ソフトウェア開発工程における不具合発生要因の分析

── ベイジアンネットワークによるモデル構築と評価 ──

菅谷　克行

要旨

近年、ソフトウェアに対する要求は大規模・複雑化する一方である。そのため、ソフトウェア開発工程で発生・観測される不具合は増大しており、それら不具合の対処にプログラマーやプロジェクト・マネージャは苦労している。そこで本研究では、組み込みソフトウェア開発プロジェクトにおける不具合発生のマネージメント（予測と対策）を支援する方法を考察・提案することを目的とした。まず、ソフトウェア開発現場に対するヒアリング調査により問題点や支援のポイントを洗い出し、不具合情報を活用したソフトウェア開発現場の支援サイクルを提案した。次に、機械学習の一手法であるベイジアンネットワークを用いて、観測された不具合データから不具合発生モデルを構築した。構築したモデルと、モデル上のシミュレーションで得られたルールから、複数の不具合発生要因とプロジェクト・マネージメント上の有用な知見（方策）が得られた。その結果、構築したモデルが不具合発生マネージメントの支援につながる可能性を示すことができた。

キーワード：ソフトウェア開発、組み込みソフトウェア、不具合発生モデル、ベイジアンネットワーク、シミュレーション、ルール抽出

1. はじめに

現代の社会や人々の活動の多くの部分は、電子システムによって支えられていると言っても過言ではない。電子システムにはマイクロコンピュータが搭載されており、その中でソフトウェアの制御によって多種多様な機能やサービスを実現している。発電プラントや交通機関の制御などの公共サービスから、デジタル家電や携帯電話・端末などの身の回りの電子機器や自動車にまで様々な電子システムが搭載され、その中でソフトウェアが制御・機能を実

現することにより現代社会の基盤は成り立っている。この意味で、ソフトウェアは現代社会の中心に位置していると言える。そのため、ソフトウェアの不具合は人々の社会活動に対して影響が大きく、実際、銀行ATMや自動改札、航空管制のシステムトラブルで人々が混乱している報道は、枚挙にいとまがない。

ソフトウェアの中でも、自動車や携帯電話、家電製品などの内部に、製品の一部として組み込まれて制御などの機能を実現しているソフトウェアを組み込みソフトウェアと呼び、企業や行政など組織の経営や活動を支援しているエンタープライズ系のソフトウェアとは区別される。近年、組み込みソフトウェアに対しては、製品競争の激化から、短期間に新機能や新製品の開発が求められ、同時に高い品質・信頼度が求められている。実際、ソフトウェアに関わる不具合が、製品のリリース時期や品質問題に大きな影響を及ぼしていることが明らかにされている[1][2]。

通常、ソフトウェア開発プロジェクトの管理は、プロジェクト・マネージャが行なっている。プロジェクト・マネージャは、担当プロジェクトに不確実性から発生し得るリスクを予測し、その対策を準備しながらプロジェクト全体のマネージメントをする必要がある[3]。ソフトウェア開発の現場では、リスク発生とは不具合発生に他ならない。つまり、プロジェクト・マネージャは、各開発工程における不具合の発生を予測し、その対策を準備（必要に応じて仕様発注側と折衝）しながら、開発プロジェクト全体をマネージメントしなくてはならない。

しかしながら、近年のソフトウェアの大規模・複雑化により、プロジェクトの各開発工程レベルや、プロジェクト全体レベルで、現在どのような事が起きていて、将来どのような不具合が発生する可能性があるのかを、正確に予測することが困難になってきている。原因としては、開発するソフトウェアの規模が大きすぎて人間個人が認知・把握できる限界を超えている場合や、不具合発生の要因となり得るものとして観測できる事象もあれば観測できない事象（不具合が発生し実被害として表出するまで気付かない事象）もあり要因を明確化できない場合などが考えられる。

そこで本研究では、ソフトウェア開発工程における不具合発生の要因を、開発現場レベルで観測された不具合発生データに基づいた分析により明らかにし、ソフトウェア開発プロジェクトにおける不具合マネージメントを支援する方法を提案することを目的とする。方法としては、不確実性を含んだ事象に対しても適用可能なベイジアンネットワークを用い、観測データに基づいた不具合発生モデルを構築・評価することにより、不具合発生要因を探ることとした。

また、目的達成に向けたサブゴールとして、下記の3点を設定した。

(1) 組み込みソフトウェア開発の現場レベルでヒアリングを行い、現状の問題点を明確化し支援方法を提案すること

(2) 観測された不具合データを用いて不具合発生モデルを構築すること

（3）構築した不具合発生モデルの評価を通じて不具合発生の要因を分析すること

本稿の構成は、次のとおりである。本章では、研究背景としてソフトウェア開発における問題点を概観したのち、本研究の目的を示した。2章では、本研究ターゲットである組み込みソフトウェア開発現場の現状・課題をヒアリング調査の結果から明らかにし、支援の方法を探る。3章では、まず、ソフトウェア開発現場で収集した不具合発生データを機械学習の一つであるベイジアンネットワークの手法を用いてモデル構築をする。次に、構築したモデルの評価と、構築モデル上でのシミュレーションにより、不具合発生要因を明らかにする。4章で、本研究の成果をまとめ、今後の課題を示した後、本稿をむすぶ。

2. 組み込みソフトウェア開発プロジェクトの問題点

2.1 組み込みソフトウェア開発の特徴

組み込みソフトウェアとは、主にマイクロコンピュータが搭載された製品に組み込まれ、その製品の機能実現や制御を行うソフトウェアである。そのため、製品が使用される実環境からの影響も大きく受け、同時に処理時間やリソースに厳しい制約があり、安全面・信頼性の実現においても厳重に管理されている。また、組み込みソフトウェアはハードウェアデバイスと緊密に連携して動いているため、その開発には、多数のソフトウェア技術者とハードウェア技術者が共同で取り組むことになる。そのため、主に仕様書を中心とした技術者間のコミュニケーションが開発現場において大きな問題となることもある[4]。

ソフトウェア開発に関する問題点の指摘は、「ソフトウェア危機」[5]という言葉に発端を見ることができる。40年以上前に指摘されたことであるが、そこで指摘された（1）要素間の情報交換オーバーヘッドによる危機が存在すること、（2）開発に携わる技術者間の情報交換にも同様の危機が存在することは、むしろ、近年のシステムの大規模・複雑化によって、問題点がより顕著になってきている。そのため、ソフトウェア開発現場を支援するための研究・実践は、現時点の問題解決に向けた取り組みのみに留まらず、情報化社会の将来にとって重要な知見を探究することにつなげていかなくてはならない。

これまでも、ソフトウェア開発プロジェクトを支援する取り組みは多く存在する。例えば、リスクマネージメントの手法で実践したもの[6]、計数管理フレームワークにより定量的管理を試みたもの[7]、規模や工数、工期などの量的データと質的データからプロジェクト遂行に及ぼす影響要因を探ったもの[8]、プログラムコードの自動生成により支援を試みたもの[9][10]、協調フィルタリングを用いて開発工数を予測するもの[11]など、多数の試験的取り組みや実践報告がある。これらの報告からも、ソフトウェア開発プロジェクトの多くが同じような問題を抱え、苦労を味わいながらも、改善に向けた努力をしていることがよく判る。

2.2　ソフトウェア開発現場におけるヒアリング調査

本研究では、組み込みソフトウェア開発プロジェクトの現場で、現時点で抱えている問題点を明確にするためにヒアリング調査を実施した。この調査の結果をもとに、本研究目的へのアプローチを考えることにする。

調査期間、対象、主な内容は下記のとおりである。

期間：2010年5月〜2011年2月（計6回：1〜2ヶ月に一度）
対象：組み込みソフトウェア開発プロジェクト・マネージャ、技術者（計3名）
方法：グループインタビュー形式
内容：これまで担当した組み込みソフトウェア開発プロジェクトで発生した不具合・問題点と対処内容

ヒアリング調査は、研究打ち合わせ会議というかたちで集合してもらい、毎回、本研究の趣旨を説明し、調査の目的を十分理解してもらいながら、グループインタビュー形式（自由会話形式）で実施した。特に調査したかった内容は、ソフトウェア開発工程で発生した問題点や不具合内容と、発生した問題・不具合への対処内容であったが、グループインタビュー形式であったために、時によっては技術者としての悩みや苦労している点なども話題にのぼり、より広い範囲で、問題点や支援の必要性を知ることができた。

2.3　調査結果

ヒアリング調査によって得た結果は、(1) 不具合発生について、(2) 知識・ノウハウをマネージメントする対策について、(3) 開発現場が求める支援について、という3つのカテゴリーに分けることができた。以下、カテゴリー毎に簡単にまとめる。

(1) 不具合発生について

- 開発するソフトウェアの規模にもよるが、経験的に、新規プロジェクトでは1000件程度、継続・リソース流用のプロジェクトでは200件程度の不具合が発生する
- 大きな不具合ではなく、小さい不具合（ミス）が多く発生する
- 技術者個人レベルでも、プロジェクト全体レベルでも、同じようなところで同じような不具合（ミス）が発生していると経験的に感じる（データに基づいて話しているわけではないので真偽は判らないが）
- 不具合の情報は、不具合情報報告シートとして記述・保存してある
- 不具合情報報告シートに書かれた情報を（当該プロジェクトが終了後に）活用しているとはいえない

・どこに不具合・リスクがあるのか、なかなか見えないし予想も難しい
・プロジェクト進行中に、経験的に不具合が発生する可能性を感じる時もある

(2) 知識・ノウハウをマネージメントする対策について

・コーディングやドキュメンテーションのルール・テンプレートなどを作っているが、表現が稚拙であったり、記載要素が不十分であったりして、他者（記入者以外）がそのテンプレート記載のみから内容が理解できないことが多々ある
・技術者個人のスキルの差が、知識・技術の伝承では問題になっていると感じる
・知識・ノウハウをどんなに書き出してみても、それを受け取る側（読む側）がその準備ができていない（予備知識・経験がない）と、正しく受け取ってもらえないと思う
・不具合が発生後、「あの人に聞け」「あの本に書いてある」等の方法で、その場その場での対処に追われる場合が多い
・プロジェクト・チームでレビューすること（ソフトウェア開発においては、仕様書・設計書の査読やプログラムコードの内容をチェックすること）が、重要な対策として実施していることである
・いろいろな対策に取り組んでみたいが、低価格と短納期という根本的な問題があるために、結局、十分な取り組みができないのが現状である
・書籍等でリサーチした範囲では専任管理者を置くことが推奨されているが、中小の企業では会社の負担が大きく、現実的には困難である

(3) 開発現場が求める支援について

・技術者個人に適応した支援が必要と考えられる
・各開発工程で、システムから自動的に不具合発生を予測し指摘・警告してもらえれば、それだけでも大きな支援になる
・開発工程毎に、各種チェックリストを作っているので、それが支援に向けた貴重なデータ・情報になるのではないか
・チェックリストの内容をデータベース化して、修正箇所や不具合情報を共有できるようになれば、技術者間のコミュニケーションツールにもなり得る
・上記データベースから個々の技術者に対し、自動的に注意や警告などのアドバイスができると大きな助けになる

2.4 考察と提案

ヒアリング調査の結果から、まず、ソフトウェア開発現場においては（開発ソフトウェアの規模にもよるが）相当数の不具合が発生していることが確認できた。その多くが小さい不具合ということであったが、これら多数の不具合に対処するだけでも、技術者にとっては非

常に大変な作業である。この点からのみでも、ソフトウェア開発プロジェクトを支援することの必要性が判る。

また、不具合の発生情報は報告書として記録・保存はされているが、それらが有効活用されていないことも確認できた。過去の不具合発生データを、不具合発生時に、その場その場で調べる程度の活用はなされていたが、例えば、不具合を未然に防ぐための組織的な取り組みであったり、不具合発生の可能性を予測するなど、組織として取り組むべき対策の中に、過去の不具合データが有効活用されているとは言えないものであった。

知識やノウハウをマネージメントする点については、コーディングやドキュメンテーションのルール・テンプレートは作ってあるが十分とは言えない点、技術者個人のスキルや経験に大きく依存していると考えられている点などが問題として指摘できる。また、実践的な対策として、プロジェクト・チームでレビューを徹底して行っていることなども判った。

ソフトウェア開発現場では、要求仕様を満たすための作業と、その場その場の不具合の対処に追われることが中心であり、組織として知識やノウハウをマネージメントする取り組みは不十分であることが判った。もちろん、それらの対策に取り組みたいという意志や意欲は十分あるのだが、低コスト・短納期という現状では、開発作業以上の組織としての対策には手が回らないという事情も十分理解できた。実際、プロジェクト・マネージメントに関するリサーチから、理想としては専任管理者を置くことが推奨されていることも把握されていたが、現実としては（中小企業としては負担が大きく）困難であることを語ってくれた。これらは、理想と現実（理論と現場）のギャップと言ってもよいかもしれない。

現場が求めている支援については、技術者個人に適応したもの、不具合の発生を予測して警告を発してくれるもの、不具合情報の共有を促すコミュニケーションツール、などが求められていた。技術者個人に適応した支援となると、一般に、支援の仕組みの中に人の属性データを含ませる必要があるが、例えば、過去の不具合発生データに、その発生不具合の主体を加えることで対処できる可能性がある。誰が、どの工程で、どんな作業をしていた時に発生した不具合なのか。このような情報を蓄積すれば、そのデータを用いて対象主体に適応した支援ができると考える。

本研究では、開発現場を支援することを目標として位置付けている。そのため、ソフトウェア業界を支えている上記のような中小企業の開発現場を支援するために、方策の枠組みを提案しなくてはならないと考えた。そこで、ヒアリング調査の結果をもとに考案した、不具合情報を活用したソフトウェア開発現場の支援サイクル（図1）を提案する。

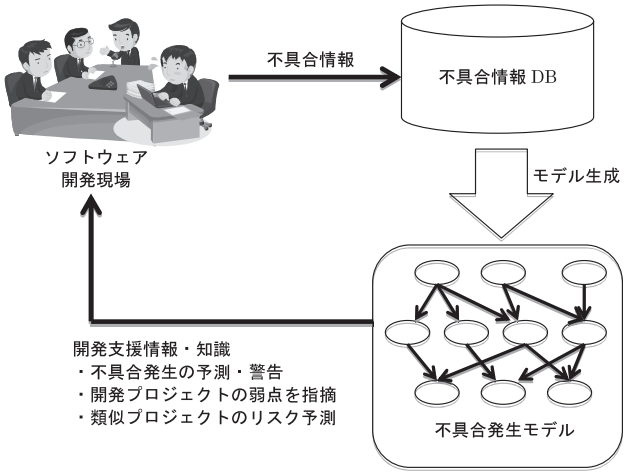

図1　不具合情報を活用したソフトウェア開発現場の支援サイクル

本支援サイクルは、ソフトウェア開発現場で発生した不具合情報を積極的に活用して、開発現場にフィードバックすることにより支援を行う点が基本コンセプトである。まず、開発現場で発生した不具合情報を紙シートではなく電子データ形式の不具合報告フォーマットに記入し、不具合情報データベース（DB）として蓄積する。不具合情報DBに蓄積された情報は、組織内で必要に応じて閲覧・共有することが可能であるが、より積極的な支援をするために、当該DB内の情報から不具合発生モデルを生成する。不具合発生モデルとは、ソフトウェア開発プロジェクトで発生した不具合情報に基づいて機械学習の手法により生成するもので、不具合が発生した因果関係を表現する数式モデルである。数式モデルであることによって、入力値を与えることによりシミュレーションすることが可能になり、進行中のプロジェクトや類似プロジェクトにおいて、不具合の発生を予測できるようになる。このように、モデル生成によって、DBに蓄積されていた個別の不具合データ群が組織化・知識化され、より高度な支援を開発現場にフィードバックすることが可能になる。

モデルを用いた具体的な支援として、以下の点が考えられる。

- 依存関係を表現するモデルであれば、不具合発生の要因候補が把握できる
- 因果関係の逆向き推論により、進行中プロジェクトの弱点（不具合原因）を予測・指摘できる
- 進行中プロジェクトの各工程における、発生リスクを予測できる
- 進行中プロジェクトの完成見通しが予測できる
- 複数プロジェクトでモデルを生成・比較し、各プロジェクトの特徴を把握できる（→新規プロジェクトの見積などの参考情報になる）

以上の支援が可能となれば、開発プロジェクトで発生し得るリスクの回避策を事前に準備でき、その結果として、不具合発生数の減少・工数の短縮・完成ソフトウェアの品質向上につながることと考えられる。また、これらの支援内容は、主にプロジェクト・マネージャの業務が支援の対象となっている。この点は、プロジェクトの専任管理者を置くことが困難な中小企業にとって、非常に有効な支援方策として実現されることが期待できる。

3. ベイジアンネットワークによるモデル構築と評価

前章で提案した、不具合情報を活用したソフトウェア開発現場の支援（図1）を実現するためには、不具合情報のモデル化が必要である。モデル化には、不具合報告データに基づいて構築する手法であること、因果関係を表現できるモデルであること、シミュレーション可能な計算モデルであることが、本研究で必要としている条件である。この条件を満たすものとして、本研究では、ベイジアンネットワークをモデル化の手法として採用した。

3.1　ベイジアンネットワーク

ベイジアンネットワーク（Bayesian Network）は、確率変数をノードとするグラフ構造で表現した、機械学習による計算モデルの一つである[12]。具体的には、複数の変数間の定性的な依存関係を有向グラフによって表し、変数間の定量的関係を条件付確率で定義した、非循環型有向グラフ構造の確率モデルである。条件付確率の分布は、取り得るすべての状態における条件付確率を並べた表（CPT：Conditional Probability Table）により定義される。

ベイジアンネットワークによるモデル構築過程では、実際に発生した事象（観測された事象）は確定値として当該確率変数に代入されるが、その際、ベイズの定理を連鎖的に用いて当該確率変数の前後のノード（確率変数）に確率が伝播する。確定値が代入される前の確率を事前確率、確定値の代入によりベイズの定理を用いて計算された後の確率を事後確率という。つまり、ベイジアンネットワークは、事前知識に基づいたモデル構築と、実際に観測された確定値からの学習（機械学習）によるモデル構築という両面を持ったモデルである。そのため、不完全なデータや観測が困難な要素を持った事象（不確実性を含んだ事象）に対しても適用することが可能な計算モデルといえる。

さらに、生成された非循環型有向グラフ構造として表現されたモデルは、変数間の依存・因果関係を視覚的に表しているため、人間にとって視覚的に理解しやすい確率モデルといえる。

3.2　ベイジアンネットワークによるモデル構築

ベイジアンネットワークは、収集した事象データのみから機械学習の手法（統計的学習）

によりモデルを一から構築することもできる。しかし一般に、事象として観測・収集されたデータ（確定値）が、直面している課題の全事象を表現しているとはいえない（質・量ともに必要十分な観測データ群を収集することは容易ではない）ため、機械学習からのみで正確なモデルを構築することは困難である。そこで、当該課題に携わっている人間の知識・経験を事前知識として初期モデルの構築に入れることで、高い精度でありかつ人間が理解・納得しやすいモデルを構築することにした。その後、初期モデルに対し、収集した事象データに基づく機械学習を適用することにより、シミュレーション可能な確率モデルとして構築することとした。

モデル構築の具体的手順は以下のとおりである。

（1）事前知識から三層構造モデルを定義：

ベテラン技術者の事前知識から、図2に示す不具合発生の三層モデルを定義した。これは、発生した不具合事象が、観測可能な表面的問題と一対一でつながっているわけではなく、それらの中間に観測困難な内在している問題があり、その問題が複数間で影響を受け合いながら不具合発生の可能性を上下させているという状態をモデル化したものである。現場の経験知識を元にしているため、非常に現実に即しており、また、ベイジアンネットワークにより計算モデルを構築する場合にも適した構造である。

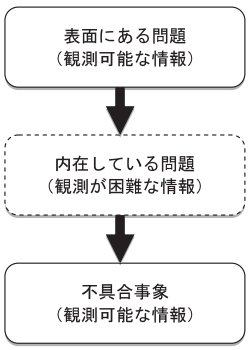

図2　不具合発生の三層構造モデル

（2）状態項目（変数）と主体の定義：

手順（1）で示した三層について、各層の中で取り得る状態項目とその主体を、ベテラン技術者の事前知識をもとに定義した（表1〜3）。各状態項目は、ベイジアンネットワークでモデル化する際のノード（変数）となり、すべてYes/Noで状態表現する（開発現場から不具合情報を収集する場合も、この状態項目に従って記述してもらう）。状態項目の中には、技術者個人の属性（性格やスキルレベル等）も含めることとした。これは、後々、構築したモデルを使用して個人に適応した支援を実現することを想定

しているためである。

また、各状態項目の主体を明示しておくことにより、生成したモデルを理解・分析する時に役立つことと考えた。

表1　状態項目と主体（第一層：表面にある問題）

状態項目（変数）	主体
理解力__低	プログラマー
作業ワークフロー違反	プログラマー
内向的な性格	プログラマー
思い込みが強い性格	プログラマー
経験年数__少	プロジェクト・チーム
リソース（人材）不足	プロジェクト・チーム
不測の事態が発生	プロジェクト・チーム
開発難易度__高	要求側（設計者）
プロジェクト規模__大	要求側（設計者）
仕様書の量__多	要求側（設計者）

表2　状態項目と主体（第二層：内在している問題）

状態項目（変数）	主体
仕様の理解不足	プログラマー
スキル不足	プログラマー
最終確認不足	プログラマー
意識の不統一	プロジェクト・チーム
打ち合わせが長過ぎる	プロジェクト・チーム
工数見積りの甘さ	プロジェクト・チーム
工程管理の甘さ	プロジェクト・チーム
作業負荷__大	プロジェクト・チーム
指示不足	プロジェクト・チーム
バージョン管理ミス	プロジェクト・チーム
報連相の頻度__低	プロジェクト・チーム
ルールが不明確	プロジェクト・チーム
レビューの人選ミス	プロジェクト・チーム
レビュー不足	プロジェクト・チーム
エビデンスが残っている	プロジェクト・チーム
リリース物の内容不備	要求側（設計者）
レスポンスの悪さ	要求側（設計者）
仕様書の誤記	要求側（設計者）
仕様書の不備（記述漏れ）	要求側（設計者）
度重なる仕様変更	要求側（設計者）

表3　状態項目と主体（第三層：不具合事象）

状態項目（不具合）	主体
仕様の誤認	プログラマー
コーディング・ミス	プログラマー
考慮漏れ	プログラマー
コード流用時の確認漏れ	プログラマー
処理追加前評価（未実装項目）	プログラマー
仕様漏れ	プログラマー
修正による新規不具合	プロジェクト・チーム
工程遅延	プロジェクト・チーム
デグレ発生	プロジェクト・チーム
提出物の不足	プロジェクト・チーム
外部要因	要求側（設計者）

（3）状態項目間の関係性の定義（初期モデルの構築）：

ベテラン技術者の事前知識から、手順（2）で定義した各層の状態項目間の定性的関係を定義した。簡単に言えば、第一層の各状態項目と第二層の各状態項目の間で関係性があると判断できるものすべてに有向リンク（第一層から第二層へ）を、第二層の各状態項目と第三層の各状態項目の間で関係性があると判断できるものすべてに有向リンク（第二層から第三層へ）をつけたということである。これにより、初期モデルとして三層構造の非循環型有向グラフが構築される。

（4）収集データ（確定値）からCPT作成：

手順（3）で構築した初期モデルのノード（状態項目：確率変数）に、不具合発生情報として収集したデータ（確定値）から条件付確率表（CPT）を作成し代入する。その際、ベイズの定理を連鎖的に用いて当該確率変数の前後のノード（確率変数）に確率を伝播させる。ここが、ベイジアンネットワークのモデル構築方法の中心である。

（5）モデルの最適化：

データに基づいた計算モデルには過適合問題（Overfitting）が存在する。データに対して適合し過ぎると、データ内のノイズ部分にも適合した複雑なモデルとなってしまい、計算モデルとしては不適となる。そのため、モデルの複雑さとデータ適合のバランスを取ることにより、モデルの最適化をする必要がある。本研究では、AIC（Akaike’s Information Criterion）により最適化したモデルを、最終生成モデルとする。

以上の手順によって生成した不具合発生モデルを用いて、モデルで表現された因果関係（有

向グラフ）を分析することにより不具合発生要因を分析する。分析の結果と考察を次節で述べる。

3.3　モデルの評価とシミュレーション結果の解釈

ソフトウェア開発の工程は、図3のようにV字モデルとして表現できる。これは、各設計工程で要求されるソフトウェア仕様を検証するために、設計工程とテスト工程を対応付けて表現したものである。一般的にソフトウェア開発プロジェクトは、このモデルに従って進められる。

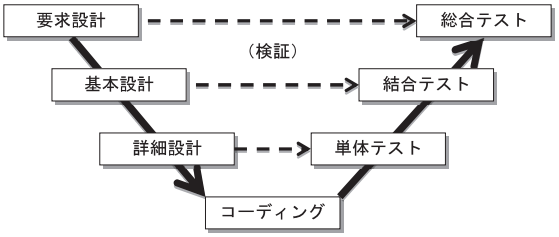

図3　ソフトウェア開発工程のV字モデル（設計工程とテスト工程の対応）

本研究では、3つのテスト工程毎に発生した不具合を分けてモデル化を行った。テスト工程毎に分けた理由は、工程毎に発生する不具合に特徴・違いがあると考えたためである。できるだけ、現場の特徴に適応した支援を目標としているため、このような対処をした。

モデル生成に用いた不具合発生データは、2011～2012年に実施された組み込みソフトウェア開発プロジェクトにおいて収集したものである。各工程における不具合発生データ数は、下記のとおりである。

・単体テスト工程：206件
・結合テスト工程：77件
・総合テスト工程：190件

以下、各テスト工程に分けて生成した不具合発生モデルについて、結果を提示し考察を加える。

3.3.1　単体テスト工程

単体テストにおいて発生した不具合情報から生成されたモデルを図4に示す。

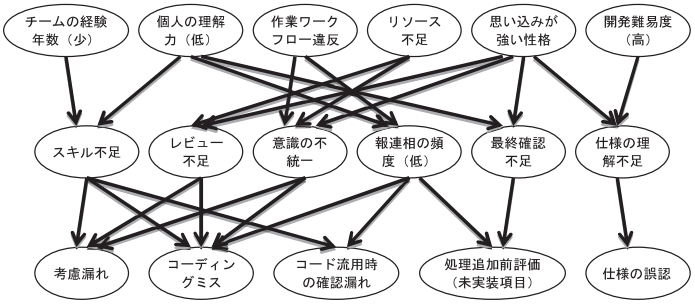

図4　単体テスト工程の不具合発生モデル（データ数：206件）

この工程は、ソフトウェアのモジュール（部品）単位での検証テストであり、ソフトウェアを実現するための最初のテストとなる。そのため、テスト数（最低でもモジュール数必要のため）が多くなり、発生する不具合数も必然的に多くなる。本研究で収集したデータは206件であり、3つのテスト工程中で最大数であった。

まず、生成されたモデルから、状態項目の主体について（図4と表1を対応させながら）考察する。第一層のプログラマー個人（3項目）とプロジェクト・チーム（2項目）と要求側（1項目）を主体とする問題が、結果として（有向グラフの行き着く先として)、第三層のプログラマー個人（5項目）を主体とする不具合のみに発生が集約されている。つまり、このモデルからは、不具合発生の原因となる問題（第一層）には3主体が関わっているが、不具合発生事象（第三層）としては主体がプログラマー個人のみに関わっている。つまり、プログラマー個人への不具合対処負担が集中している可能性があると読み取れる。以上により、この工程では、プログラマー個人に対する適切な支援が必要であることを示していると考えられる。

また第二層の状態項目の主体については、プログラマー個人（3項目)、プロジェクト・チーム（2項目）であり、プログラマー個人だけでなく、プロジェクト・チームとしての作業の重要性も示している。第二層は内在している（隠れた）問題点を示しているため、不具合としては表面上見えにくいものである。しかしながら、本モデルを読み解くことにより、レビューや意識の統一、報連相の頻度など、たとえ単体モジュールを開発している段階であっても、チームとしてプロジェクトを動かすことの重要性を確認することができた。

次に、要因となる項目の確率値を変動させ（第一層の入力値に変化を与え)、その計算結果として不具合事象（第三層）の発生確率にどれくらい影響が出るのかを、生成したモデル上でシミュレーションすることにより導き出す。その結果として、影響が大きかった項目の組み合わせを、「ルール（知識)」として示すことができる。本工程においては、以下のルールが抽出できた。各ルールに対し、簡単に考察を加える。

・「開発難易度が高い」と「仕様書の誤認」の発生確率が高くなる

これは、納得できるルールである。通常、要求側（設計者）が出してきた仕様書から開発難易度が判断されるが、生成モデルから、「開発難易度」は中間層の「仕様の理解」に影響を与え、そして第三層の「仕様の誤認」に影響を与えていることが見て取れる。シミュレーションにおいても、その関係性が強く出たことは、納得できる結果である。

このように、生成モデルから視覚的に理解できる部分と、シミュレーションの結果から理解できる部分が、同一の解釈として成り立つ場合には、このルールは信頼できるものと判断できる。

・「担当者の思い込みが強い」と「仕様書の誤認」の発生確率が低くなる

このルールは、ノイズの可能性があると考えられる。通常、思い込みが強い性格の人だと、他者の意見や要求仕様に対する理解不足または誤解したまま作業を続けて、その結果、不具合を発生させてしまうと考えられる。ここで抽出されたルールでは、「思い込みが強い技術者の方が仕様書を誤認する可能性が低い」というように、我々の経験から導き出される一般的ルールとは異なったルールとなっている。もちろん、他の工程での生成モデルや、他の開発プロジェクトでの生成モデルと比較する等して、ルールを再度検証する必要があると思うが、現段階においては、モデルの過適合（ノイズに適合）が発生してしまっている可能性があると考察する。

このように、実際に観測されたデータから機械学習によってモデル・ルールを生成する場合、一見、人間の経験則からは納得できないルールも抽出される。このような場合、モデルの過適合であることが多い。しかしながら、このようなルールが、人間の経験則からは見えない（隠れた）新たな知見である場合もある。この点が、データに基づいて生成されたモデルやルールを評価する上で難しい点でもあるが、新たな知見に出会う可能性を秘めている、非常に興味深い点でもある。

・「作業ワークフローを違反する」と「コーディングミス」の発生確率が高くなる

これは、納得できるルールである。プログラマー個人が、組織で決めた作業ルールに違反していることに気付かない状態で作業を進めていることから（場合によっては違反承知で進めていることもあるかもしれない)、様々なミスにつながることは想像できる。このルールでは、コーディングという、プログラマーの中心作業でのミスにつながっている。作業ワークフローとは、このようなミスを未然に防ぐ目的で規定されている。その意味において、本抽出ルールは核心を衝いていると考えられる。

・「作業ワークフローを違反する」と「コード流用時の確認漏れ」の確率が高くなる

これも、上記ルールと同様、作業ワークフローを遵守することの重要性が示された、納得できるルールである。コード流用とは、過去に開発した（実績のある）類似プログラムコードを流用することであるが、その際、入出力スコープや型などを確認して、当該プロジェクト用に一部修正しないと正しく動作しない。そのため、流用時に確認すべき作業ワークフローがあるはずである。作業ワークフローを違反する傾向のあるプログラマーは、このような不具合を発生させる可能性が高い、という解りやすいルールである。

・「作業ワークフローを違反する」「チームの経験年数少」「担当者の思い込みが強い」の3条件が揃うと、「コーディングミス」の発生確率が高くなる

これも、上記2つのルールと同様に納得できるルールである。この3条件は、それぞれが単独であっても少なからず不具合発生に影響を与えていると考えられるが、3つ揃った場合には、「コーディングミス」という不具合の発生に大きく影響を与えていることが判明した。プロジェクト・チームを構成する場合には、これらの要素を事前にマネージメントすることが、不具合を減少させるため（未然に防ぐため）の要点の一つと考えられる。

・「リソース不足」のときに「考慮漏れ」の発生確率が高くなる

まず「考慮漏れ」という事象であるが、これは「短納期でじっくりと考慮できなかった」という点と、発生したソフトウェア上の不具合について、その原因特定が難しい場合にも「考慮漏れ」として不具合報告している点、その両方を含んでいることが、技術者からの説明にあった。つまり、単に「プログラミング時に考慮（する時間）が足りなかった」のか、「発生してしまった不具合の原因が特定できなかったので、よく考え直して再度プログラミングして不具合修正した」のか、またはその両方なのか、という複数の状況を表している。そのため、この抽出ルールを、「リソース不足（人材不足）のために、開発に携わるプログラマー個人に負荷が大きくかかり、その結果、時間をかけて考慮することができなかった」という解釈をすることも可能だし、「リソース不足（人材不足）のために、発生した不具合の原因を特定することができなくて、再度、一から考え直すことになった」という解釈も可能である。ただし、「リソース不足」というプロジェクト・チームを主体とする問題が、プログラマー個人の問題・負担となって影響していることは、明らかである。そして、人材確保という、ソフトウェア開発業界全体が抱える問題を表したルールであるとも言える。

以上が、単体テスト工程における不具合発生モデル、およびシミュレーションで抽出したルールに対する考察である。生成されたモデル全体としては、非常に納得できるモデルでありルールである。一部、再検討が必要なルールがあったが、上述したように、これがデータ

に基づいた機械学習によるモデル生成の特徴でもあり、このようなルールが新たな知見になる可能性もある。

3.3.2　結合テスト工程

結合テストにおいて発生した不具合情報から生成されたモデルを図5に示す。

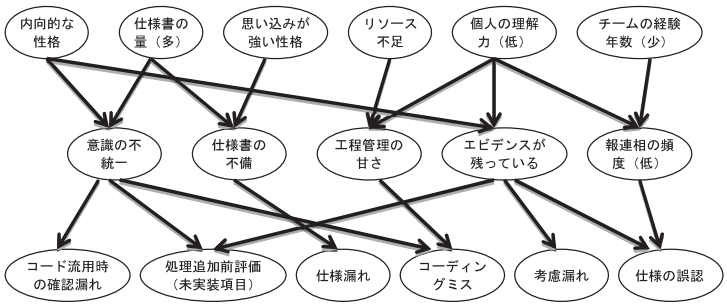

図5　結合テスト工程の不具合発生モデル（データ数：77件）

この工程は、ソフトウェアのモジュール（部品）を結合し、基本設計で要求されている機能を検証するテスト工程である。単体テスト工程を経てモジュール単位の欠陥は修正されているが、モジュール同士の相性や入出力関係の適合などが複雑に影響し合うために、この工程のテストも容易ではない。場合によっては、単体テストが不十分なまま結合テスト工程に来てしまう場合もあり、このような可能性に対する配慮も必要となる。

本研究で収集したデータは77件であり、比較的少ない量であった。観測された不具合が少ないことは、ソフトウェア開発プロジェクトにとっては素晴らしことである。しかし、研究として不具合発生モデルを生成するという観点からは、モデルの精度を低くしてしまう可能性があることと考えられる。この点を踏まえ、以下で考察を述べる。

まず、生成モデルから、状態項目の主体について（図5と表2を対応させながら）考察する。第一層のプログラマー個人（3項目）とプロジェクト・チーム（2項目）と要求側（1項目）を主体とする問題が、結果として、第三層のプログラマー個人（5項目）を主体とする不具合のみに発生が集約されているという、単体テスト工程と同様の特徴が出ている。つまり、この工程のモデルからも、不具合発生の原因となる問題には3主体が関わっているが、発生した不具合事象については、その主体はプログラマー個人のみに関わっているという、プログラマー個人への負担集中という可能性を示している。

また中間である第二層の状態項目の主体については、プロジェクト・チーム（4項目）、仕様要求側（1項目）となっており、個人よりチームを主体とする、結合テストという工程を特徴付けたものとなっている。上述したように、結合テスト工程は、単体モジュールを結合

するという、部品の組み合わせテストである。同一のプログラマーが作っている単体モジュールを組み合わせる場合もあれば、他のプログラマーが作っているモジュールを組み合わせる場合もあるのが、このテスト工程の作業である。モデルを見れば、プロジェクト・チームが主体となる隠れた問題点が、この結合テスト工程では大きく影響していることが理解できる。チームとしてプロジェクトを動かすことの重要性を示していることと考えられる。

次に、生成したモデル上でシミュレーションすることによって導き出した「ルール」を以下に示し、考察を加える。

・「リソース不足」「理解力が低い」「担当者の性格が内向的」の3条件が揃うと、「コーディングミス」の発生確率が高くなる

これは納得できるルールである。チームとして人材不足の状況で、技術者の理解力が低く、かつ内向的な性格のため他者とのコミュニケーションが不十分である状況においては、ソフトウェア開発がうまく進まない。特に、この抽出ルールからはプログラマー個人を主体とする、コーディングにミスが発生するということである。本工程が結合テストという点から推察すると、単体モジュールとしては動作するものを作ったが、モジュールの組み合わせ部分での理解力不足とコミュニケーション不足がコーディングミスにつながる可能性が大きいことを示していると考えられる。

・「担当者の思い込みが強い」と「仕様漏れ」の発生確率が高くなる

このルールも納得できるものである。仕様漏れとは、必要な機能が実現されていない（仕様の要求を満たしていない）というものである。プログラマー個人の「思い込みの強さ」が、プログラム開発に悪影響を及ぼす可能性があることを示している。

・「担当者の性格が内向的」「理解力が低い」の2条件が揃うと、「考慮漏れ」の発生確率が高くなる

単体テスト工程の考察でも述べたが、「考慮漏れ」とは、単に「プログラミング時に考慮できなかった」のか、「発生してしまった不具合の原因が特定できなかった」のか、またはその両方なのか、という複数の状況を表している。プログラマー個人の内向的な性格や理解力が、「考慮漏れ」につながることを示した本ルールは十分納得できる。特に本テスト工程においては、チームとしての作業が不具合発生に大きく影響する可能性が高いため、これら2条件に重要性があることは理解できる。

・「担当者の性格が内向的」のとき「コード流用時の確認漏れ」の可能性が高くなる

このルールも、上記ルールと同様、納得できるものである。コード流用時の確認作業に

ついては、単体テスト工程のルール考察の部分で述べた。本ルールにおいて、プログラマー個人の内向的な性格によるコミュニケーション不足が、確認漏れを起こす可能性を高めてしまうことは十分納得できるルールである。

以上が、本テスト工程の生成モデルおよび、各ルールの考察である。本工程は、単体モジュールを結合する、プロジェクト・チームとしての活動が中心となる工程である。そのため、チーム活動を主体とした項目が生成モデルには多く出現した。抽出ルールについても、不具合発生事象としてはプログラマー個人を主体とするものであっても、それらがチーム活動に影響を及ぼす可能性が高いと考えられる項目が多く出現した。そのため、全体的に納得できるものが生成・抽出できたと言える。

また、本工程で収集できたデータ数は決して多くなかった（77件）が、生成モデル、抽出ルールともに納得できるものが得られたという点から、本テスト工程に典型的な不具合発生データ（つまりノイズが少ないデータ）が収集できたのであろうと推測する。そのため、本モデルを類似プロジェクトの不具合発生予測として使用する場合においても、非常に有用な（正確に予測できる可能性が高い）モデルになっていることと考えられる。

さらに、典型的なデータから典型的な結果を得たということは、分析手法としての適切さも示すことにもつながる。この点からも、非常に満足できる結果である。

3.3.3　総合テスト工程

総合テストにおいて発生した不具合情報から生成されたモデルを図6に示す。

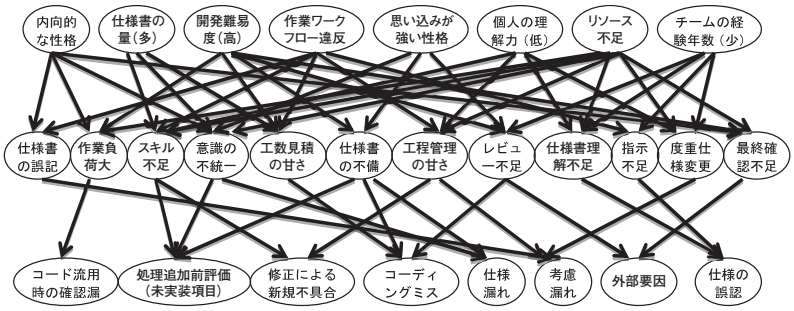

図6　総合テスト工程の不具合発生モデル（データ数：190件）

この工程は、結合テスト工程を経て基本設計の要求を満たしたプログラム群を組み合わせて、システム全体としての要求設計を満たしているのかどうかを検証するテスト工程である。システムとして完成したか否かを総合的に判断するテストであり、厳密かつ多くのテストを実施しなくてはならない。特に組み込みソフトウェアの場合、信頼性を保証するために、想

定外の操作や環境の影響なども含めたテスト項目を作成して実施する必要がある。そのため、必然的にここで発生する不具合は多くなる傾向にある。

本研究で収集した不具合発生データは190件であった。不具合数として決して少なくはないが、本テスト工程の性格上、決して多過ぎるともいえない量である。

生成されたモデル（図6）について考察をすすめる。モデルを一見して、これまでのテスト工程（単体テスト、結合テスト）とは明らかに異なったモデルになっていることが判る。まず、前2つのテスト工程における生成モデル（図4、図5）と比較して、本テスト工程での生成モデルは項目数が多い。特に、中間である第二層の項目数が非常に多くなっている。これは、本テスト工程が、ソフトウェアの完成に直結する多種多様なテストを必要としており、その分、不具合発生の要因も複雑な構造をしていることを表現していると考えることができる。不具合発生件数のみを比較すると、本テスト工程の190件は、単体テスト工程の206件に近いが、生成されたモデルの複雑さは明らかに違う（図4と図6を比較）。

また、状態項目の主体を図6と表3を対応させながら読むと、第一層のプログラマー個人（4項目）とプロジェクト・チーム（2項目）と要求側（2項目）を主体とする問題が、結果として、第三層のプログラマー個人（6項目）、プロジェクト・チーム（1項目）、要求側（1項目）を主体とする不具合の発生に影響を与えていることが判る。前2つのテスト工程と同様に、プログラマーを主体とする項目が多いが、第三層に現れている項目の主体には、これまで（前2つのテスト工程では）現れていなかった、プロジェクト・チームを主体としたものと要求側を主体としたものが含まれている。このことから、最終的にはプログラマー個人に不具合発生の負担が大きく影響することは前2つのテスト工程と同様であるが、それに加え、プロジェクト・チームと要求側を主体とする不具合も発生する可能性があるということが、本テスト工程の特徴の一つであると考えられる。

12項目も出現した第二層の状態項目の主体については、プログラマー個人（3項目）、プロジェクト・チーム（6項目）、仕様要求側（3項目）となっている。これも明らかに前2つのテスト工程のモデルとは異なり、本テスト工程がプロジェクト・チームが中心となってテスト作業を実施していることを表しているものと考える。

また、仕様要求側の問題も大きく関わっていることも、本工程のモデルの特徴である。要求側が出したソフトウェア仕様に対して、完成したものを出す直前のテストであることを考えれば、当然の結果であると思われる。

次に、生成したモデル上でシミュレーションすることによって抽出した「ルール」を以下に示し、考察を加える。

・「個人の理解力が低い」「チームの経験年数が少」の条件が揃うとき「仕様書の誤認」の発生確率が高くなる

このルールは十分納得できるものである。個人レベルでの理解力と、チームレベルでの経験年数が、仕様書誤認という不具合発生の要因になることは容易に理解できる。このルールは、理解力が低いプログラマーをプロジェクト・チームに入れた場合、それをカバーできるような経験年数を持つベテラン技術者もプロジェクト・チームに入れる等の対策で、不具合を未然に防ぐことも可能となることも示している。

・「ワークフロー違反」「思い込みが強い」の条件が揃うとき、「コーディングミス」の発生確率は高くなる

このルールも納得できるものである。作業ワークフローは、上述したとおり、プロジェクト内での作業がスムーズに進むように規定された、組織内のルールである。その組織内ルールを違反し、かつプログラマー個人の思い込みが強い場合には、ミスにつながるという、非常に分かり易いルールである。

・「作業ワークフロー違反」「個人の理解力が低い」「リソース不足」の3条件が揃うと、「修正による新規不具合」の発生確率が高くなる

このルールも、非常に分かり易く、納得できるものである。プログラムの一部を修正することによって新たな不具合を発生させてしまうということは、ソフトウェアを開発していると必ず経験することである。その要因が、作業ワークフロー違反、理解力不足、人材不足の3条件が揃った時である、ということに疑問はない。この3条件が揃う時ということは、プログラマー個人としても、プロジェクト・チームとしても、開発作業に非常に苦しんでいる時である。プロジェクト・チームを組む時、もしくは、このような不具合の発生が予想された時に、不具合の発生を防ぐためには、プログラマー個人とプロジェクト・チームの双方に対して配慮が必要だということが、このルールから判る。

・「仕様書の量が多い」「思い込みが強い性格」「作業ワークフロー違反」の3条件が揃うと、「処理追加前の評価」の発生確率が高くなる

このルールも納得できるものである。未実装項目について処理を追加する必要が生じた際、その処理を追加する前に評価を行う必要があるが、それがなされなかったという不具合事象である。要求側が出してくる仕様書の量が多いこと、プログラマー個人が思い込みの強い性格であること、作業ワークフロー違反をする、の3条件が揃う時とは、仕様書の量が多い（大規模な）プロジェクトに対して、プログラマー個人が自らの思い込みで突っ走ってしまうような状況を表しているのではないかと考えられる。このような状況で、追加前評価という、一歩立ち止まって検討する作業がなされずに不具合に至るということは、想像に難くない。

・「リソースは不足していない」が「ワークフロー違反」をしたとき、「考慮漏れ」の発生確率が高くなる

このルールについては、少々疑問点が存在する。一見、納得できそうなルールであるが、条件が2つ揃った時に発生する点に疑問がある。つまり、「ワークフロー違反」のとき「考慮漏れ」が発生する可能性があることは納得できるが、「リソース（人材）は不足していない」という条件が加わったときに、発生確率が上がることがシミュレーションによって示されている。強引に解釈すれば、「人材が充足しているために気持ちに余裕ができてしまい、それが油断につながってワークフロー違反をしてしまい、その結果として考慮漏れが発生してしまった」ということであろうか。それとも、機械学習による過適合による影響なのか。このルールについては、再検討の余地がある。

・「担当者が内向的」で「リソース不足」しかし「ワークフロー違反していない」とき、「コード流用時の確認漏れ」の発生確率が高くなる

このルールに関しても、上記ルールと同様に疑問がある。「担当者が内向的」で「リソース不足」という条件に「作業ワークフローを違反していない」が加わった時に不具合が発生する可能性が上昇するというルールである。これも強引に解釈すれば、「人材不足であるが、ワークフローを違反していないという慢心と担当者が内向的でコミュニケーションが不足気味のとき、確認漏れという不具合を引き起こしている」とも解釈できるが、やはり、再検討が必要なルールであることには違いない。

以上が、本テスト工程の生成モデルと各ルールに対する考察である。本テスト工程の不具合発生モデルは、他の2つのテスト工程に比べると複雑であることが、図6で示したモデルからも、シミュレーションで抽出したルールからも判断できた。一部の抽出ルールには疑問が残り、再検討を要するルールも存在するが、機械学習手法の特有の「データへの過適合」の可能性を考慮に入れれば、本研究手法の特徴が出たともいえる。

以上により、本研究での不具合発生モデルおよび抽出ルールは、全体的には納得できる部分が多く適切なものであると評価できる。

3.4 本章の考察

本章では、ベイジアンネットワークを用いることにより、実際にプログラム開発プロジェクトのテスト工程において観測・収集された不具合データから、不具合発生モデルを構築した。構築したモデルは、3つのテスト工程（単体テスト、結合テスト、総合テスト）の特徴をよく表しており、そこに表現された不具合発生の因果関係も納得できるものであった。

また、構築したモデルを用いたシミュレーションによって得られたルールについても、概ね納得できるルールであった。3つのテスト工程合わせて16ルールを抽出することができ、

そのうち13ルールは十分納得できる有用なものであった。残りの3ルールについては、ノイズに過適合してしまった可能性もあると判断できたが、解釈によっては、新たな知見としての可能性も残している。この3ルールについては、現場の技術者を交えて再検討しようと考えている。このように、人間の経験則からは、すぐに思いつかないようなルールに出会うことも、（たとえそのルールがノイズ過適合であったとしても）機械学習という手法の利点であり、非常に興味深い点でもある。

不具合を発生させる要因としては、プログラマー個人を主体とするものが全体的に多く出現していた。特に単体テスト工程では、その多くがプログラマー個人に起因していると考えられた。他のテスト工程でも、最終的に不具合として発生してしまう事象（第三層）の主体の多くは、プログラマー個人であった。このことから、プログラマー個人をいかに支援するのかという点が、本研究課題（ソフトウェア開発現場を支援すること）を解決するポイントであることと考えられる。プログラマーを支援しない限り、不具合の発生は抑えられないと考えてもよいのかもしれない。

ただし、内在している問題（第二層）としては、プロジェクト・チームを主体とするものが多く出現していた。このことは、現在のソフトウェア開発体制が、チームによって動いていることを示唆している。そのため、プロジェクト・チームや、そのチームを管理しているプロジェクト・マネージャも、非常に重要な支援対象として位置付けることができる。

更に考察を深めると、内在している問題（第二層）の多くは、第三層のプログラマーを主体とする不具合発生事象に強くつながっていたことから、内在している問題（チームを主体とする問題）の多くを抑えることができれば、それが第三層のプログラマー主体とする不具合の発生を抑えることにつながるとも言える。つまり、プロジェクト・チームの対策によって、プログラマー個人を支援できる可能性があるという関係性を示している。

また、モデル構築手法として、ベイジアンネットワークという手法も、本研究課題にとって非常に有用なものであった。一番の理由は、構築されたモデルの分かりやすさである。本研究では、三層の非循環型有向グラフによってモデル表現したが、その視覚的理解の容易さは、非常に優れていた。特に、因果関係を表現できる点は、不具合発生の要因を明らかにするという本研究課題に非常に適しており、その分析に大きく貢献したと言える。

ベイジアンネットワーク技術は、近年の情報処理技術の向上により、その応用・適用範囲が広がりつつある[12]。本研究と同様、ソフトウェア開発分野に適用した事例もある。例えば、ソフトウェア開発工程の最終品質予測モデルを提案したもの[13]、オープンソースソフトウェアの信頼性評価に利用したもの[14]、品質特性と実装技術の因果関係をモデル化して実装技術の選択を支援するもの[15]、などがある。どれも、ソフトウェア開発に対する改善・支援をターゲットにした取り組みであるが、モデル化の項目（要素）や支援ターゲットなど、少しずつ異なっている。いろいろな取り組み事例から、多くの優れた支援・改善策が示されることが今後も期待できる。

4. おわりに

本稿では、組み込みソフトウェア開発の現場を支援することを目標に、ソフトウェア開発プロジェクトにおける不具合マネージメントを支援する方法について探究した。ここでは、1章で設定した、3つのサブゴールに対応付けて、本研究の成果をまとめる。

(1) 組み込みソフトウェア開発の現場レベルでヒアリングを行い、現状の問題点を明確化し支援方法を提案した

ソフトウェア開発現場に対するヒアリング調査により、不具合発生状況、知識・ノウハウのマネージメント対策、現場として必要だと考えられる支援策についてまとめた。その結果を受けて、不具合発生時に記録する不具合データを積極的に活用して、開発現場に有用な情報をフィードバックすることを基本コンセプトとした、ソフトウェア開発現場の支援サイクルを提案した。支援サイクルが繰り返し回転することにより、過去に発生した不具合のデータが、進行中や次期のソフトウェア開発現場に有効に活かされていくという支援モデルである。

(2) 観測された不具合データを用いて不具合発生モデルを構築した

上記（1）で提案した支援モデルを実現するためには、観測された不具合データに基づいて計算モデルを構築しなくてはならない。そこで、機械学習の一手法であるベイジアンネットワークを用いて、観測された不具合発生データに基づいた計算モデルを構築した。ベイジアンネットワークによって構築したモデルは、因果関係をグラフ構造で示した階層型非循環有向グラフであるため、視覚的に理解しやすいモデルであり、かつ、シミュレーション可能な計算モデルでもある。このことは、ベイジアンネットワークによって構築したモデルが、不具合発生マネージメントの支援サイクルにつながる（容易に使用出来る）可能性を示している。

(3) 構築した不具合発生モデルの評価を通じて不具合発生の要因を分析した

ベイジアンネットワークによって構築したモデル内で表現されている因果関係の分析と、モデル上でのシミュレーションから導き出されたルールを分析することによって、不具合発生の要因を探った。その結果として、モデルで表された因果関係は経験的に納得できるものであること、不具合要因の多くがプログラマー個人に起因していること、内在している（隠れた）問題の多くはプロジェクト・チームが主体となっていること、シミュレーションによって得られたルールは、ほとんどが有用かつ納得できるルールであったが一部再検

討が必要なルールが抽出されたこと、などが得られた。

今後の課題として、まず、本研究で得られた不具合発生モデルを、新規開発プロジェクトの工程に適用して不具合発生を予測し、そのプロジェクト全体の流れを観察・評価すること等により、不具合予測の精度を検討することが必要だと考える。

次に、より実用的なモデル生成に向けたモデルのブラッシュアップが必要である。具体的には、複数のプロジェクトにおいて同様の不具合データを収集・分析し、本手法の汎用性・信頼性を評価したり、モデルの各ノードである状態項目を見直したり、曖昧な項目については再定義したりするなど、モデルの再構築・更新を随時進める必要がある。

また、他の手法で生成したモデルとの比較も必要になるかもしれない。本研究では、ベイジアンネットワークを用いてモデルを生成し分析したが、同様の機械学習の手法は他にも複数ある。それぞれ適用範囲や適した題材などに違いがあるが、本研究の題材としては、どの手法のどんなモデルが最適なのかを検討することは意義深いことと考える。

情報化社会を支えているのは、様々なソフトウェア技術である。そのソフトウェアを開発している現場は、日々、ソフトウェアに対する要求の大規模・複雑化、短納期、低コストへの対応に四苦八苦している。本研究は、そのようなソフトウェア開発の現場のプログラマーやプロジェクト・マネージャ、開発企業を少しでも支援したいという思いのもと実施したものである。このような開発現場を支援することは、単にソフトウェア開発企業のみを支援することに留まらず、情報化社会の基盤を支えることにつながっていると考えている。研究のための研究ではなく、開発現場を支援するための研究として、本研究で残された課題にも継続的に取り組む必要があると考える。

今後も、多くのソフトウェア開発現場と連携・協力しながら、その現場を支援するための方策を提案できれば本懐である。

謝辞

本稿は、共同研究「ソフトウェア開発プロジェクトにおける不具合発生予測モデルの研究」の成果の一部をまとめたものである。本研究プロジェクトにご協力いただいた株式会社ロジックデザイン、渡邉秀人氏、岩崎徹氏、中庭伊織氏に謝意を表する。

参考文献

[1] 経済産業省Web：組込みソフトウェア産業実態調査報告書について（http://www.meti.go.jp/policy/mono_info_service/joho/ESIR/）（2013年5月12日最終閲覧）

[2] 独立行政法人情報処理推進機構（2013）「2012年度ソフトウェア産業の実態把握に関する調査報告書」（http://sec.ipa.go.jp/reports/20130426/reports_20130426_02.pdf)(2013年5月20日最終閲覧）

[3] Project Management Institute (2013) “A Guide to the Project Management Body of Knowledge (PMBOKGuide) Fifth Edition”, Project Management Institute.

[4] 菅谷克行（2009）組み込みソフトウェア開発の問題点の分析－技術者間のコミュニケーションに注目して－，茨城大学人文学部紀要『人文コミュニケーション学科論集』6, 137-157

[5] 高橋秀俊（1969）ソフトウェア危機, 情報処理 10(6), 373-374

[6] 工藤孝一（2012）ソフトウェア開発プロジェクトにおけるリスクマネジメントの実践, プロジェクトマネジメント学会誌 14(4), 15-19

[7] 梶山昌之, 合田英二, 千野智子（2011）ソフトウェア開発プロジェクトの計数管理フレームワークによる定量的管理, プロジェクトマネジメント学会誌 13(5), 3-8

[8] 古山恒夫, 菊地奈穂美, 安田守, 鶴保征城（2007）ソフトウェア開発プロジェクトの遂行に影響を与える要因の分析, 情報処理学会論文誌 48(8), 2608-2619

[9] 柿崎成章（2002）エンジン制御開発における自動コード生成技術の適用, 計測と制御 41(2), 147-150

[10] Ohsuga S., Chigusa S., and Sugaya K. (2000) , “Model Based Program Specification and Program Generation -In a Case of Vehicle-Engine Control System Design-”, Advances in Artificial Intelligence, PRICAI 2000 Workshop Reader, Springer-Verlag, Berlin.

[11] 角田雅照, 大杉直樹, 門田暁人, 松本健一, 佐藤慎一（2005）協調フィルタリングを用いたソフトウェア開発工数予測方法, 情報処理学会論文誌 46(5), 1155-1164

[12] 本村陽一, 岩崎弘利（2006）ベイジアンネットワーク技術, 東京電機大学出版局

[13] 天嵜聡介, 水野修, 菊野亨（2003）ベイジアンネットに基づくソフトウェア開発工程の最終品質予測モデルの提案, 電子情報通信学会技術研究報告 SS102(617), 19-24

[14] 田村慶信, 竹原英秀, 山田茂（2009）組込みオープンソースソフトウェアに対する移植性評価法に関する一考察, 電子情報通信学会技術研究報告 R信頼性109(67), 13-17

[15] 風戸広史, 林晋平, 小林隆志, 佐伯元司（2010）ベイジアンネットワークを用いたソフトウェア実装技術の選択支援, 情報処理学会論文誌 51(9), 1765-1776