み付けされた commit 数))
- Experience on a subsystem(SEXP:サブシステム上での開発者の経験数 (総 commit 数))

これら 13 の指標に加えて，変更点での不具合予測結果をラベル付けして，まとめたデータを公開している．Commit Guru で行われる不具合予測は，ログに含まれる commit メッセージを利用したものである．先行研

究 [11] に基づいたキーワード，’bug’，’fix’，’wrong’，’error’，’fail’，’problem’，’patch’ が commit メッセージに含まれていた場合，その変更点では不具合が修正されたと考えられる．不具合を修正している変更点が発見されると，そこから不具合が発生した変更点を特定する．そして，不具合が含まれている変更点の場合 bug=1 とし，不具合が含まれていない変更点の場合 bug=0 としてラベル付けを行う．ソフトウェア不具合予測モデルの研究において重要であるとされてきた実験データの公開性と透明性を保つため，本研究では Commit Guru で得られるデータを用いることとする．

3.2 Keras

Keras[2]は Python(汎用のプログラミング言語) で書かれており，迅速な実験を可能とするために開発された，TensorFlow や CNTK，Theano 上で実行可能な高水準のニューラルネットワークライブラリである．次のような場合に，Keras を利用すると良いとされている．

- ユーザーの技能やモジュールの内容，また拡張性によるが，簡単で，かつ短時間で試作の作成を行う場合
- 畳み込みニューラルネットワークや再帰型ニューラルネットワーク (RNN: Reccurent Neural Networks) を用いたり，これらの組み合わせでニューラルネットワークを作成を行う場合
- CPU 上，GPU 上で操作感を変えずに動作を行う場合

本研究では提案手法の試作が主目的となるため，短期間で開発が可能な Keras を利用した．

3.3 TensorFlow

TensorFlow[3]とは，Google が開発し公開している，機械学習で使用するためのソフトウェアライブラリである．TensorFlow では，ニューラルネットワークを柔軟に構築することができる．例えば，ニューラルネットワークのモデルをツールで組み立てたり，Python で直感的に書いたりなど，ニューラルネットワークを柔軟に記述できる．また，機械学習は計算量が多くなるため，グラフィックボードの計算能力を使用した演算が一般的であるが，CPU 演算や GPU 演算といった使用できるコードが異なるフレームワークがある．異なるフレームワークを使用するたびにコードを書き換えるのは手間がかかる．しかし，TensorFlow を用いるとコードを書き換える手間がなく，CPU と GPU 共に同じコードを使うことができるため，これらの切り替えがスムーズに行える．

本研究ではニューラルネットワークモデルを作成する際に 3.2 節で述べた Keras と TensorFlow を使用することとする．

[2]Keras:https://keras.io/ja/
[3]TensorFlow:https://www.tensorflow.org

4 実験準備

4.1 使用するソースコード

本研究では，Commit Guru で得られるデータを利用するため，Commit Guru で公開されている Git リポジトリの中から，bitcoin という仮想通貨を取り扱うシステムのプロジェクトを取得し，このソースコードを使用して，実験を行った．bitcoin のソースコードは汎用プログラミング言語の 1 つである C++を用いて書かれている．実験に使用するプロジェクトに bitcoin を選択したのは，十分な commit があることと，汎用プログラミング言語で書かれていることである．このプロジェクトの Git 上に記録されている commit 数は約 16000 個だが，今回は各コミットに対する不具合の有無を Commit Guru によるラベル付けから得ているため，Commit Guru によりすでに不具合ラベル付けがなされていた約 15000 個の commit を対象とする．

4.2 ソースコードの変更点の画像化

本研究では，画像分類による不具合予測を行うため，入力データとして使用する画像を用意する必要がある．取得したソースコードから変更点のみの画像化を行うため，今回は `git diff` を使用し，commit 同士の差分をとることとする．git diff による出力例を図 1 に示す．’-’ から始まる行が commit で削除された箇所で，’+’ から始まる行が commit で追加・修正された箇所である．今回は追加・修正された箇所に焦点を当てることとする．git diff で取り出した差分から ’+’ から始まる行を取り出し，取り出した行から先頭の‘ + ’を削除する．こうして取り出されたテキストを，画像サイズ 1024×1024 で RGB 画像として書き込む．なお，フォントは “NotoMono-Regular” と

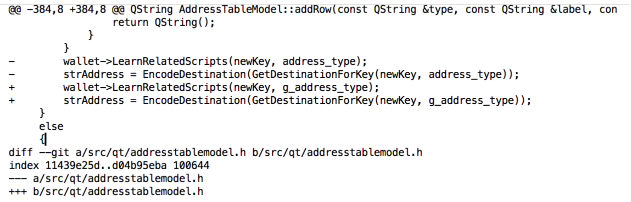


図 1. git diff コマンドの出力例

図 2. 変更点の画像化例

いう Google が公開している等幅フォントを使用し，フォントサイズは 8 ポイントとした．作成した画像例を図 2 に示す．画像例から分かるように，今回の実験で用いた画像は白黒画像であるが，RGB 画像として扱っている．これは今後の研究ではシンタックスハイライトなどの色情報を取り入れることを考えているためである．なお，画像サイズを超えるだけのテキストが抽出された場合，超えた分のテキストは縦横共に切り捨てた．機械学習を行う際は，画像サイズをこのまま用いると大きいため，計算量が大きくなりすぎる．よって，計算量を減らすために画像サイズを 256×256 に変換して利用する．

4.3 Commit Guru から不具合ラベルを取得

Commit Guru で公開されているデータを本研究では使用する．特に使用するデータは変更点における不具合の有無である．ソースコードの変更点それぞれで得られた変更情報のうち，追加・修正部分のみを取り出したテキスト画像に対して，Commit Guru から取得した各変更点の不具合の有無をラベルとして設定する．これにより，ソースコードの各変更情報画像に対して，不具合の有無情報を付与できる．

4.4 使用した機械学習モデルの構造

本研究では，機械学習モデルとして 2 種類のネットワークを利用して，その性能を比較した．ここでは，実験で使用した機械学習モデルの構造について説明する．

4.4.1 全結合ニューラルネットワーク

ニューラルネットワークとは，機械学習の手法の 1 つであり，人間の脳内にある神経細胞 (ニューロン) とその繋がりである神経回路網を人工ニューロンという数式的なモデルで表現したものである．入力層，中間層，出力層から構成され，層と層の間にニューロン同士の繋がりの強さを表す重み w が存在する．一つ前の階層から次の層へと入力が行われる時，入力される値に重み w をかけていき，最終的な値を出力する．学習時は，出力されたデータと教師データの誤差を計算し，出力データが教師データに近づくように重み w の調整を行う．分類時は，学習時に得られた重み w を利用して結果を出力する．全結合ニューラルネットワークモデルの概要は図 3 のようになっている．全結合ニューラルネットワークでは，1 つ前の階層から次の階層へとユニットの更新が行われる時に前の階層の全てのユニットにそれぞれの重み w をかけ，それらの総和をとる．この総和に対して活性化関数という，入力信号の総和を出力信号に変換する関数を用いて，次の階層のユニットを決定する．しかし，全てのユニットを使用して，次の階層のユニットを設定すると，過学習 (overfitting) が生じる危険性がある．過学習とは学習データに対しては学習されているが，未知のデータ (テストデータ) に対しては適合できていない状態となることである．つまり，過学習が行われたニューラルネットワークでは汎化性能がないため，未知のデータでの分類ができない．そこで，過学習防止のために Dropout という手法を取る．Dropout では，ニューラルネットワークでの学習の際に，次の階層への更新において，前階層の中のユニット全てを用いて次階層のユニットを求めるのではなく，前階層のユニットのうちのいくつかを無効

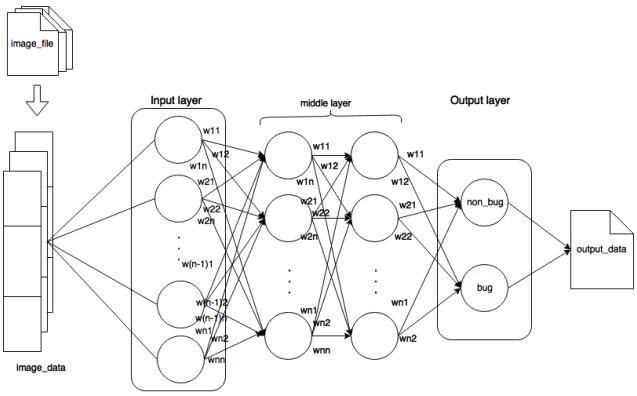


図 3. 全結合ニューラルネットワークモデルの概要

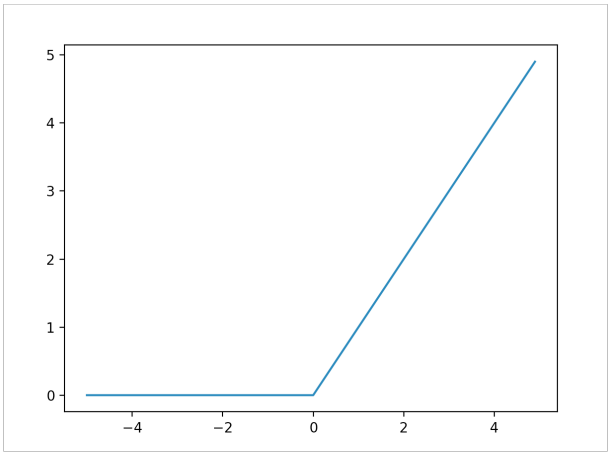


図 4. ReLU 関数

にして (存在しないものとして扱い) 学習を行う．そして，次の更新の際はまた別のユニットを無効にして学習を行うことを繰り返す．これにより，学習に際してネットワークの自由度を強制的に上げることで汎化性能を上げて過学習を避けることができる．

全結合ニューラルネットワークを使用するため，4.2 節で作成した画像情報を一次元配列に格納した．実験に使用する画像は RGB の 3 チャネルで作成しているので，一次元配列に最初の 1/3 が Red，次の 1/3 が Green，最後の 1/3 が Blue と並ぶように格納した．本モデルでは中間層は 2 層とし，各中間層のユニット数は 200 個で 20%のユニットが Dropout となるように作成した．また，活性化関数には以下の ReLU 関数 (式 1 図 4) を使用する．出力層では不具合の有無の 2 値を最終的な出力となるようにし，活性化関数として softmax 関数 (式 2 図 5) を用いる．

$$Relu(x) = \begin{cases} x & (x \leq 0) \\ 0 & (x < 0)ring \end{cases} \tag{1}$$

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \quad (i = 1, 2, \cdots, n) \tag{2}$$

4.4.2 畳み込みニューラルネットワーク

畳み込みニューラルネットワークは画像分類においてより高い成果をあげている機械学習アルゴリズムである．このニューラルネットワークは「畳み込み層」「プーリング層」「全結合層」を積み上げることで構成される．図 6 に示すのは畳み込みニューラルネットワークモデルの概要である．

畳み込み層では，入力された画像の特徴を捉えるためのフィルタを用いて，入力画像の特徴を捉えた画像データの様な二次元配列を作成し，この配列の各値に対して活性化関数を適用させた値を出力をとする．学習を行う際は，このフィルタの値がどの様になるかの学習を行い，分類を行う際は，学習によって作成されたフィルタを用いて，入力画像の特徴を抽出する．プーリング層では，畳み込み層で出力された二次元配列の中から有効な値だけを残す処理を行う．つまり，二次元配列を区切り，区切った小領域から有効な値を選択する．これにより画像分類を行う上であまり重要でない情報を削り，より特徴的な情報のみを残すことができる．全結合層は 4.4.1 節で説明した通りである．全結合層では一次元配列を取り扱うため，全結合層に入る前に二次元配列を平滑化し，一次元配列とする必要がある．

今回作成した畳み込みニューラルネットワークでは，まず入力層から畳み込み層を 2 個積み上げた後，プーリング層を積み上げた．なおプーリング層ではマックスプーリングという手法を用いる．マックスプーリングとは，それぞれの小領域に対して最大の値を選択するものである．この畳み込み層では，どちらもフィルタ数は 32 個とし，サイズは 3×3 とする．プーリング層では，マックスプーリングを適用する領域サイズを 2×2 とする．その後，さらに，畳み込み層とプーリング層を 1 個ずつ積

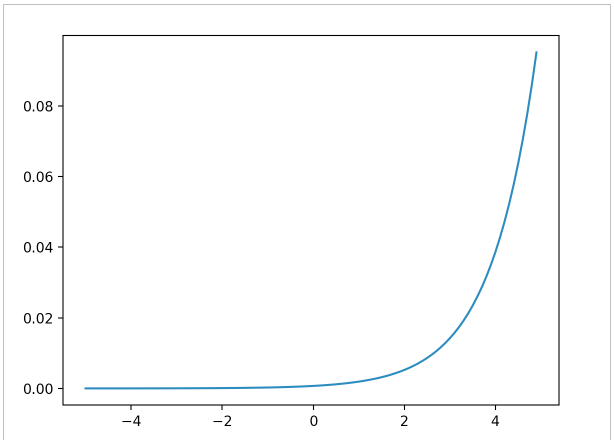


図 5. softmax 関数

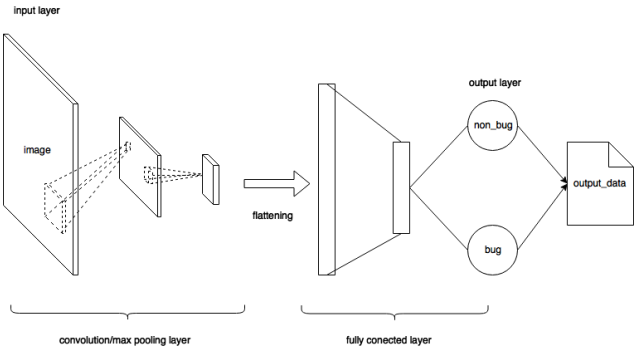


図 6. 畳み込みニューラルネットワークモデルの概要

み上げた後，得られたデータを平滑化し，全結合層で出力層と繋ぐこととする．こちらの畳み込み層では，フィルタ数を 64 個とし，サイズは 3×3 とする．プーリング層は先ほどと同じくマックスプーリングを適用し，領域サイズを 2×2 とする．畳み込み層と全結合層で利用する活性化関数は ReLU 関数とする．また，全結合層ではユニット数を 128 個とし，50%を Dropout となるようにした．全結合型ニューラルネットワークと同じく，最終的な出力は不具合の有無の 2 値とし，活性化関数として softmax 関数を用いる．

表 1. 混合行列

		分類結果	
		bug	non-bug
真のクラス	bug	TP	FN
	non-bug	FP	TN

4.5 評価尺度

本研究における評価尺度は，混合行列 (Confusion Matrix) と呼ばれる行列から求められる値に従って，定量的に評価を行うこととする．混合行列は表 1 で示す．混合行列とは，分類処理を行った際に，分類結果の値 (本研究では不具合が有無の 2 値) とその正誤 (真か偽か) についてまとめた表である．表で示されている各項目は以下のように示される．

- 真陽性 (TP:True Positive): 分類結果が不具合有り (bug) で，予測結果も不具合有り (bug) の場合
- 偽陽性 (FP:False Positive): 分類結果が不具合有り (bug) で，予測結果が不具合無し (non-bug) の場合
- 偽陰性 (FN:False Negative): 分類結果が不具合無し (non-bug) で，予測結果が不具合有り (bug) の場合
- 真陰性 (TN:True Negative): 分類結果が不具合無し (non-bug) で，予測結果も不具合無し (non-bug) の場合

これらの値を用いて，本研究で用いる指標，Recall(再現率)，Precision(適合率)，F1-score(F1 値)，Accuracy(正解率) の値を求める. 各指標について次に述べる.

4.5.1 Recall(再現率)

Recall(再現率) とは，実際に正であるもののうち，正であると予測されたものの割合である．つまり，実際に不具合 (bug) であったものの中から，不具合と分類されたものの割合である．混合行列 (表 1) を用いて定義すると以下のようになる．

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

4.5.2 Precision(適合率)

Precision(適合率) とは，正と予測したデータのうち，実際に正であるものの割合である．つまり，不具合と分類された結果の中から実際に不具合であったものの割合である．混合行列 (表 1) を用いて定義すると以下のようになる．

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

4.5.3 F1-score(F1 値)

F1-score(F1 値) とは，recall(再現率) と Precision(適合率) 率の調和平均である．以下のように定義される．

$$F1 = \frac{2 \times Rcall \times Precision}{Recall + Precision} \tag{5}$$

4.5.4 accuracy(正解率)

accuracy(正解率) とは，分類結果全体と、真のクラスが一致している割合である．混合行列 (表 1) を用いて定義すると以下のようになる．

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{6}$$

5 結果と考察

ここでは，研究設問 (RQ) に対する結果と考察を行う．

5.1 RQ1: ソースコードの変更点情報の画像で機械学習による学習は可能か?

5.1.1 概要

ソフトウェアの品質を保つ方法として，ソフトウェアに潜む不具合を早期に発見することがある．そこで，より手軽にソフトウェア不具合予測が行える様にするために，ソースコードの変更点の画像情報から，ソースコードに潜む不具合を検出できるかを提案した．そこで機械学習を利用して学習・分類を行うことができないか実験を行った．本研究では，ソースコードの変更点情報の画像で機械学習の分類を行うにあたり，学習時間が短い全結合ニューラルネットワークと画像分類に適しているが学習時間がかかる畳み込みニューラルネットワークを作成し，学習を試みた．

表 2. 学習データ・テストデータに含まれる画像データの内訳

	nunber of files
train-bug	2,071
train-nonbug	11,984
test-bug	208
test-nonbug	1,354

今回の実験では，学習データ約 14000 個，テストデータ約 1500 個を用いて実験を行うこととするここで用意した学習データ・テストデータはともに不具合有・不具合無の両方の画像が含まれているデータである．内訳は表 2 の様になっている．

このデータを用いて，本研究で用いた全結合ニューラルネットと畳み込みニューラルネットの 2 種類のモデルについて，それぞれ学習性能を示す訓練誤差 (train loss) を記録することで，これらのモデルで学習が行われているか，検証を行う．

5.1.2 結果と考察

全結合ニューラルネットワークと畳み込みニューラルネットワークそれぞれでの学習時に記録した訓練誤差 (train loss) をグラフ化したものを図 7 と図 8 に示す. 図 7 では訓練誤差の減少が見られない．よって，全結合ニューラルネットワークでは学習が行われていないことがわかる．一方，図 8 を見るとこれらを見る訓練誤差は減少している．つまり畳み込みニューラルネットワークによる学習は可能であると考えられる．

5.2 RQ2: 機械学習による画像分類でソフトウェア不具合検出を行った結果，精度はどの程度か?

5.2.1 概要

RQ1 で畳み込みニューラルネットワークによる学習が可能であることが分かった．よって，学習可能であった畳み込みニューラルネットワークを用いて，画像分類

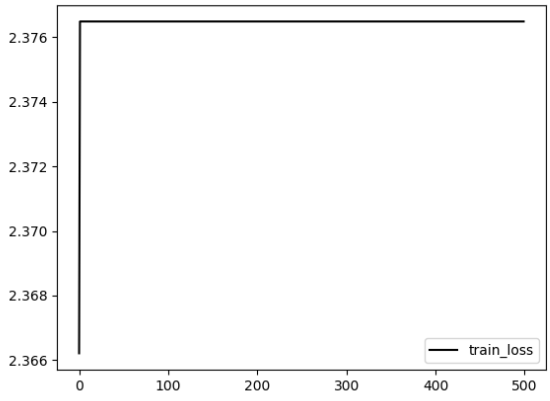


図 7. 全結合ニューラルネットワークの学習時の訓練誤差 (train loss)

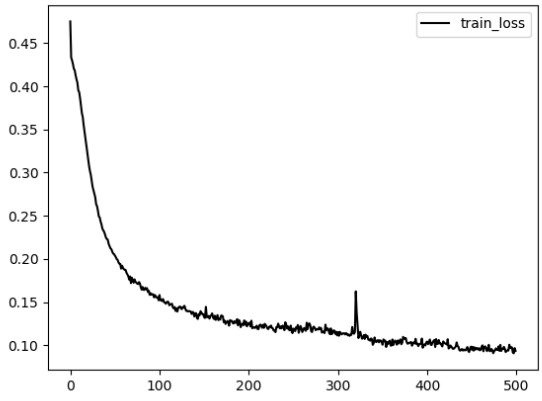


図 8. 畳み込みニューラルネットワークの学習時の訓練誤差 (train loss)

によるソフトウェア不具合検出を行い，その精度を検証する．

RQ1 で使用したデータと同様のものを用いて，畳み込みニューラルネットワークでの画像分類を行い，分類結果を記録した．さらに分類結果から，それぞれのニューラルネットワークの精度を考察した．

5.2.2 結果と考察

ソースコードの変更点の画像を用いて，畳み込みニューラルネットワークで学習を行い，分類結果を記録した．この分類結果を受け，本研究で使用した畳み込みニューラルネットワークモデルの精度について考察を行う．記録した分類結果から計算して得られた評価尺度の値を表 3 に示す. 表 3 によると，Accuracy(正解率) は高いが，Recall(再現率)，Precision(適合率)，F1-score(F1 値) はかなり低い結果となっている．使用した畳み込みニューラルネットワークの分類結果の値と真の値とで判断される混合行列 (各項目には当てはまる個数が示される) は表 4 の通りである．本研究の目的としては，ソフトウェアの変更点に潜む不具合を検出することにある．しかしながら，この結果を見ると，全ての不具合 (208 個) のうち，不具合として検出されたのはわずか 18 個である．これでは，Accuracy(正解率) が高くともソフトウェアの不具合検出器としての精度が高いとは言えない．

では，何故畳み込みニューラルネットワークを分類器として用いた時，精度が低くなったのかの考察を行う．RQ1 より，畳み込みニューラルネットワークでの学習は行われていた．しかし，この学習が正しく行われているかは不明である．そこで，畳み込みニューラルネットについて，学習性能を示す訓練誤差と汎化性能を示すテスト誤差を記録することで，学習が正しく行われているかを確認を行った．図 9 に記録した訓練誤差 (train loss) とテスト誤差 (test loss) のグラフを示す．

これを見ると，訓練誤差 (train loss) は減少しているが，テスト誤差 (test loss) は学習回数が増す毎に上昇していることがわかる．これは，畳み込みニューラルネットワークが過学習していることを示している．つまり，今回学習した畳み込みニューラルネットワークには汎化性能は全くないと言える．

過学習となった原因を考えると，次の 2 点が挙げられる．

- 学習データ数が少ない．
- 中間層のパラメータ数が多すぎる．

前者に関しては，本研究にあたり Commit Guru で公開されているソースコードから commit 毎の変更情報を画像化するということで，データ数を増やすのが困難であったため，少ないデータ数での実験となった．そのため，作成した画像データにノイズを加えたりなどして，学習データの水増しを行ったりする必要がある．また，明らかに不具合有りのデータが少ないので，不具合有りと不

表 3. 画像分類による不具合検出の評価

	畳み込みニューラルネットワーク
Recall	0.087
Precision	0.261
F1-score	0.130
Accuracy	0.846

表 4. 畳み込みニューラルネットワークの分類結果から得られる混合行列の値

		分類結果	
		bug	non-bug
真のクラス	bug	18	190
	non-bug	51	1303

具合無しの画像データ数を揃えることも必要である．

後者に関しては，畳み込みニューラルネットワークでは，画像サイズが 256×256 の分類をするためには中間層のパラメータを大きくする必要がある．本研究で用いた中間層のパラメータは調整を行わず，適度な値を設定していた．そのため，データ数に対する中間層のパラメータが大きいため，過学習となってしまったと考えられる．よって，データ数を増やすことで改善が見られるかもしれない．また，中間層を必要最小限まで減らしたり，Dropout を適切に設定することで改善が見られる可能性も考えられる．しかし，テスト誤差を見る限りでは，全く減少が見られないため，汎用性能は上がらない可能性もある．

6 今後の課題

本研究の結果・考察 (5.1.2 節) から，全結合ニューラルネットワークでは学習が行われず，畳み込みニューラルネットワークでは学習は行われたが，過学習となってしまったため，画像分類は上手く行えなかった．そこで，今後は畳み込みニューラルネットワークが過学習をせずに学習を終え，画像分類ができるようになることを目標とする．畳み込みニューラルネットワークが過学習する原因として，以下の問題点が与えられた．

- 実験で使用した学習データ数が少ない．

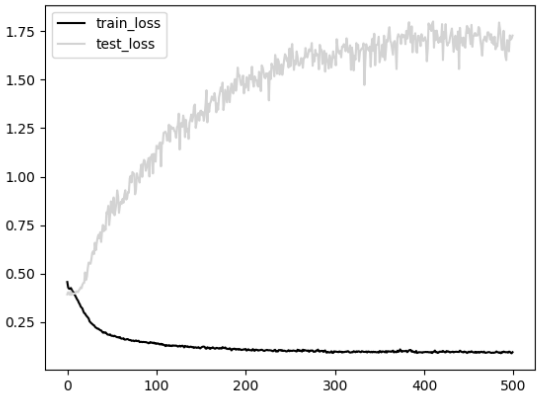


図 9. 畳み込みニューラルネットワークの訓練誤差とテスト誤差

- 実験で使用した畳み込みニューラルネットワークの中間層のパラメータ数が多すぎる．

これらの問題点を解決すれば，畳み込みニューラルネットワークでの学習が正しく行えるという仮説のもと，今後の課題を次に設定する．

1. 実験で使用するデータ数を増やす．
2. 畳み込みニューラルネットワークのモデル作成の際，適切な中間層の値を設定する．
3. 利用する画像データをシンタックスハイライトを利用したものにする．

本研究では，妥当性に対する措置を取っていなかった．今後は妥当性に対する検証も行わなければならないため，上記の他に新たに以下を今後の課題として挙げる．

1. 実験で使用するデータの偏りを減らす
2. 実験で使用するプロジェクトの種類を増やす
3. 10 重交差検証を行う
4. 作成したプログラム (畳み込みニューラルネットワーク) に不具合がないかを確認する．

また，畳み込みニューラルネットワークでは画像のどこに注目し判断しているか，画像の特徴箇所を特定することができる．よって，これからの研究で畳み込みニュー

ラルネットワークでの学習・分類ができることがわかれば，この技術を用いて人の目による不具合検出の糧としたい．

7 まとめ

本研究では，機械学習の画像分類を用いてソフトウェアに含まれている不具合を判定する研究を行った．全結合ニューラルネットワークでは実験データの特徴を捉えきれず，学習が行われなかった．そこで畳み込みニューラルネットワークを使用することにしたが，学習の際に過学習を行ってしまい，正しく画像分類は行われなかった．今後の課題としては，より多くのデータを用いて畳み込みニューラルネットワークでの学習を行うことと，畳み込みニューラルネットワークモデルにおける中間層の値を見直すことが挙げられる．これにより，畳み込みニューラルネットワークでの画像分類でソフトウェア不具合予測が行えるかを再度検証する．

参考文献

[1] C. Rosen, B. Grawi, and E. Shihab, “Commit guru: Analytics and risk prediction of software commits,” Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp.966–969, New York, NY, USA, July 2015.

[2] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” QRS’15: Proceeding of the 2015 IEEE International Conference on Software Quality, Reliability and Security, pp.17–26, Washington, DC, USA, Aug. 2015.

[3] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” ICSE ’16: Proceedings of the 38th International Conference on Software Engineering, pp.297–308, New York, NY, USA, May 2016.

[4] R. Sharma and P. Kakkar, “Software module fault prediction using convolutional neural network with feature selection,” International Journal of Software of Software Engineering and Its Applications, vol.10, no.12, pp.307–318, 2016.

[5] 近藤将成，森　啓太，水野　修，崔　銀惠，“深層学習による不具合混入コミットの予測と評価,” ソフトウェアエンジニアリングシンポジウム 2017 論文集，vol.2017，pp.35–45，Aug. 2017.

[6] A. Hindle, M. Godfrey, and R. Holt, “Reading beside the lines: Indentation as a proxy for complexity metrics,” In Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, pp.133–142, May 2008.

[7] S. Kim, J.E.J. Whitehead, and Y. Zhang, “Classifying software changes: Clean or buggy?,” IEEE Transactions on Software Engineering, vol.34, no.2, pp.181–196, March 2008.

[8] L. Aversano, L. Cerulo, and C.D. Grosso, “Learning from bug-introducing changes to prevent fault prone code,” IWPSE’07: Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, pp.19–26, NY, USA, July 2007.

[9] T. Jiang, L. Tan, and S. Kim, “Personalized defect prediction,” ASE’13: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp.279–289, NJ, USA, Nov. 2013.

[10] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” IEEE Transactions on Software Engineering, vol.39, no.6, pp.757–773, June 2013.

[11] A. Hindle, D.M. German, and R. Holt, “What do large commits tell us?: a taxonomical study of large commits,” MSR’08: Proceedings of the 2008 international working conference on Mining software repositories, pp.99–108, New York, NY, USA, May 2008.

ソフトウェア・シンポジウム 2018

日程 ： 2018 年 6 月 6 日（水）～ 8 日（金）
場所 ： かでる 2・7
主催 ： ソフトウェア技術者協会（SEA）

プログラム [6/6（水） 1 日目]

時間	内容	
12:30 13:00	＜受付＞ ※1 受付場所： 820 研修室（8F）	
13:00 13:15	＜オープニング＞ 8F 820 研修室 実行委員長： 本多 慶匡（東京エレクトロン） プログラム委員長： 安達 賢二（HBA）	
	＜形式手法＞ 5F 520 研修室 司会：酒匂 寛（デザイナーズデン）	＜品質管理＞ 8F 820 研修室 司会：大平 雅雄（和歌山大学）
13:20 13:45	研 研究論文 アジリティのある探索的形式仕様記述のためのテストフレームワーク 小田 朋宏（SRA）	研 研究論文 データ値の差異とデータフローの視覚化によるデバッグ補助手法の提案 神谷 年洋（島根大学）
13:50 14:15	研 研究論文 SOFL 形式仕様に基づく C# プログラムのテストツール 網谷 拓海（法政大学）	研 研究論文 不具合混入コミットの推定手法間での整合性比較と考察 北村 紗也加（京都工芸繊維大学）
14:20 14:45	研 研究論文 ソースコードから CDFD への変換による SOFL 仕様記述の支援ツールの提案 新城 汐里（法政大学）	研 研究論文 不具合誘発パラメータ組み合わせ特定三手法の比較評価 渡辺 大輝（京都工芸繊維大学）
14:45 14:55	＜休憩＞	
	＜チーム力向上＞ 5F 520 研修室 司会：中森 博晃（パナソニック スマートファクトリーソリューションズ）	＜開発管理＞ 8F 820 研修室 司会：天嵜 聡介（岡山県立大学）
14:55 15:20	経 経験論文 モデリングによる暗黙知分解とスキル補完への取り組み～共感と共創をつくり，人材不足解消と多能工を促進～ 三輪 東（SCSK）	研 研究論文 バグ修正時間を考慮したソフトウェア最適リリース問題についての一考察 岡村 寛之（広島大学）
15:25 15:50	経 経験論文 アジャイルの振り返りとシステム・シンキングの有効性について 日山 敦生（緑ビジネスコーチ研究所）	研 研究論文 トピックモデリングに基づく開発者検索手法の構築へ向けて 福井 克法（和歌山大学）
15:55 16:20	事 事例報告 結合・総合テストフェーズにおける継続的テスト設計の取り組み 山口 真，豊田 圭一郎，田辺 紘明（SCSK）	経 経験論文 リスク構造化を用いたリスクマネジメント手法の提案と効果分析～「未来予想図」を用いたリスクマネジメント PDCA サイクル～ 水野 昇幸（TOC/TOCfE 北海道）
16:20 16:40	Coffee Time 8F 820 研修室	
16:40 17:40	キーノートスピーチ（1） 8F 820 研修室 講演題目： 北海道のおいしい時間 ～食のつくりびとの言葉 講演者： 小西 由稀（フードライター）	
18:30 20:30	情報交換会 ※2 会場：さっぽろテレビ塔 2F（18:15 開場）	

※1 オープニング以降の受付は、以下の場所となります。
6/6(水) 820 研修室
6/7(木) 12:30 まで：820 研修室，14:00 以降：520 研修室
6/8(金) 14:30 まで：520 研修室，14:30 以降：大会議室
※ 事務局の栗田が不在で、お急ぎの場合は、070-6429-0240 にお電話ください。（この番号は会期中にしかつながりません）
※2 情報交換会は 18:15 に開場し、18:30 に “開始” します。お早めにお越しください。アクセスについては P.4 をご覧ください。
※3 2 日目 , 3 日目はともに、8:55 に開場します。
※4 時間は各会場が利用できる時間帯を示しています。会議室は 21:00 まで利用可能で、終了時間は WG/TS 毎に異なります。
各 WG の開催場所については P.3 をご覧ください。
運営方法は WG ごとに異なります。詳細は各 WG のリーダーまでお問い合わせください。
※5 **ワーキング（レビュー）「WG からの報告」は、お申し込みいただいた WG と異なるグループに参加されてもかまいません。**

プログラム [6/7（木）２日目]

時間	内容	
	<個と組織の成長> ※3 5F 520 研修室 司会：米島 博司（パフォーマンス・インプルーブメント・アソシエイツ）	**<要求工学>** ※3 8F 820 研修室 司会：鈴木 正人（北陸先端科学技術大学院大学）
09:00 09:25	研 研究論文 **システム理論に基づくモデリングと質的研究を併用したソフトウェアプロセス教育の動機づけシナリオ開発** 日下部 茂（長崎県立大学）	研 研究論文 **要求獲得のためのヒアリングにおけるゴール指向要求分析の活用～「ゴール指向 Lite」の提案～** 菅原 扶（インテック）
09:30 09:55	事 事例報告 **現場に寄り添った教育が品質を支える ～ディスカッション教育に込めた想い～** 渡辺 聡美（富士通エフ・アイ・ピー）	事 事例報告 **Applying PReP Model to a Service Development Process** 木ノ内 浩二（ウェザーニューズ）
10:00 10:25	事 事例報告 **勉強会を活用した組織成長モデル ～活動 2 年目の成果と課題～** 伊藤 修司（SCSK）	経 経験論文 **要求記述のスキル不足に対する SRS 記述ガイドの有効性評価** 不破 慎之介（デンソークリエイト）
10:25 10:40	**<休憩>**	
10:40 11:40	**キーノートスピーチ（２）** 8F 820 研修室 **講演題目：ファームノートの挑戦。 Internet of Animals で切り拓くこれからの農業** 講演者： 小林 晋也（株式会社ファームノートホールディングス）	
11:40 11:50	**<休憩>**	
	<Future Presentation (1)> 5F 520 研修室 司会：本多 慶匡（東京エレクトロン）	**<Future Presentation (2)>** 8F 820 研修室 司会：安達 賢二（HBA）
11:50 12:40	**問題提起：提案依頼書（RFP）に含まれる「無理難題」を話題にして** 神谷 芳樹（みたに先端研）	**システム思考のモデリングはこれからのソフトウェアプロセスに有効か？** 日下部 茂（長崎県立大学）
12:40 14:00	**<昼食>**	
	<ワーキンググループ・チュートリアル> ※4	
14:00 18:00	研 研究論文 （14:15-14:40） 9F 930 研修室 **ソフトウェア不具合予測への画像分類手法の適用** 廣瀬 早都希（京都工芸繊維大学） ※こちらの研究論文は、「WG14(XS)：ソフトウェア開発の現状と今後の発展に向けたディスカッション」の中での発表となります。 5F 520 研修室 WG1(EF) ：未来に活躍できるソフトウェアエンジニア 9F 910 会議室 WG2(FM) ：形式仕様言語を用いたモデリング 5F 520 研修室 WG3(HG) ：開発をもっと楽しく！ ゲーミフィケーションを使った開発ハック 5F 520 研修室 WG4(ME) ：Software Maintenace and Evolution，現場に笑顔と「ありがとう」をもっと！ 5F 520 研修室 WG5(OP) ：「組織パターン」を使って組織運営を振り返る 5F 540 会議室 WG6(OS) ：OSS のこれまで、いま、これから 5F 520 研修室 WG8(PS) ：新サービス創出のための Model Based アプローチ 5F 520 研修室 WG9(PR) ：要求技術者の責任と社会システムの狭間 5F 520 研修室 WG10(SA) ：きちんと動くモデルカーを作ってみよう 6F 620 会議室 WG11(SC) ：ソーシャルコーディングとソフトウェア進化 7F 750 研修室 WG12(ST) ：システム思考アプローチでモデリングをやってみた 7F 740 研修室 WG13(TS) ：エンジニアのトリセツ２ ～寿命 100 年時代：70 才まで現役で働けますか？～ 9F 930 研修室 WG14(XS) ：ソフトウェア開発の現状と今後の発展に向けたディスカッション 6F 610 会議室 TS1(SM) ：ソフトウェア技術者のためのマルウェア入門 10F 1020 会議室 TS2(TT) ：人と人、チームとチームを繋ぐコーディネーターになろう！ ※ 会議室は 21:00 まで利用可能です。 ※ WG リーダの方へ 13:55 以降に 520 研修室に鍵を取りにきてください。 18:15 までに 520 研修室に鍵を返しにきてください。 (18:00 以降も延長する場合は，当日は 520 研修室にいる栗田か本多か三輪にお知らせください。)	

プログラム [6/8（金）３日目]

時間	内容
	<ワーキンググループ・チュートリアル>※3 , ※4
09:00 12:00	※ 各 WG と TS の部屋は、2 日目と同じなります。 ※ WG リーダの方へ 8:55 以降に 520 研修室に鍵を取りにきてください。 14:45 までに 大会議室に鍵を返しにきてください。
12:00 13:30	**<休憩>**
13:30 14:30	**WG からの報告（レビュー）** 各 WG の部屋 ソフトウェアシンポジウムでは、興味深いワーキングが毎年開催されておりますが、自身が参加したもの以外はその内容を知ることが困難です。 通常行われる最後の報告の全体セッションは時間の制約もあり、かならずしも躍動感のある議論の内容を肌身で感じることができません。 そこで、2 日間のワーキングに参加された以外の方も 2 日間の議論の最後のまとめの時間帯に参加できれば、ソフトウェアシンポジウム参加者にとって、有益な情報を得れるチャンスが増え、ソフトウェアシンポジウムに参加した意義がより増加するのではと考え、本年度は実施報告を聞くワーキング（レビュー）のセッションを設けました。 **※ワーキング（レビュー）「WG からの報告」は、お申し込みいただいた WG と異なるグループに参加されてもかまいません。**
14:30 14:50	**Coffee Time** 4F 大会議室
14:50 15:50	**キーノートスピーチ（3）** 4F 大会議室 **講演題目：日本初の民間宇宙ロケット MOMO のアビオニクス開発** 講演者：森岡 澄夫（インターステラテクノロジズ株式会社）
15:50 16:15	**クロージング** 4F 大会議室 実行委員長：　中野 秀男（帝塚山学院大学） 本多 慶匡（東京エレクトロン） プログラム委員長：　落水 浩一郎 (University of Information Technology, Myanmar) 安達 賢二（HBA）

会場案内

かでる 2.7

情報交換会（1 日目　18:30 ～）

会場案内

会場：さっぽろテレビ塔　　**開場 :18:15 ～　開始 :18:30 ～**
住所：札幌市中央区大通西一丁目

※ 情報交換会は 18:15 に開場し、18:30 より “開始” します。お早めにお越しください。
※名札の着用をお願いいたします。

その他

最新情報

最新情報について

SS2018 の最新情報は、随時 Web ページに掲載いたします。公式ページの「新着情報」をご覧ください。
http://sea.jp/ss2018/news.html

MEMO

ソフトウェア技術者協会（SEA）

http://sea.jp/ss2018/

6月に札幌で「開かれたソフトウェア開発」について話しませんか

ソフトウェア・シンポジウム 2018（SS2018 in 札幌）

第 38 回目を迎える 2018 年のソフトウェア・シンポジウムでは，この数年間で試みてきた新しい取り組み（チュートリアルや Future Presentation など）をさらに発展させたものにしたいと考えています．このほか，SS2017 に引き続き，論文発表や事例報告と，ワーキンググループで議論を行います．

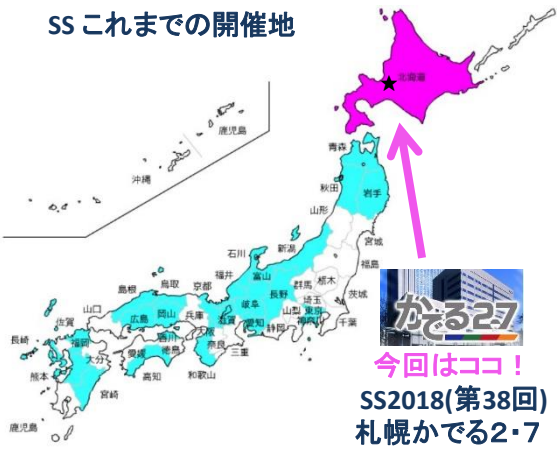
【開催要領】
◆日程: 2018 年 6 月 6日（水）～ 8 日（金）
◇場所: かでる 2・7

【論文投稿スケジュール】
◆投稿締切: 2018 年 3 月 12日（月）
◇採否通知: 2018 年 4 月中旬（予定）
◆最終原稿締切: 2018 年 5 月 18 日（金）

【募集要領】
◆研究論文 (A4 版 5 ページ以上 10 ページ以内)
◇Future Presentation (A4 版 1 ～ 2 ページ)
◆経験論文 (A4 版 5 ページ以上 10 ページ以内)
◇事例報告 (要旨 A4 版 1 ページと，スライド原稿 12 枚 ～ 18 枚)

SS2018プログラム委員長
落水 浩一郎 (University of Information Technology, Myanmar，金沢工業大学)
安達 賢二 (HBA)

SS これまでの開催地



ソフトウェア・シンポジウム（SS）とは

第 1 回目のソフトウェア・シンポジウムは，1980 年の末に開催された．主催は，ソフトウェア産業振興協会 (当時) の技術委員会 (SIA/TC) であった．通常のアカデミックなコンファレンスとは違って，開発現場で働くソフトウェア技術者たちが，自らの経験を通じて獲得した実践的な技術や知識を交流する場が必要だという認識は，この委員会の発足当時から強く存在していたのだが，ある技術調査のためのワーキング・グループ活動をきっかけとして，その成果発表を兼ねたイベントとして，このシンポジウムが企画されたのである．それからほぼ 1 年後の 1982 年冬には，第 2 回目がやはり同様なコンテクストで開催された．この 2 回のプロトタイピングの成功にもとづいて，翌 1983 年以降のソフトウェア・シンポジウムは，毎年 6 月に定期的に開催するようになった．

ソフトウェア技術者協会 (SEA，「シー」) は，ソフトウェアハウス，コンピュータメーカ，計算センタ，エンドユーザ，大学，研究所など，それぞれ異なった環境に置かれているソフトウェア技術者あるいは研究者が，そうした社会組織の壁を越えて，各自の経験や技術を自由に交流しあうための「場」として，1985 年 12 月に設立されました．

分科会活動

- 環境分科会 (SIGENV)
- 教育分科会 (SIGEDU)
- ネットワーク分科会（SIGNET）
- プロセス分科会 (SEA-SPIN)
- フォーマルメソッド分科会 (SIGFM)
- オープンソース分科会（SIGOSS）
- 品質保証分科会 (SigSQA)
- システムオブシステムズ分科会（SigSoS）
- 信頼性研究会（FORCE）

支部活動

- 北海道支部
- 東北支部
- 横浜支部
- 北陸支部
- 名古屋支部
- 関西支部
- 広島支部
- 九州支部
- 上海支部

ソフトウェア・シンポジウム 2018
論文集

2018 年 6 月 30 日　初版発行

編者　　小笠原秀人
　　三輪東

発行者　ソフトウェア技術者協会

〒157-0073 東京都世田谷区砧二丁目 17 番 7 号
株式会社ニルソフトウェア内
ソフトウェア技術者協会　事務局
TEL: 03-6805-8931
http://sea.jp/
ISBN 978-4-916227-24-9

ソフトウェア技術者協会