プロダクトライン開発における
組込み機器のテスト設計・実行の自動化

――仕様書の論理記述変換によるテストケース自動生成と
モデル検査技術を用いたテストケース自動実行環境の構築――

青山裕介

目次

図目次

表目次

第 1 章　序論

1.1　研究背景

近年、組込み機器の開発では、グローバルな製品展開が行われており、日本では、2018 年時点で品目別輸出額のおよそ 6 割を占める組込み機器が海外に展開されている [1]。グローバルに展開される機器は、販売先地域の要求に合わせてカスタマイズされる。加えて、市場の要求の変化へ迅速に対応していくために、これらカスタマイズを行った製品は短期間に繰り返し開発される。

こうした多品種の製品を短納期で開発するための手法として、組込み機器開発の現場ではプロダクトライン開発 [2] が導入されている。プロダクトライン開発では、ソフトウェア製品を Domain Artifact と Application Artifact という 2 種類の部品に分け、両部品の組み合わせにより製品を実現する。Domain Artifact は、製品シリーズに共通の機能を提供するコアの部品であり、この部品を繰り返し再利用することで開発工数を削減する。Application Artifact は、製品ごとの特徴となる機能を提供する部品であり、搭載する Application Artifact を変えることで、多品種製品を実現する。

上述のような部品の柔軟な組み合わせや再利用による開発の効率化は、V 字モデルにおける設計・実装工程に留まり、テスト工程は未着手であった。このため、テストケースの設計と実行に関してそれぞれ課題があった。

テストケース設計における課題は、テストケースのレビューで誤りを指摘し、改善を促すことが難しいこと、テストケース設計に工数が掛かることである。

日本を代表する企業 2 社の技術者複数名へのヒアリングと観察の結果によると、組込み機器開発の現場では、テスト設計者が自然言語で記載された仕様書を精読し、各文からテストにおいて機能に入力するための動作条件・確認する期待動作を抽出することによってテストケースが設計されている。また、自然言語で記載された仕様書にはしばしば記述漏れや曖昧な表現と言った欠陥を含むため、テスト設計者は、上記作業の遂行中に、システム仕様書の欠陥の修正・補完も併せて行っている。設計されたテストケースは、欠陥の修正・補完内容やテストケースへの変換の誤りを除去するために、システム設計者・テスト設計者を交えてレビューされる。一方で、上記テストケースへの変換作業、及び欠陥の修正・補完作業は各テスト設計者に閉じた暗黙的な手順で実施されるため、テスト設計者がどのような根拠でテストケースを出力したのかがわからず、これがテストケースのレビューによる誤りの修正を困難にしていた。

また、テストケースのレビューでは、テスト設計者が実施した仕様書の欠陥の修正・補完結果も合わせて検証されるため、このレビューで修正・補完の誤りや仕様書の欠陥が発見された場合、テストケース設計をやり直す必要があり、この手戻りがテストケース設計の工数を大きくしていた。この結果、ある新規製品の 520 ページ程度の仕様書を用いたテストケース設計にはおよそ 700 時間、

派生製品の 640 ページ程度の仕様書を用いたテストケース設計に 255 時間もの工数を要していた。

テストケース設計実行における課題は、テストケース実行に工数がかかることである。

近年では、IoT をはじめとした機器間のコミュニケーションを実現するために、ネットワークを介した通信機能を備える製品が開発されている。組込み機器が備える機能は、ネットワークを介して受信したメッセージによって非決定的に起動されるため、機能の実行順序に起因する不具合を除去するために、機能を実行する順序を入れ替えたテストが必要となる。起こりうる実行順序の数は、組み合わせる機能の数によって指数関数的に増加する。

テスト工程では、新しい機能の組み合わせの製品が開発される度に、複数の機能を組み合わせ、かつ複数通りの順序で機能を実行するテストケースを設計し直し、テストを実行している。プロダクトライン開発では、個々の製品に比べて規模がより大きく、複雑となる [3] ため、上述のような派生製品開発の度に要するテストケースの再設計・実行工数が、多品種・短納期の開発の障害となっていた。共同研究先のあるビル用空調システムの例では、室外機一台あたりおよそ 30–50 機能ほどが動作している。これら複数機能の組み合わせで行うテストケースの設計には、およそ 500 時間もの工数を要していた。

以上のように、プロダクトライン開発を用いた組込み機器の開発には、テスト工程において、テストケースの設計と実行に課題が存在していた。

第 2 章　プロダクトライン開発を用いた組込み機器の既存テスト技術

本章では、1.1 節で述べたプロダクトライン開発におけるテストの課題について、適用可能な既存技術を調査した結果について述べる。

2.1　テストケース設計の支援技術

本研究では、テストケース設計に当たり、自然言語仕様書に含まれる欠陥が暗黙的なプロセスによって除去されることをテストケース設計における課題として取り上げる。本節では、これに関連する技術として、自然言語仕様書の欠陥修正手法、及び自然言語仕様書からのテストケース設計手法について調査した。以降、その結果について述べる。

自然言語で仕様を書くことによる欠陥の混入を防止するために、自然言語の曖昧さを排して仕様記述を実現する手法として、Z 記法 [4]、VDM++ [5]、B Method [6] といった仕様の形式記述手法がある。これらは厳密な仕様記述のための記法を提供し、記述した仕様に対する自動検証機能として、それぞれ [7]、 [8]、 [9] が提案されている。

[10] では、自然言語の記述ルールを制限したシナリオの記述手法が提案されている。記述ルールを制限することで、自然言語仕様書に書かれている表層的な表現から、そのシナリオ中で実施されるべきイベントに機械的に置き換えられる。

一方で、これら手法は、既存の自然言語仕様書に対して適用できず、予め仕様書を上記手法で提案する文法で記述するか、自然言語仕様書を書き直す必要がある。

[11] では、日本語文を論理式に変換する手法を提案している。自然言語で書かれた仕様書では、仕様書中に含まれる各入力項目や確認項目間の論理関係を記述できないため、論理式として表現することで、論理関係の曖昧さの解消が期待できる。一方 [11] では、述語「Action する」の知識表現のための論理式変換手法として提案されているため、例えば「Condition 中に Action する」のような表現においては、「Action する」は「Condition 中に」を項として持つ、という知識を表現するために、「Condition 中に」と「Action する」が一つに併合される。テストでは、「Condition」を機能の入力項目、「Action する」を機能の確認項目として別々に扱い、入力項目を機能に与えながら、確認項目を確認する作業を行う。 [11] で変換された論理式をテストケース設計で応用する場合、変換された論理式から「入力項目」、「確認項目」を改めて取り出す必要となる。

自然言語の仕様書自体から欠陥修正を実現する手法としては、 [12–15] が提案されている。

[12] は、複数通りに解釈できるフレーズを曖昧さと定義し、この曖昧さの検出を行う手法を提案している。 [12] の手法では、曖昧な用語と共起、及び構文構造のパターンを、それぞれキーワード検索や正規表現での検索、及び Part-of-Speech (POS) tag に対する正規表現での検索によって曖昧さを検出する。

[13] は係り受けと照応の曖昧さを検出する手法を提案している。 [13] では、複数のヒューリスティクスによって曖昧さをスコア付けしたベクトルを入力に、機械学習によって与えられた語の曖昧さを判断する。[13] の手法では、教師データに複数人の確からしさについての判断を用い、確からしさがしきい値を下回るものを曖昧と判断する。しきい値の制御によって曖昧さの検出の厳密さを制御することが出来る。

[14] は、Requirement Engineering の文脈で問題となる曖昧さをカテゴリ分けし、カテゴリに応じた曖昧さを検出する手法を提案している。 [14] では、カテゴリごとの checklist と scenario-based reading によって曖昧さ検出を行う。カテゴリごとに検討することで、検討漏れの防止に寄与する。

[15] は、自然言語で書かれた仕様から、曖昧さを除去するための修正のガイドラインを提示する。[15] では、自然言語の仕様中の各操作を、格文法を元に、主体語・対象語などを明らかにした形式的な内部表現へ変換する。変換された結果に対し、語や action の不足といった欠陥の検出を行うルールを適用し、検出結果に基づいてユーザに修正を促す。修正・補完された中間表現を自然言語の表現に戻し、曖昧さの除去された自然言語の仕様記述を得る。

自然言語仕様書からテストケースを生成する手法としては、 [16–18] が提案されている。

[16] は、本論文での提案手法と同じく、自然言語仕様書からデシジョンテーブル形式のテストケース生成を行う。[16] では、日本語の仕様文から句を抽出し、句がデシジョンテーブル中の条件に当たるのか、動作に当たるのかを自動的に判定する。一方で、句の間に存在する論理関係の同定や、句の真理値割当を求める手法については提案されていない。

[17] では、自然言語文の文法に制限を加えた Controlled Natural Language (CNL) という仕様記述文から、テストケース生成ツール T-VEC Tabular Modeler (TTM) への入力となる Software Cost Reduction (SCR) のコードを生成する手法を提案している。 [17] では、文法を制限することで、文中の条件部分、アクション部分を機械的に抽出可能にしている。また、格文法に基づいて、動詞とその動詞の項の意味役割、及びその動詞と項の記述文法も定義されており、CNL をパースの結果では、動詞の各項が持つ意味役割も機械的に解釈している。CNL のパース結果は自動的に SCR へ変換され TTM によるテストコードの自動生成を実現している。

[17] の手法では、文法を制限しつつも自然言語の表現を用いることで、形式手法について熟練せずに、曖昧さを排除した仕様記述を実現している。一方で、 [17] で提案する仕様記述手法は、自然言語表現ではあるものの、厳格な文法に基づいて仕様記述を行う必要があるため、既存の製品の仕様書には直接適用できない。

[18] では、ユースケース記述における各文の actor と action について意味ラベルを付与し、Use Case Description (UCD) モデルとして構築する手法を提案している。プロジェクトごとに専用の用語を用いられうる actor の表現について、意味ラベルを付与することで、ユーザなのかシステムを可能にしている。 [18] では、actor に付与された意味ラベルを元に、ユーザが actor の文を実行ステップ、システムが actor の文を評価ステップとするテストケースのテンプレートへの応用が提案されている。

[18] は、ユースケース記述を対象としている。ユースケース記述は、ある機能の取りうる入力のひとつを一ステップずつ列挙していく文書である。一方、本研究で対象とするシステム仕様書では、

複数の入力項目や確認項目が一文に統合された形式の文書である。このため、一文から一つの入力・出力が書かれるということを仮定できず、一文から複数の入力項目の確認項目の抽出、及びその項目間の論理関係の同定といった操作が必要となる。

2.2 テストケース実行の支援技術

本節では、プロダクトライン開発において膨大になる複数の機能を決定的・非決定的に組み合わせたテスト実行を支援する技術について調査した結果を述べる。

[19, 20] は、プロダクトライン開発を適用した製品群全体を評価するモデル検査手法を提案している。 [19, 20] では検査対象のモデルに差分機能の有無を組み込んでおり、起こりうる全ての製品群で所定の評価項目を満たすかを検査する。 [19, 20] は製品の仕様を対象にテストを実施する。

[21] は、Java で書かれたプログラムについて、プロダクトライン開発に搭載される feature に対応する変数の有効・無効を切り替え、全プロダクトファミリーに対応するソフトウェアを生成し、検証する手法を提案している。ソフトウェアが満たすべき仕様は、Java Modeling Language (JML) を用いて、Java プログラム中にコメントとして記載される。JML では、不変性やメソッドの事前条件・事後条件が記載でき、生成されたソフトウェアについて、JML で書かれた仕様を満たすか定理証明器を使って検証する。

[22] は、プロダクトラインにおける feature の組み合わせによる不具合を見つける技術として、プロダクトラインに含まれる全 feature を備えるひとつの製品を生成し、また、同時に feature の有効・無効をグローバルな boolean 変数により切り替え可能にする。生成された製品について、モデル検査器により、feature の有効・無効を非決定的に選択し、起こりうる全製品上で safety property の検証を行う。

非決定的なテスト手順を自動的に網羅することのできるモデル検査の技術を用いて実行テストを行う技術としては、 [23–26] が提案されている。

[23, 24, 26] は、ソフトウェアに対する入出力の関係（本統合テスト環境におけるブラックボックステスト）によってテスト結果を評価する。一方、ソフトウェアに実装されるメソッドから必ずしも評価に十分なだけの情報を取得できないことがあり、例えばソフトウェアの内部変数についての評価は不可能なことがある。

[25] は C/C++のソフトウェアをモデル検査技術によってテストする手法が提案されている。共有メモリ上の値を状態として保存、復元して探索する。イベントドリブンのシステムに適用し、イベントハンドラを非決定的な順序で呼び出してテストを実施する。テスト結果の成否は、ソースコード中に埋め込んだアサーションや Boolean 関数の戻り値によって判定される。アサーションの埋め込みのためにソースコードを改変する必要があり、製品ソフトウェア自体をテストできない。

モデル検査技術とシミュレーション技術の融合により組込みシステムの制御ソフトウェアの検査を行う技術として [27] がある。 [27] は、組込みシステムの制御ソフトウェアのモデル（コントローラモデル）の検査のために、SMT ソルバ Yices と Simulink によるシミュレーションを組み合わせて有界モデル検査を行う。 [27] では、Simulink 上で制御対象（プラント）の振る舞いをシミュ

レーションし、シミュレーション結果として得たプラントのセンサ値をコントローラのモデルに与える。コントローラモデルは、SysML から SMT ソルバ Yices 記述に変換されたものを用い、プラントのセンサ値をもとに順次 Yices 上のセンサ値（コントローラの入力値）を更新しながら設計制約の検証とプラントに対する制御量を算出する。算出された制御量をもとにプラントのシミュレーションを行い、再度コントローラモデルで検証する、ということを繰り返すことで、コントローラモデルの検証を行う手法である。

[27] は検設計段階のコントローラのモデルを検証対象とするため、 [27] では、コントローラの動作モデル自体を実行・検証するが、組込みソフトウェアはテストしない。

非決定的なテスト手順を網羅可能なモデル検査技術を応用し、複数の機能を組み合わせたテストケースの設計を支援する手法も提案されている。 [28–31] では、モデル検査器によってテストケースを生成する方法を提案している。これらは、モデル検査器にテスト失敗となるようなモデルを与えて、テスト失敗時に得られる反例に含まれるテスト実行系列を使って、実装済みソフトウェアのテストケースを生成する。

[32] は、モデル駆動開発で用いられる Simulink ツールで設計されたモデルを用いたテスト技法として、Simulink のモデルからテストケースの生成・優先度付けを行う技術を提案している。signal feature という特徴的な信号の形状（定数値を取るもの、一次導関数の符号が一定のもの、第二次導関数の符号が一定など）を定義している。signal feature とのユークリッド距離による類似度によって、出力信号を特徴づけした feature vector というものを定義し、feature vector 同士のユークリッド距離 diversity がより大きくなる入力信号の集合を生成することで、多様な信号をテストするテストケースの生成を行っている。

テストケースの優先度付けでは、テストをされていない Simulink モデル上のノードをよりカバーし、選択済みのテストケースに含まれる出力信号との diversity がより大きくなる出力信号を含むテストケースを優先的に選択する。これにより、より多くのノードをカバーし、より他のテストケースではカバーできないテストケースを優先的に選択可能にしている。

以降では、決定・非決定的なテスト実行を行う対象機能を削減することで、テスト実行にかかる工数を削減する手法について述べる。

[33, 34] は、プロダクトライン開発における派生製品のモデル同士の差分を元に、モデルの差分がテストに影響を与えないものを除外し、より少ない数のテストケースを選択する手法を提案している。 [33, 34] では、model slicing という手法を用いることで、派生製品同士のモデルの差分がモデル中の特定の要素（状態遷移等）に至る状態や状態遷移を求め、この状態・状態遷移の差分からテストケースに影響を与えるかを決定し、テストケースの選択を行っている。[34] では、プロダクトのファミリー全体の更新についても考慮した、テストケースの選択手法を提案している。

[35] は、プロダクトライン開発において、control-flow と data-flow の解析を行い、特定のテスト結果に影響を与える feature の組み合わせを求めることで、全ての feature の組み合わせに全てのテストを実行することに比べ、テストを削減する手法を提案している。 [35] の手法では、特定のテストで実行されるコードのうち、ある feature の有効・無効を切り替えることで、別の feature の control-flow や data-flow が変換する関係にある feature を関連する feature として取り出す。この関

連する feature についてのみ、feature の有効・無効の組み合わせをテストすることで、実行されるテストを削減する手法である。feature 間の関連に関する不具合については、 [35] の提案するようなホワイトボックスのアプローチと、仕様書に基づいたブラックボックスのアプローチの両面がある。ソースコードを元にしたホワイトボックスのアプローチでは、ソースコード上で意図せず起きてしまった関連を発見できる一方、誤って feature 同士に関連がなくなってしまっているなど、ソースコード上の問題によって、feature 間の関連が見逃されることがある。

2.3 本研究で取り組む課題

テストケース設計では、最終成果物としてテストケースを得ることが目的であるが、従来では、自然言語仕様書に含まれる欠陥除去に取り組む手法 [12–15] はあるものの、テストケース設計までをスコープに含めていなかった。あるいは、自然言語仕様書からのテストケース設計は行うものの、[16] のように、仕様書に含まれる欠陥にフォーカスしていなかった。 [4–9, 17] では、専用の記述言語で書かれた仕様書を対象にしてテストケースを生成する（ [4–9]）あるいは検証する（ [17]）手法のため、既存の自然言語の仕様書を用いたテストケース設計には適用できなかった。

また、テストの実行においては、並列に動作しうる機能のテストを行う手法として、仕様のモデルを対象とする手法 [27] や、ブラックボックスな方法でソフトウェアの行う手法 [23, 24, 26]、製品のソースコードを改変し、ホワイトボックスな方法でソフトウェアのテストを行う手法 [25] が提案されている。しかし、組込み機器では、組込み機器のソフトウェアが外部に公開しているメソッドからは評価に十分な情報——例えば組込みソフトウェアの内部変数——を得ることができないことがあり、ブラックボックスな手法のみではテストが困難なことがあった。

以上を踏まえ、本研究では次の課題にフォーカスし、これら課題の解決に取り組む。

1. 自然言語で書かれ、かつ欠陥を含む既存のシステム仕様書を用いたテストケース設計を対象に、テスト設計者が暗黙的なプロセスで欠陥の修正・テストケースへの変換を行うため、結果修正の誤りがあった場合に、手戻りが大きくなること。
2. 複数機能を衝突させたテストを派生製品の開発の中で繰り返し行うに当たり、テストケースの設計・実行に工数がかかること。本研究では組込み機器のソフトウェア内部の状態変数を評価するホワイトボックスなテスト、及びハードウェアを用いたブラックボックスなテストの双方に掛かる実行工数の課題を対象とする。

2.4 本論文の構成

第 3 章–第 5 章では、図 2.1 のような関係にあり、本研究で提案するアプローチをそれぞれ述べる。第 3 章では、本研究で提案するテストケースの設計プロセスとその支援ツールについて述べ、第 4 章では、部品化した機能ごとのテストケースを組み合わせて決定・非決定的なテストケースの生成を実現するテストケースリポジトリ、及びテストケースリポジトリから生成されたテストケー

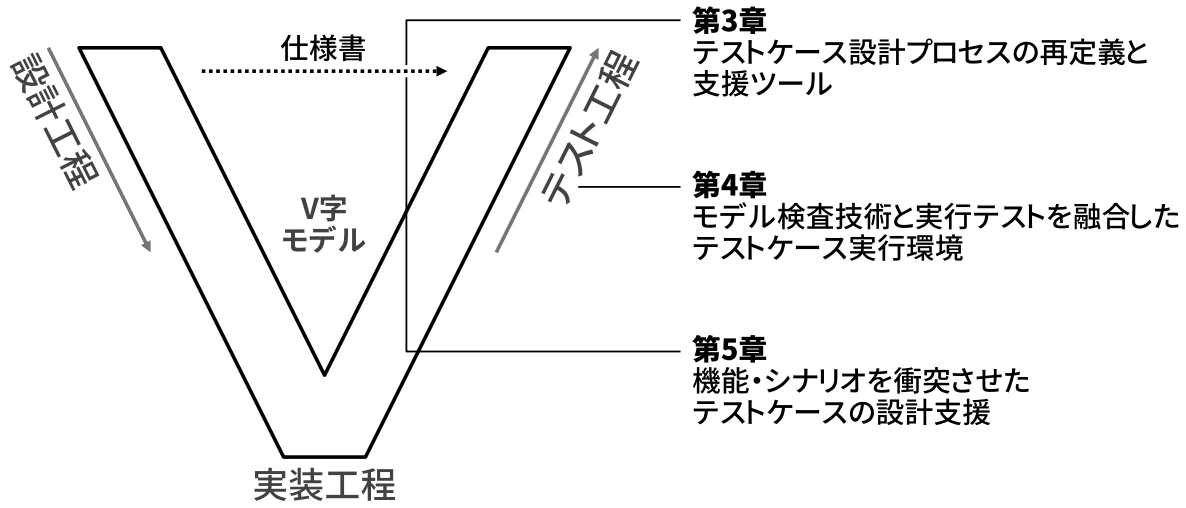


図 2.1　提案手法の全体像

スを用いて、決定・非決定的なテストを自動評価するテスト環境について述べる。並列動作する機能の取りうる実行順序は、機能の数が増加するにつれて指数関数的に増加する。構造的に並列に動作しうる全ての機能を網羅的に実行することは工期の都合上困難となる。第 5 章は、並列に動作する機能のうち、影響関係にある機能の選択支援を支援することで、起こりうる実行順序を削減する手法を紹介する。第 6 章にて本研究をまとめる。

第 3 章　テストケース設計プロセスの再定義と支援ツール

本章では、自然言語で書かれたシステム仕様書から統合テスト環境のテストケースリポジトリの入力となるテストケースを設計するためプロセスとその支援ツールについて述べる。

システムテストのテストケース設計は、略言すると、自然言語で書かれたシステム仕様書からテストに用いる項目を抜き出して、テストケースの形式へ変換する作業である。ここで、自然言語で書かれたシステム仕様書は、曖昧な表現や記述漏れといった欠陥を含みうるので、この変換作業にはしばしばこれら欠陥の修正作業が伴う。従来では、この欠陥の修正・テストケースへの変換作業は各テスト設計者の頭の中で暗黙的なプロセスとして手作業で実施されており、属人性が高い作業であった。また、システム仕様書の記述内容を見落としなくテストケースに反映するために、テスト設計者は、システム仕様書を一文一文精読しながらテストケースを設計する。このため、システム仕様書から入力項目・確認項目に当たる文を抜き出す作業だけでも、新規製品の 520 ページ程度の仕様書を用いた全 700 時間テストケース設計のおよそ 26%、派生製品の 640 ページ程度の仕様書を用いた全 255 時間テストケース設計のおよそ 31%もの多くの工数を要していた。

設計されたテストケースは、欠陥の修正やテストケースへの変換結果の誤りを除去するために、システム設計者・テスト設計者を交えてレビューが実施される。一方で、テストケース中には、入力されたシステム仕様書を元に、テスト設計者がどのような根拠でそのテストケースを出力したのかが現れないため、レビューによる誤りの修正が難しかった。

本研究では、以降の節にてそれぞれ詳述する次の項目により上記課題の解決を図った。

1. 複数企業へのヒアリング結果に基づき、システム仕様書からテストケース設計プロセスを再定義し、プロセス中のステップをアルゴリズム化してテストケース設計の属人性を削減
2. システム仕様書からテストの入力項目・確認項目の抽出と言った、定型的な作業の自動化してテストケース設計工数を削減
3. テストケースへの変換の中間成果物を用いてシステム仕様を可視化し、システム仕様書の欠陥のレビューを容易化

3.1　現場で実施されているテストケース設計のヒアリングと課題

テストケース設計プロセスを再定義するに当たり、テストケース設計プロセスに含めるべきステップや自動化可能なステップ、及び修正される欠陥の種類について明らかにするために、テレビや空調機の組込みシステムの開発を行う企業 2 社にテストケース設計業務についてヒアリングと観察を行い、従来のテストケース設計プロセスをモデル化した。この結果、どちらの企業も次のような手順でテストケース設計を行っていることがわかった。

1. **テストケースの大項目の抽出:** システム仕様書の章立てからテストケースの大項目を抽出する。
2. **機能の条件文・動作文の抽出:** システム仕様書をラインマーカでマーキングしながら、一文一文精読し、「A **ならば** B する」のような関係にある「ならば」の前件部分（条件文）と後件部分（動作文）を手作業で抜き出す。
3. **テストケースの小項目の設定:** 機能の入力項目をテスト入力とし、動作文を確認項目としてテストケースの小項目を設定する。
4. **テストケースの小項目の具体化:** テスト条件に合致する入力値と、対応する確認項目を求め、テストケースの小項目を具体化する。
5. **テストケースの中項目の作成:** テスト手順を簡便化するために、テストケースの小項目まとめ、テストケースの中項目を設定する。

上記作業中に発見したシステム仕様書の欠陥（記述漏れ・論理誤り）は、システム仕様書の作成者に問い合わせを行って逐次修正される。テスト設計者が自明と判断するものについては問い合わせを行わずに、修正結果が直接テストケースに反映される。

3.2 提案するテストケース設計プロセスの概要

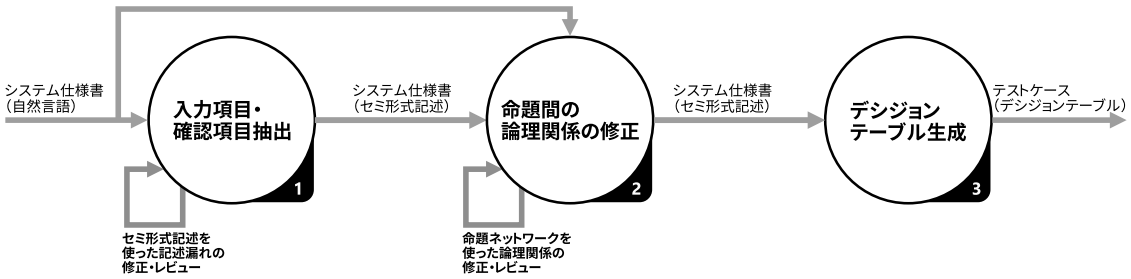


図 3.1 提案するテストケース設計プロセス

本研究では、3.1 節のテストケース設計手順を踏まえ、図 3.1 のようなテストケース設計プロセスを定義した [36, 37]。テストケース設計プロセスでは、システム仕様書からテストケースへの変換作業の属人性の排除と、工数の削減、及び人為的な変換ミスを防止するために、図 3.1 のステップ 1–3 によって、システム仕様書からテストケースへの定型的な変換作業を自動化する。

自然言語で記述されるシステム仕様書には欠陥が含まれることがあるため、これら欠陥はシステム設計者にフィードバックし是正される必要がある。この修正のために、本テストケース設計プロセスでは、テストケースへの変換の中間成果物に対して、システム設計者・テスト設計者を交えてレビューを実施する。仕様書の欠陥をテストケースの前にレビューすることで、テストケースへ混入したシステム仕様書の欠陥に伴う手戻りの削減が期待できる。また、テストケースへの変換ステップがアルゴリズム化されたことで、従来のテスト設計者毎に暗黙的に実施されていたシステム

仕様書の欠陥修正・テストケースへの変換プロセスに比べ、変換誤りの所在を明らかにして指摘・改善が可能となる。本テストケース設計プロセスでは、システム仕様のレビューを容易化するために、修正すべき欠陥が明らかになる形式でシステム仕様を可視化する。

以降の節では、提案するテストケース設計プロセスの各ステップについて述べる。

3.3 自然言語仕様書のセミ形式記述化

3.3.1 セミ形式記述の記述方法

```
/* Written in Extended Backus-Naur Format (EBNF) */
statement = expression, "." ;
expression = "(", expression, ")",
   [ "<", p_constraint, { ",", p_constraint }, ">" ] |
   expression, "&", expression |   /* and */
   expression, "|", expression |   /* or */
   expression, "->", expression | /* imply */
   "!", expression |              /* not */
   clause ;
/* the operator precedence is "!" > "&" > "|" > "->" */
clause = verb, "(", subject, ",", object,
   { ",", verb_constraint }, ")",
   [ "<", p_constraint, { ",", p_constraint }, ">" ] ;
p_constraint =
   [ "!" ], "O", "<", group_name, ">" /* One */ |
   [ "!" ], "E", "<", group_name, ">" /* Exclusive */ |
   [ "!" ], "I", "<", group_name, ">" /* Inclusive */ |
   [ "!" ], "R", "<", direction, ",",
     group_name, ">"                  /* Require */ ;
direction = "s" | "d"; /* Source or Destination */
group_name = ident;
verb = ident;
subject = ident;
object = ident;
verb_constraint = ident;
ident = /* Japanese and English Letters */;
```

図 3.2 セミ形式記述の文法

組込み機器開発を行う企業 2 社へのヒアリングと観察の結果から、システムテストのテストケース設計では、まず、システム仕様書中の記述から機能の入力項目・確認項目を抜き出す作業が実施されることがわかった。本小節では、この作業をアルゴリズムとして明確化・自動化することで、変換誤りの所在を明らかにしたレビューと工数の削減を実現するための、自然言語の仕様文の論理記述変換手法について述べる。

自然言語で書かれたシステム仕様書は、論理関係を複数通りに解釈可能という曖昧さを含むことがあり、この結果、入力項目・確認項目間の誤った論理関係解釈のもとで、誤ったテストケースを設計してしまうことがある。また、日本語で書かれた仕様書では、しばしば主語や対象が省略されることがあるが、これら省略語は、日本語仕様文を読む際に、無意識に補って読んでしまうため、欠落している語の発見が難しく、誤った補完内容がテストケースの誤りを引き起こすことがある。

このため、本研究では、後述のアルゴリズム（3.3.2 小節）によりシステム仕様書から入力項目・確認項目を抽出し、かつ抽出した項目を論理記述することで、抽出工数の削減と、レビューによる

欠落語の発見の容易化、論理関係の解釈誤りの防止を図る。

本研究で提案するシステム仕様の論理記述は、形式的な仕様記述手法 [15] をテスト設計への活用のために本研究において拡張したものであり、セミ形式記述と呼ぶ、図 3.2 に定義する文法を持つものである。セミ形式記述では、欠落語の発見の容易化と、論理関係の明確な表現を目的に、1 つ 1 つの入力項目･確認項目を命題プリミティブと呼ぶ「関係語（主体語，対象語，制約語）」という形式で表現する（図 3.2 の clause に相当する）。関係語・主体語・対象語・制約語を記述する順序を決定的にすることで、主体語や対象語が省略や関係語が自動詞のために対象語を持たないなど、主体語・対象語に該当する語がシステム仕様書に記述されていない箇所を明らかにできる。システム仕様書中に記載のない語は記号にて代替表記を行い、省略により主体語や対象語が不明な場合には「?」を用いて代替表記し、対象語を持たない場合には「_」を用いて代替表記する。この代替表記の記号「?」により、欠落語を含む命題プリミティブを可視化する。

関係語には静的な状態を表現するためのものに「is」を、アクションを表現するためのものに動詞の終止形を用いて、主体語・対象語・制約語の間の関係を表現する。「is」は、例えば図 3.3 の「アイドル中に」のように用言を持たず、静的な状態（付帯条件）を表す句に用い、「is（?，_，アイドル中）」のように記述する。また、「A は B だ」、「A は B である」といった、A と B の間の静的な同値関係を表す「……だ」、「……である」といった判定詞についても、「is」を用いて「is（A，_，B）」のように記述する。

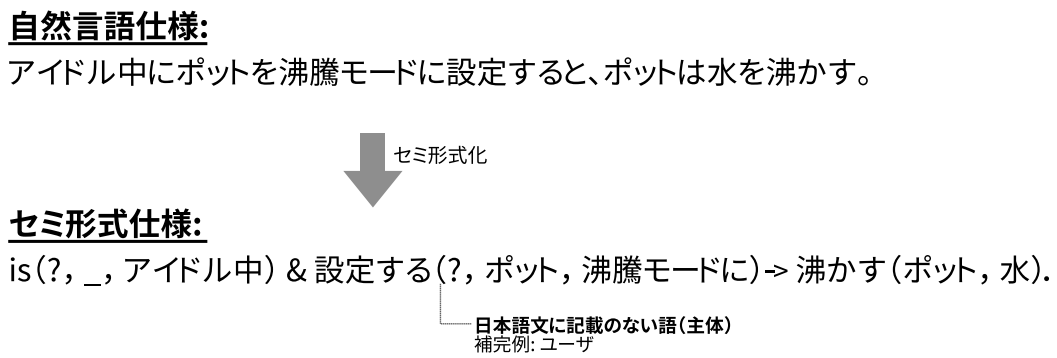
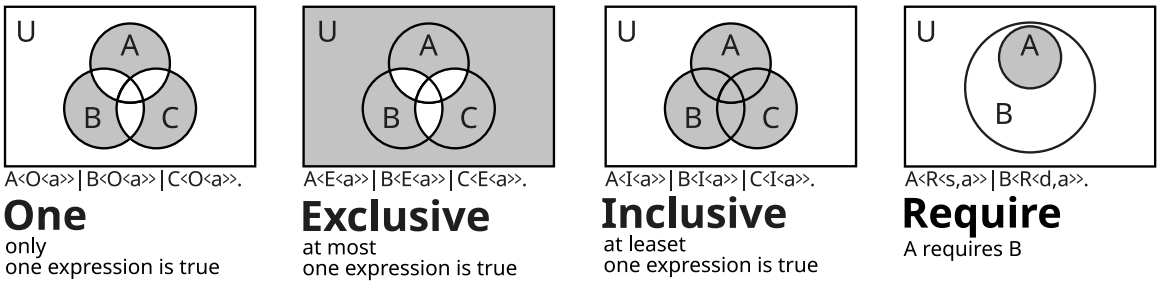


図 3.3　セミ形式記述の例

テストに際しては、関係語・主体語・対象語の 3 項目により各入力項目・確認項目で行われる操作やシステムの状態を表し、続く制約語により入力項目・確認項目のパラメータを列挙する。

命題プリミティブ間の論理関係については、元の自然言語の仕様書にあるような解釈の曖昧さを排除するために、命題論理の 4 つの論理演算子―― Not（記号: 「!」）、And（記号: 「&」）、Or（記号: 「|」）、Imply（記号: 「–>」）―― を用いて厳密に記述する。加えて、排他関係といった上記論理演算子での記述の難しい命題[1)]間の制約を表現するために、原因結果グラフ [38] から図 3.4 に示す 4 つの命題間制約――One（ただ 1 つの命題が成り立つ）、Exclusive（多くとも 1 つの命題が成り

1)　命題プリミティブ、及び命題プリミティブに Not をつけたもの、And・Or・Imply で結合したものを総称してここでは命題と呼ぶ。



※網掛け部分にて命題が真になる

図 3.4　セミ形式記述の命題間制約（p_constraint）

立つ)、Inclusive（少なくとも 1 つの命題が成り立つ)、Require（ある命題 A の充足には命題 B の充足が必要である）——を採用し、図 3.2 の p_constraint の記法で表現する。

入力項目と確認項目の間の論理関係は、自然言語においては「入力項目が成立する**ならば**、確認項目をもたらす」というように、「ならば（Imply)」に対する前件・後件として記述されることが技術者へのヒアリングと観察の結果判明したため、セミ形式記述上でも論理演算子 Imply（「–>」）を使い、図 3.3 のように「入力項目 –> 確認項目」と表現する。

図 3.1 1 のレビューでは、「?」を持つ命題プリミティブを修正することで、欠落語を含む命題プリミティブを修正する。セミ形式記述上では、元の仕様書に記述されていない欠落語は「?」として明示されるので、「?」に当てはまる語をレビューで埋めることで欠落語を補完できる。

3.3.2　セミ形式記述への変換アルゴリズム

本小節では、システム仕様書からテストの入力項目・確認項目を抽出作業の容易化のために、システム仕様書の文章をセミ形式記述へ変換するアルゴリズム [39] について述べる。

セミ形式記述への変換アルゴリズムの構築にあたっては、セミ形式記述が本研究で新規に提案する手法であり、機械学習によるアプローチに必要な教師データ（対訳データ）を入手できないため、本研究では、ルールベースのアプローチを採用した。ルールの構築には形態素解析器 Juman++ [40] （v2.0.0-rc2[2)]）及び、日本語構文解析器 KNP [41] （v4.1.9[3)]）から得られる feature [42] のデータ（格 [43]、品詞、係り受けなどを含む文法データ）と並列な係り受け構造の検出結果を用いる。本論文で提案する日本語の仕様文からセミ形式記述への変換アルゴリズムの概要は図 3.5 である。

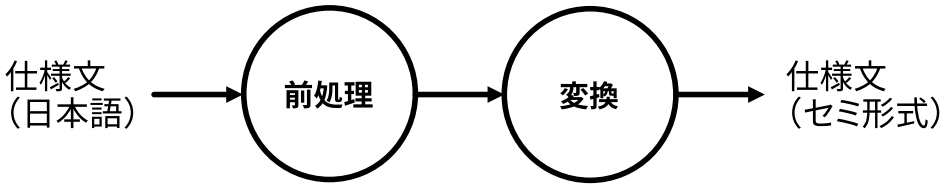


図 3.5　セミ形式記述への変換プロセス

2）https://github.com/ku-nlp/jumanpp/releases
3）http://nlp.ist.i.kyoto-u.ac.jp/index.php?KNP

図 3.5 の前処理ステップは、形態素解析の正確さの向上と変換ルールの簡単化のために次のような処理によって、入力された日本語仕様書の記述を統一する。

1. Juman++で未定義語となる文字種の置換（例: 半角カナから全角カナへ置換）
2. 文の終了スタイルの統一（「……すること。」のように「こと」で文を終えるスタイルを「……する。」のように用言で文を終えるスタイルに統一）

Algorithm 1 日本語文をセミ形式記述へ変換するアルゴリズム

Definition:
- children(b): 文節 b にかかる子文節のリストを返す。
- push(l, e): リスト l に要素 e を追加する。
- pop(l): リスト l から末尾の要素 e を取り除き、e を返す。
- last(l): リスト l の末尾の要素を返す。
- reversed(l): リスト l を逆順にして返す。
- applyF(op, s_t, s_f): オペレータ op の $func_f$ を s_t、s_f を引数に呼び出す。
- applyS(op, s_t, s_f): オペレータ op の $func_s$ を s_t、s_f を引数に呼び出す。
- NewSemiElement(b): 文節 b を句・命題へ変換した結果を返す。
- NewOperator(s_t, s_f): セミ形式記述と文節のペア $s_t = (d_t, B_t)$ と $s_f = (d_f, B_f)$ を結合するためのオペレータを返す。（なお、d_t と d_f は句・命題、またはセミ形式記述へ変換過程のデータ構造であり、d_f が d_t に係る。B_t と B_f はそれぞれ d_t と d_f の変換元となった文節のリストである）。

Input:
b: 文末文節

Output:
d: 命題（命題プリミティブまたは二項演算オブジェクト）

```
 1: procedure Convert(b)
 2:     stack ← []
 3:     C ← children(b)
 4:     s_t ← (NewSemiElement(b), [b])                    ▷ 文節のセミ形式記述変換ルール
 5:     s_f ← nil
 6:     while true do
 7:         if s_f = nil then
 8:             if |C| = 0 then
 9:                 break
10:             c ← pop(C)
11:             s_f ← (Convert(c), [c])
12:         op_{s_f} ← NewOperator(s_t, s_f)              ▷ セミ形式記述結合ルール
13:         if |stack| = 0 ∨ op_{s_f} is prior to last(stack) then
14:             push(stack, (op_{s_f}, s_f))
15:             s_f ← nil
16:             continue
17:         (op_last, s_last) ← last(stack)
18:         if applyS(op_last, s_f, s_last) ≠ nil then
19:             s_f ← applyS(op_last, s_f, s_last)        ▷ Conversion 1
20:             continue
21:         s_t ← applyF(op_last, s_t, s_last)            ▷ Conversion 2
22:         s_f ← nil
23:     for all (op, s) in reversed(stack) do
24:         s_t ← applyF(op, s_t, s)                      ▷ Conversion 3
25:     return d_t of s_t where s_t = (d_t, B_t)
```

図 3.5 の変換ステップでは、変換ルールを元に Algorithm 1 によって日本語の仕様文をセミ形式記述の仕様文へと変換する。Algorithm 1 では、大きくは次の流れでセミ形式記述への変換を行う。

1. 各文節を再帰的にセミ形式記述に変換する（Algorithm 1 の line.4）、
2. 子文節[4]から変換されたセミ形式記述と親文節から変換されたセミ形式記述を再帰的に結合する（Algorithm 1 の Conversion 2–3）

4) 本論文では、文節 c が文節 p に係るとき、p から見た c を子文節、c から見た p を親文節、同じ親文節を持つ子文節同士を兄弟文節と呼ぶ。

なお、子文節と親文節のセミ形式記述の結合に際しては、子文節の兄弟文節が And や Or などの論理関係にある場合には、左記論理関係を維持してセミ形式記述変換するために、兄弟文節から変換されたセミ形式記述同士を先に結合（Algorithm 1 の Conversion 1）し、兄弟文節の結合結果を親文節から変換されたセミ形式記述と結合させる。

本変換アルゴリズムでは、セミ形式記述への変換ルールの蓄積を容易化するために、日本語の文節に対する直接の変換手続き（Algorithm 1 の Conversion 1–3）を部品化し、Juman++/KNP から得た構文解析結果のデータを元に変換手続きを定義する部品を選択する（Algorithm 1 の文節のセミ形式記述変換ルール、セミ形式記述結合ルール）構成をとった。これにより、変換手続きを直接記述することなく、ルールを蓄積していくことによって漸進的に変換の正確さを改善できる。

変換手続きの部品はオペレータと呼び、オペレータは 2 つのセミ形式記述を 1 つのセミ形式記述に結合する関数のペア $(\mathit{func}_s, \mathit{func}_f)$ として変換アルゴリズムに組み込まれている。func_s、func_f は次の記号を使って定義される関数 $\mathit{func}_s : s_t, s_f \mapsto s_r?$ と $\mathit{func}_s : s_t, s_f \mapsto s_r?$ である（「?」のついた変数「$x?$」は、x または nil の値を表すものとする）。

- $s_t = (d_t, B_t)$: 結合先のセミ形式記述と文節リストのペア
- $s_f = (d_f, B_f)$: 結合元のセミ形式記述と文節リストのペア
- $s_r? = (d_r, B_r)|\text{nil}$: d_t と d_f が結合可能なときは結合結果のセミ形式記述と文節リストのペア、結合不可のときは nil
- B_t、B_f、B_r: それぞれ s_t、s_f、s_r に対応する文節オブジェクトのリスト
  - 文節オブジェクト: KNP の構文解析結果のオブジェクト。日本語文中の文節のテキスト表現に加え、日本語文中で何番目に出現するか、係り受け構造が並列か否か、前述の feature データといった文法上のデータをプロパティとして有する。以降では文節オブジェクトのことを単に文節と呼ぶ。
- d_t, d_f, d_r: セミ形式記述のオブジェクト。句[5]、命題、または変換過程のデータ構造（句の二項関係ペア結合オペレータ等で用いる）に相当する。
  - 句: 文節か命題を要素として持つリスト
  - 命題: 次のいずれか
    - 命題プリミティブ $= (w_{\text{verb}}?, w_{\text{subj}}?, w_{\text{obj}}?, W_{\text{constraints}}, \mathit{neg})$: w_{verb}、w_{subj}、w_{obj} はそれぞれ関係語・主体語・対象語の句、$W_{\text{constraints}}$ は制約語を表す句のリスト、neg は w_{verb} が否定表現のときに true を取る真偽値。
    - 二項演算オブジェクト $= (\mathit{biop}, d_l, d_r)$: biop は二項演算子の種類（「and」、「or」、「imply」）、d_l、d_r はそれぞれ二項演算子の左、右オペランドに相当する命題。
  - （変換過程のデータ構造は、左記データ構造を用いるオペレータと合わせて説明する）。

変換アルゴリズムは、厳密には日本語文を上記セミ形式記述のオブジェクトへ変換する。上記オブジェクトは、次のように文字列化することで、図 3.2 の文法に沿ったテキスト表現と対応づくた

5） 命題プリミティブの各項（関係語・主体語・対象語・制約語）を本論文では句と呼ぶ。

め、以降ではセミ形式記述のオブジェクトのことも単にセミ形式記述と呼ぶ。

- 句:
  1. 句の中に含まれる全文節をリスト化し、日本語文中での出現順にソートする。
  2. 各文節を日本語文中の表記にしたがって文字列化し、文節同士を結合する。
- 命題プリミティブ:
  1. 句 $w_{\mathrm{subj}}?, w_{\mathrm{obj}}?$ が nil のときは「?」そうでなければ左記の句を文字列化したものを命題プリミティブの主体語・対象語に設定する。
  2. 句のリスト $W_{\mathrm{constraints}}$ 中の各要素を文字列化した後、各要素同士をカンマで結合したものを制約語に設定する。
  3. 句 $w_{\mathrm{verb}}?$ が nil のときは「is」を関係語に設定し、そうでなければ
     (a) w_{verb} の句の中に含まれる全文節をリスト化し、日本語文中での出現順にソートする。
     (b) 最後の文節以外の各文節を日本語文中の表記にしたがって文字列化し、最後の文節の用言を終止形にする。
     (c) 文節同士を結合して関係語に設定する。
  4. $neg = true$ のときは関係語の先頭に「!」を付ける。
- 二項演算オブジェクト:
  1. d_l、d_r を文字列化してそれぞれ二項演算子の左オペランド、右オペランドに設定する。
  2. d_l が二項演算オブジェクト、かつ d_l の二項演算子 $biop_l$ が $biop$ よりも低優先度のとき、左オペランドを丸括弧で囲む（ただし二項演算子の優先順位は「and」>「or」>「imply」）。
  3. d_r が二項演算オブジェクト、かつ d_r の二項演算子 $biop_r$ が $biop$ よりも低優先度のとき、右オペランドを丸括弧で囲む。
  4. $biop$ を次の対応で文字列化する（「and」：「&」、「or」：「|」、「imply」：「–>」）。
  5. 左右ｎｏオペランドの間に $biop$ の文字列を置く。

2 つのセミ形式記述を結合する関数 $func_s$、$func_f$ は、結合する各セミ形式記述を構成する文節の間にある、次の 2 種類の係り受け関係にしたがって、セミ形式記述同士を結合する。

1. 兄弟文節（$func_s$）：Algorithm 1 の Conversion 1 に相当する。図 3.6 の ① では、兄弟文節に当たる「沸騰モード中か」と「給湯 LOCK 中に」の間に Or の論理関係があり、この論理関係を反映した結合を実現する。
2. 親子文節（$func_f$）：Algorithm 1 の Conversion 2、3 に相当する。図 3.6 の ② では、「押したら」を命題プリミティブとした「押す（?, ?)」の対象語に、「給湯ボタンを」を結合している。

本アルゴリズムで定義したオペレータの一部を以降に挙げる。$func_s$、$func_f$ に相当する関数は、それぞれオペレータ名の末尾に「S」、「F」をつけた関数として表記する。任意の入力で nil を返す関数の定義は、紙面の節約のため省略する。結合の例では、表記を簡潔にするためにオペレータの引数のうち一部記号のみを記載する。例文中の「/」は、s_f、s_t に相当する部分を区切るための説明用の記号とし、「/」の前、後からそれぞれ s_f、s_t が構成されるものとする。

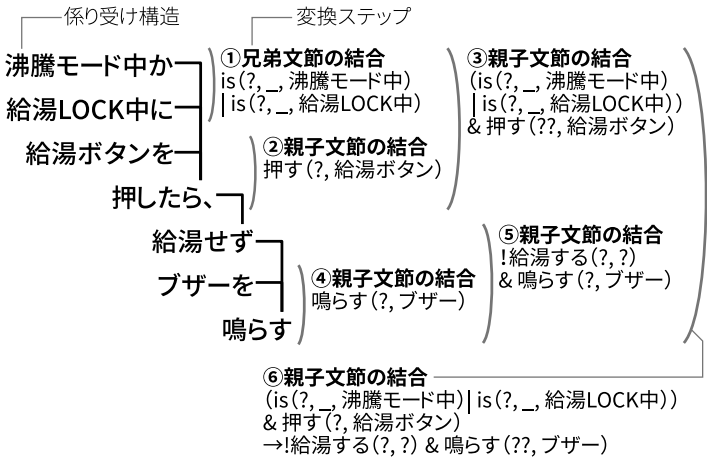


図 3.6　係り受けの構造の例と変換ステップ

句の結合オペレータ: 連体修飾による句の統合と、命題プリミティブの項への挿入を行う。

- アルゴリズム:

Definition:
- insertPhrase(d_t, d_f): 命題 d_t 中の命題プリミティブに句 $d_{c_{d_f}}$ を挿入した結果を返す。句の挿入位置（主体語・対象語・制約語）は、機械的な同定を行うために、[44] を元に、表層格を元に決定する。[44] では、主体を表す格（深層格「動作主」）はガ格が最も頻度が多く、対象を表す格（深層格「対象」）はヲ格が最も頻度が多いことが示されているため、ガ格の句を主体語に、ヲ格の句を対象語に、それ以外の句を制約語の位置に挿入する。判定の誤りはレビューにて修正する。

procedure 句の結合オペレータ F(s_t, s_f)
  if d_f が句でない **then**
    return nil
  if d_t が句 **then**
    $d \leftarrow$ 句 ($[d_f, d_t]$)　▷ 句のインスタンス
    return (d, $B_f \cup B_t$)
  if d_t が命題 **then**
    $d \leftarrow$ insertPhrase(d_t, d_f)
    return (d, $B_f \cup B_t$)
  return nil

- 例 1:「赤色の/LED を……」
  - 入力: d_f: 赤色の、d_t: LED
  - 出力: d_r: 赤色の LED
- 例 2:「再沸騰ボタンを/押す」
  - 入力: d_f: 再沸騰ボタンを、d_t: 押す（?，?）
  - 出力: d_r: 押す（?，再沸騰ボタン）

命題の結合オペレータ: 命題を二項演算子（And・Or・Imply）で結合する。

- アルゴリズム:

Definition:
- *biop*: 二項演算子（NewOperator で設定する）
- join(d_t, d_f, *biop*): (*biop*, d_f, d_t) なる二項演算オブジェクトを返す。

procedure 命題の結合オペレータ S(s_t, s_f)
  return 命題の結合オペレータ F(s_t, s_f)
procedure 命題の結合オペレータ F(s_t, s_f)
  if d_f が命題でない **then**
    return nil
  if d_t が句 **then**

```
 7:          d ← 句 ([d_f, d_t])                                  ▷ 句のインスタンス
 8:          return (d, B_f ∪ B_t)
 9:      if d_t が命題 then
10:          d ← join(d_t, d_f, biop)
11:          return (d, B_f ∪ B_t)
12:      return nil
```

- 例:「給湯ボタンを押すか/再沸騰ボタンを押すと、……」
  - 入力: d_f: 押す（?，給湯ボタン）、B_f: [給湯ボタンを、押すか]、d_t: 押す（?，再沸騰ボタン）、B_t: [再沸騰ボタンを、押すと]
  - 出力: d_r: 押す（?，給湯ボタン）| 押す（?，再沸騰ボタン）

句の二項関係ペア結合オペレータ: 句を二項演算子と共にペアにする。

- アルゴリズム:

```
Definition:
    • biop: 二項演算子（NewOperator で設定する）
 1: procedure 句の二項関係ペア結合オペレータ S(s_t, s_f)
 2:     if d_f と d_t が句 then
 3:         return { 二項演算子: biop、句のペア:(d_f, d_t)}
 4:     return nil
 5: procedure 句の二項関係ペア結合オペレータ F(s_t, s_f)
 6:     if d_f と d_t が句 then
 7:         return { 二項演算子: biop、句のペア:(d_f, d_t)}          ▷ 辞書形式（key:value）のデータ
 8:     if d_f が句かつ d_t が命題 then
 9:         d ← insertPhrase(d_t, d_f)
10:         return (d, B_f ∪ B_t)
11:     return nil
```

- 例: 「給湯ボタンか/再沸騰ボタンを……」
  - 入力: d_f: 給湯ボタンか、d_t: 再沸騰ボタンを
  - 出力: d_r: {二項演算子: “or”、句のペア:（d_f，d_t）}

二項演算子を持つ句のペアの分割挿入オペレータ: 句のペアを命題プリミティブに分割して挿入する。

- アルゴリズム:

```
Definition:
    • clone(d): 命題 d の複製を返す。
 1: procedure 二項演算子を持つ句のペアの分割挿入オペレータ F(s_t, s_f)
 2:     if d_t が句 then
 3:         d ← 句 ([d_f, d_t])                                  ▷ 句のインスタンス
 4:         return (d, B_f ∪ B_t)
 5:     if d_t が命題 then
 6:         if d_f のデータ構造 ≠ { 二項演算子: biop、句のペア:(d_{f_{d_f}}, d_{t_{d_f}})} then
 7:             return nil
 8:         d'_t ← clone(d_t)
 9:         d ← insertPhrase(d_t, d_{f_{d_f}})
10:         d' ← insertPhrase(d'_t d_{t_{d_f}})
11:         d'' ← join(d', d, biop)
12:         return (d", B_f ∪ B_t)
13:     return nil
```

- 例: 「給湯ボタンか再沸騰ボタンを/押す」
  - 入力: d_f: {二項演算子: “or”、句のペア:（給湯ボタンか，再沸騰ボタンを）}、d_t: 押す

（?，?）

– 出力: d_r: 押す（?，給湯ボタン）| 押す（?，再沸騰ボタン）

親文節スキップオペレータ: 親文節との結合をスキップする。(「……場合」や「……とき」等の付帯的な状況を表現する体言に命題プリミティブが係るとき、命題の結合オペレータにより句とするのではなく、命題プリミティブとして維持するためのオペレータ)。

- アルゴリズム:

```
1: procedure 親文節スキップオペレータ F(s_t, s_f)
2:     return (d_t, B_f ∪ B_t)
```

- 例: 「給湯ボタンを押す/とき、……」
  - 入力: d_f: 押す（?，給湯ボタン)、d_t: とき
  - 出力: 押す（?，給湯ボタン）

文節の結合順序を制御するために、オペレータ間には優先順位を設定する。オペレータ間の優先順位は、アルゴリズム内に静的に設定されており、句の二項関係ペア結合オペレータ $>$ 句の結合オペレータ $>$ 二項演算子を持つ句のペアの分割挿入オペレータ $>$ 親文節スキップオペレータ $>$ 命題の結合オペレータである。優先順位に従うオペレータの適用は、Algorithm 1 中では、まず line.13–17 で優先順位の高いオペレータを選択し、line.18–21、23–24 でそれぞれ選択されたオペレータを適用するというように実現されている。

命題の結合オペレータ同士は、文節内の読点の有無によってさらに優先度を付け、読点ない文節 $>$ 読点がある文節とした。これにより、読点によって二項演算子の結合順序が順序付けられた「b_a かつ、b_b または b_c」のような文を「$b_a \wedge (b_b \vee b_c)$」と変換する。

上記オペレータを選択するための変換ルールは、文節のセミ形式記述変換ルールとセミ形式記述結合ルールの大きくは 2 種類のルールで構成する。以下にルールの例を挙げる。

文節のセミ形式記述変換ルール: 1 つの文節を句または命題プリミティブへ変換するルールを規定する。

```
Input:
    • b: 文節
 1: procedure NewSemiElement(b)
 2:     if b が用言 then
 3:         d ← 命題プリミティブ（verb=句（[b]））        ▷ 関係語に句（[b]）を持つ命題プリミティブのインスタンス
 4:         if b の feature に「否定表現がある」 then
 5:             (neg of d) ← true                                  ▷ 否定の命題プリミティブにする
 6:         return d
 7:     if b が「場合」、「時」、「中」のいずれかで終わるか、左記に助詞・句読点がついて終わる then
 8:         d ← 命題プリミティブ（constraints=[句（[b]）]）▷ 句（[b]）を制約の先頭に持つ命題プリミティブのインスタンス
 9:     return 句（[b]）                                          ▷ 句のインスタンス
```

例:

- 文節「ボタンを」は句「ボタンを」に変換される。
- 文節「押す」は命題プリミティブ「押す（?，?)」に変換される。

セミ形式記述結合ルール: セミ形式記述同士を結合するためのオペレータを選択するルールを規定

する。

Definition:
- GetBioperator(s_t, s_f): セミ形式記述と文節のリストのペア s_t、s_f から二項演算子決定ルールに従う二項演算子を返す（ただし、$s_t = (d_t, B_t)$、$s_f = (d_f, B_f)$、d_t, d_f はセミ形式記述の句・命題、B_t, Bf はそれぞれ d_t, d_f に対応する文節のリスト）。

Input:
- s_t, s_f: $s_t = (d_t, B_t), s_f = (d_f, B_f)$。$d_t, d_f$ はセミ形式記述の句・命題、または変換過程のデータ構造、B_t, B_f はそれぞれ d_t, d_f に対応する文節のリスト

Output:
- op: オペレータ

```
 1: procedure NewOperator(s_t, s_f)
 2:     if d_f が命題 then
 3:         biop ← GetBioperator(s_t, s_f)
 4:         if biop = nil then
 5:             biop ←"and"                          ▷ デフォルト値。レビュー時に修正する。
 6:         return 命題の結合オペレータ（biop）          ▷ 命題の結合オペレータのインスタンス
 7:     if d_f が句 then
 8:         biop ← GetBioperator(s_t, s_f)
 9:         if biop = nil ∧ B_f の末尾の文節が B_t 中の文節と並列な係り受け構造にある then
10:             biop ←"and"                                  ▷ デフォルト値
11:         if biop ≠ nil then
12:             return 句の二項関係ペア結合オペレータ（biop）
13:         return 句の結合オペレータ（）
```

例: 「赤色の/LED を……」

- 入力: s_f:（d_f: 句「赤色の」、B_f: [赤色の]）、s_t:（d_t: 句「LED を」、B_t: [LED を]）
- 出力: 句の結合オペレータ

セミ形式記述結合ルール中に定義する二項演算子決定ルールの例を挙げると次である。

二項演算子決定ルール:

```
 1: procedure GetBioperator(s_t, s_f)
 2:     b ← B_f の末尾の文節
 3:     if b が接続助詞「か」∨b が接続助詞「または」 then
 4:         return "or"
 5:     if b が接続助詞「かつ」 then
 6:         return "and"
 7:     if b が用言 ∧ last(B_f) の活用形が条件形 then
 8:         return "imply"
 9:     if b の末尾が「場合」、「時」、「中」のいずれかで終わるか、左記に助詞・句読点がついて終わる then
10:         return "imply"
11:     return nil
```

変換ルールでは、Juman++/KNP より得られる複数の文法データを組み合わせてオペレータの選択条件を定義する。オペレータの選択条件を厳密な論理関係で表現するために、本研究ではスクリプト言語 Python を用い、KNP の Python バインディングである PyKNP[6] を用いて図 3.7 に示すようにルールを記述する。

図 3.7 の `new_semi_element(b)` が Algorithm 1 の NewSemiElement(b) に相当する。`new_operator(dt, bt, ds, bs)` が NewOperator(s_t, s_s) に相当し、s_t = (dt, bt)、s_s = (ds, bs) である。`b` は PyKNP より得られる文節のデータ構造、`bt`、`bs` は文節のデータ構造のリストを表す。文節のデータ構造からは、`features` という属性から辞書の形式で文法データが取得できる（line.2）。`dpndtype` という属性からは係り受けの構造が取得でき（line.15）、この値が`'P'` のとき、

6） `http://nlp.ist.i.kyoto-u.ac.jp/index.php?PyKNP`

```
def new_semi_element(b):
  if b.features.get('用言'):
    ph = 句([b])
    return 命題プリミティブ(verb=ph)
  # 以降省略

def new_operator(dt, bt, dc, bc):
  if isinstance(dc, 命題):
    # 二項演算子決定ルール
    op = get_bioperator(dt,bt,dc,bc)
    op = 'and' if not op else op
    return 命題の結合オペレータ(op)
  if isinstance(dc, 句):
    # 並列な係り受け構造
    if bc[-1].dpndtype == 'P':
      # 二項演算子決定ルール
      op = get_bioperator(dt,bt dc,bc)
      op = 'and' if not op else op
      return 句の二項関係ペア結合オペレータ(op)
  # 以降省略

def get_bioperator(dt, bt, dc, bc):
  # 句読点を除く末尾の形態素を得る
  mrph = get_last_mrph(bc[-1])
  if (mrph.bunrui == '接続助詞' and
      mrph.midasi in ('か', 'または')):
    return 'or'
  # 以降省略
```

図 3.7　Python での文節のセミ形式記述変換ルールとセミ形式記述結合ルールの記述例

並列な係り受け構造を持つことを示す。`get_bioperator(dt, bt, ds, bs)` が二項演算子決定ルールに相当する。`last_mrph` は PyKNP より得られる形態素のデータ構造であり、`bunrui` という属性で形態素の分類が得られ、`midasi` という属性で形態素の見出し語が得られる。

3.4　命題ネットワークによる論理構造の可視化

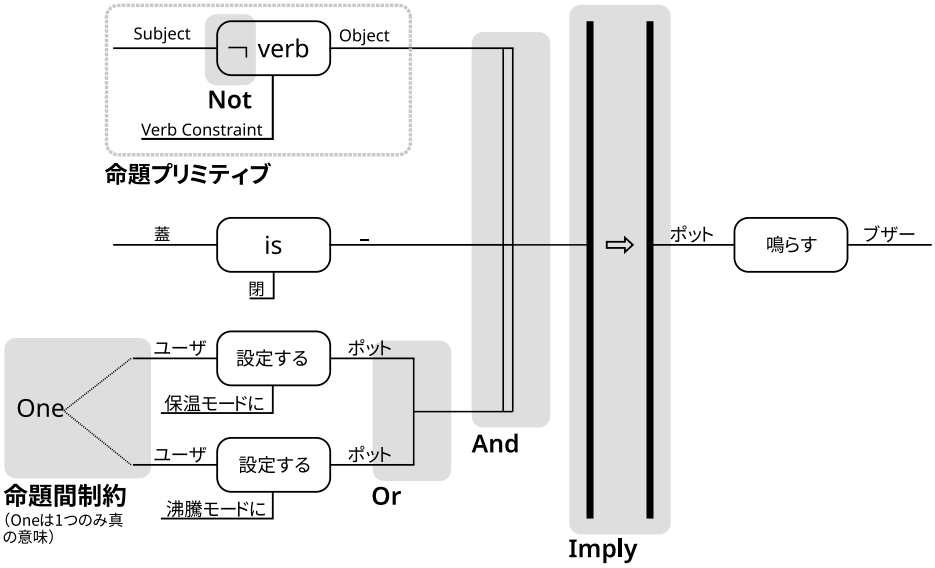


図 3.8　命題ネットワークの例

組込み機器開発を行う企業 2 社へのヒアリングと観察の結果から、システムテストのテストケース設計では、抽出されたテストの入力項目・確認項目項目は、左記項目間の論理関係に基づいて、テストで入力・確認する具体値が決定されることがわかっている。この具体値の設定にあたっては、抽出された入力項目・確認項目間の論理関係をシステム設計者の設計意図どおりに解釈する必要がある。一方、自然言語で記述される入力項目・確認項目の論理関係は曖昧で、複数通りの解釈が取りうるため、提案する変換アルゴリズムにより、可能な解釈のひとつとして機械的に変換したセミ形式記述が表現する論理関係は、システム設計者の意図する解釈とは異なる可能性がある。また、セミ形式記述への変換ルールの不足により、変換結果が誤っていることがある。

このため、変換されたセミ形式記述について、入力項目・確認項目間の論理関係の誤りを除去するために、左記論理関係に着目したレビューを実施する。ところが、テキストベースで表現されるセミ形式記述では、括弧の対応や論理演算子の結合順位（And の方が Or よりも結合順位が高い、など）に注意しながら論理関係を解釈しながらレビューを行う必要があり、複雑な論理関係は論理関係の解釈が困難となる。

本研究では、論理関係の誤りのレビューにあたり、命題間の論理関係の把握を容易化するために、命題ネットワークと呼ぶセミ形式記述の可視化図式を開発した（図 3.8）。図 3.8 では、括弧の対応や演算子の優先順位などを人手により注意せずとも、階層のある命題間の論理関係を解釈可能にするために、論理演算子で結ばれた命題同士を線で結んで表現する。また、連続する And・Or による同時に成立する命題のまとまりや、選択的に成立する命題のまとまりを把握できるように、連続する And、Or のオペランド（セミ形式記述の文法（図 3.2）における expression）は、二項演算子として階層を深く表示する代わりに、一つの論理演算子のオペランドとしてまとめて可視化する（例: 図 3.8 の And）。

上記図式表現により、テスト設計者は、数学的な演算子の結合順序を頭の中で組み立てずとも、命題プリミティブ間の論理関係の解釈を命題プリミティブ間の線の接続から解釈可能にした。

命題ネットワーク上でのレビューにより発見された入力項目・確認項目間の論理関係の誤りを元のセミ形式記述上で修正することによって、論理関係の誤りの除去されたセミ形式記述を得る。命題ネットワーク上のレビューでは次に着目して論理関係誤りを修正する。

1. 論理演算子の修正
   本節冒頭で述べたとおり、セミ形式記述への変換は、自然言語で書かれた仕様から可能な解釈のひとつを取り出したものであるため、システム設計者の意図する解釈と異なる場合がある。この解釈誤りを論理演算子の修正として実施する。
2. 入力項目・確認項目の欠落の補完
   自然言語で書かれた仕様では、テストに設定すべき入力項目・確認項目の記述が漏れている場合がある。命題ネットワーク上のレビューでは、描画されている入力項目・確認項目に着目して、十分な命題が記載されているかをレビューし、不足する記述の補完を行う。この不足する記述の発見支援のために、命題ネットワークは、後述するように複数の命題を統合して描画する機能を有する。

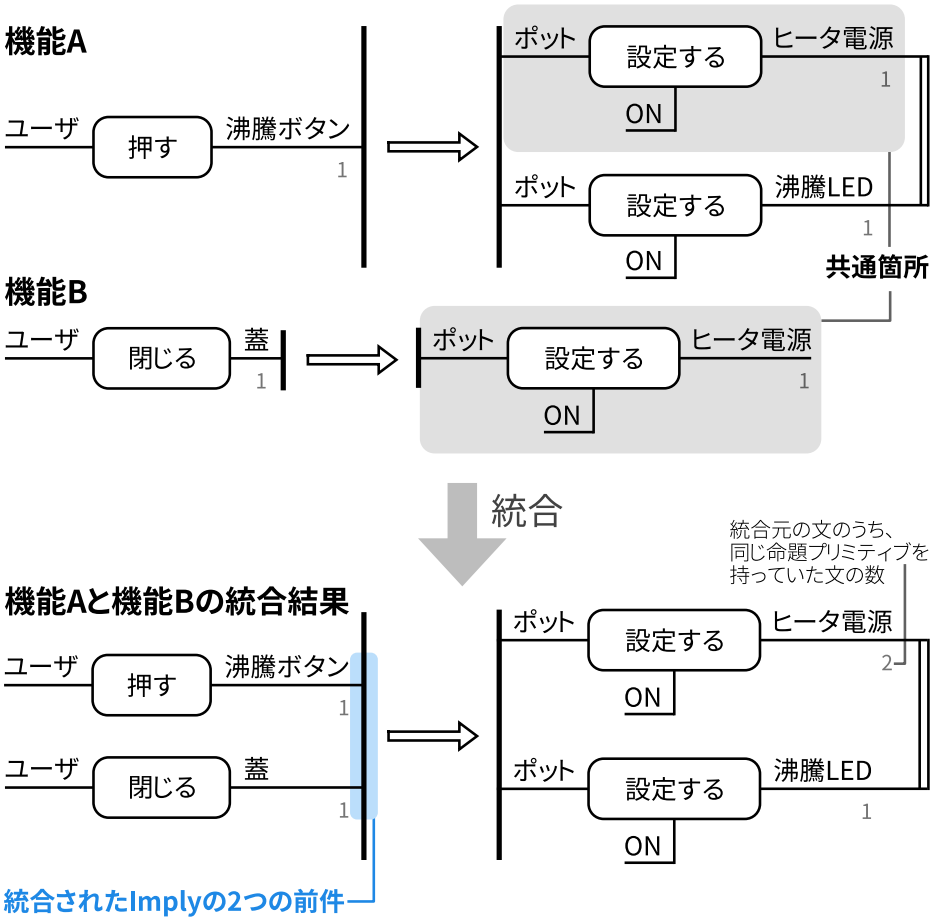


図 3.9　命題プリミティブのマージ結果の例

3. 2 文以上にまたがった文の統合
   自然言語仕様書からセミ形式記述への変換アルゴリズムは、1 文単位で変換を行うため、2 文以上にまたがって記述される 1 つの仕様（例: 「A は、B である。ただし、C が D のときに限る。」）が 2 文以上のセミ形式記述に変換されることがある。この 2 文以上に分かれている仕様の統合を行う。
4. 「→」を持たない文の修正
   セミ形式記述では、機能の仕様を「入力項目→確認項目」と表現するため、「→」を持たない文は入力項目・確認項目が明示されていない文を意味する。他の文との統合や入力項目・確認項目の補完により、Imply の形式のセミ形式記述へと修正する。

セミ形式記述への変換は、ルールベースで行うので、上記の誤りには、元の仕様書と変換ルールの不足の 2 種類がありうる。修正に当たって、元の自然言語仕様書に含まれている論理関係がセミ形式記述に反映されていない場合（誤変換や欠落している場合）には、変換ルールの問題とし、変換ルールの正確さの向上のためにルールの追加・修正を行う。

上述までは、単体の機能のテストケースのレビューについて説明をした。加えて、命題ネット

Algorithm 2 命題プリミティブのマージアルゴリズム

Definition:

T: a set of node types;
$T = \{Atomic, And, Or, Imply\}$
n: a node; $n = (t, A), t \in T$
A: the arguments of a node;

$$A = \begin{cases} (s, o, r, C) & (t = Atomic) \\ \text{a set of nodes} & (t \neq Atomic) \end{cases}$$

s: subject, o: object, r: relation, C: a set of constraints

Input:

n_l: a node; $n_l = (t_l, A_l)$
n_r: a node; $n_r = (t_r, A_r)$

Output:

N_m: a set of merged nodes

1: **procedure** MERGE(n_l, n_r)
2: $A_{new} \leftarrow \{\}$
3: $merged \leftarrow false$
4: **if** $t_l = t_r = Atomic$ **then**
5: define s_l, o_l, r_l, and C_l where $A_l = (s_l, o_l, r_l, C_l)$
6: define s_r, o_r, r_r, and C_r where $A_r = (s_r, o_r, r_r, C_r)$
7: **if** $s_l = s_r \wedge o_l = o_r \wedge r_l = r_r$ **then**
8: **return** $\{(t_l, (s_l, o_l, r_l, C_l \cup C_r)\}$
9: **return** $\{n_l, n_r\}$ ▷ not merged
10: **else if** $t_l = Atomic \vee t_r = Atomic$ **then**
11: define t_k and A_k where $t_k \in \{t_l, t_r\}$, $A_k \in \{A_l, A_r\}$, and $t_k \neq Atomic$
12: define n_k where $n_k = (t_k, A_k)$
13: **for all** $n \in A_k$ **do**
14: $A_{merged} \leftarrow Merge(n, n_k)$
15: $A_{new} \leftarrow A_{new} \cup A_{merged}$
16: $merged \leftarrow merged \vee |A_{merged}| = 1$
17: **if** merged **then**
18: **return** $\{(t_k, A_{new})\}$
19: **return** $\{n_l, n_r\}$ ▷ not merged
20: **if** $t_l = t_r$ **then** ▷ neither t_l nor t_r is $Atomic$
21: **if** $\exists n \in A_r$ such that $n \in A_l$ **then**
22: **return** $(t_l, A_l \cup A_r)$
23: **return** $\{n_l, n_r\}$ ▷ not merged
24: **for all** $n \in A_l$ **do**
25: $A_{merged} \leftarrow Merge(n, n_r)$
26: $A_{new} \leftarrow A_{new} \cup A_{merged})$
27: $merged \leftarrow merged \vee |A_{merged}| = 1$
28: **if** $merged$ **then**
29: **return** $\{(t_l, A_{new})\}$
30: **return** $\{n_l, n_r\}$ ▷ not merged

ワークでは、同じ入力項目・確認項目を持つ機能を統合・可視化を行うことで、論理関係の定義漏れの発見の支援も行う。テストケースの設計においては、機能の組み合わせによる不具合を防止するために、共通する入力項目・確認項目を持つ機能を組み合わせて動作させたときに不具合が起きないかを確認するために、複数の機能を組み合わせ、同じテストケースでテストを実施する。一方で、機能の仕様が異なる節に分散して書かれることで、それら機能の間にある論理関係が定義されないことがある。複数の機能を組み合わせてテストする際には、それら共通する入力項目・確認項目を持つ機能を発見し、さらにそれら機能の間の論理的な関係（排他的に動作するのか、同時に動作しうるのかなど）を定義する必要がある。

本研究では、こうした複数機能を組み合わせたテストケースの設計支援のために、共通する入力項目・確認項目を持つ機能を統合・可視化することで、共通する入力項目・確認項目を持つにもかかわらず、機能間の論理関係が未定義となっている機能の発見を支援する。

共通する入力項目・確認項目を保つ機能の統合は、Algorithm 2 のアルゴリズムによって実現する。Algorithm 2 では、各入力項目・確認項目に相当する命題プリミティブ「関係語（主体語，対

象語，制約語)」の同一性を、制約を除く 3 語（関係語、主体語、対象語）によって判定して、1 つの命題プリミティブに統合する。同一性の比較に制約を除くことで、パラメータが違っても、同じ入力項目・確認項目を持つ機能の抽出を行うことができる。各入力項目・確認項目が取りうるパラメータの一覧は統合された命題プリミティブの制約部分から得られる。

統合の結果は、命題ネットワークによって図 3.9 に示すように可視化される。図 3.9 では、確認項目「設定する（ポット，ヒータ電源，ON)」が共通している 2 つの機能の文が 1 つの Imply に統合され、2 つの入力項目から共通する 1 つの評価項目を生じることを表現している。テスト設計者は、複数の入力項目・評価項目を持つ統合された Imply を参照し、組み合わせテスト対象の機能群を収集する。統合の結果として複数の入力項目・確認項目を持つ Imply では、入力項目・確認項目同士に論理関係が定義されていない部分が可視化される（図 3.9 の「押す（ユーザ，沸騰ボタン)」と「閉じる（ユーザ，蓋)」の間)。命題ネットワーク上でのレビューにより、この部分の論理関係を補う——例えば Or 演算子を用いて、「押す（ユーザ，沸騰ボタン）| 閉じる（ユーザ，蓋）–> 設定する（ポット，ヒータ電源，ON）& 設定する（ポット，沸騰 LED，ON)」と表現する——ことで、論理関係の見落としを補完できる。

統合の元となる機能の文同士は、共通する命題と、差分の命題（一方に含まれて他方に含まれていない命題）が存在する。差分の命題は、仕様の違いである場合と、記述漏れによって、一方の命題にしか記述できていない場合がある。本研究では、命題の記述漏れの発見を促すために、統合した命題に含まれる命題プリミティブが、いくつの文が同じ命題プリミティブを持っているかを示す数値を表示することで、他の命題プリミティブよりも小さな値を持つ命題プリミティブとして差分の命題プリミティブを可視化する。この数値が 2 以上のとき、複数文が共通する命題プリミティブを含むことを示し、他の命題プリミティブよりもこの数値が小さい命題プリミティブについて、仕様の違いであるためか、記述漏れであるためかを検討可能にした。

図 3.9 の例では機能 A の「設定する（ポット，沸騰 LED，ON)」が、機能 B には存在しないことが統合結果の命題で示されている。共通する命題プリミティブ「設定する（ポット，ヒータ電源, ON)」と And で結合されているにも関わらず、差分の命題プリミティブとなっている「設定する（ポット，沸騰 LED，ON)」についてレビューすることで、記述漏れを除去できる。

3.5 セミ形式記述からのデシジョンテーブル生成

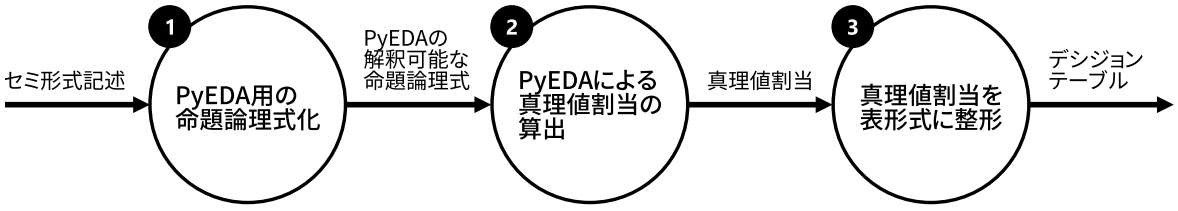


図 3.10 セミ形式仕様書をデシジョンテーブルに変換する Data Flow Diagram（DFD）

システム仕様書から抽出した入力項目・確認項目を用いてテストケースを作成するに当たり、入力項目・確認項目の組み合わせを見落としてしまうことを防ぐために、本研究では、セミ形式記述で書かれた仕様から図 3.10 のプロセスによって、デシジョンテーブル [45] 形式のテストケースを自動生成する。セミ形式記述では、システムの仕様を論理記述するため、論理記述されたシステム仕様を充足する真理値割当を求めることで取りうる入力項目・確認項目の組み合わせを得ることができる。図 3.10 のプロセスでは、セミ形式記述上で機能の仕様を表現する「入力項目 → 確認項目」という形式の命題について、真理値割当の算出と表形式への整形を行うことでデシジョンテーブルを生成する。

真理値割当の算出は、命題論理式を充足する真理値割当を自動算出可能な PyEDA [46] を用いて実現する（図 3.10 の ②）。「入力項目 → 確認項目」という演算子 Imply を用いて書かれた機能の仕様の前件部分（入力項目）の真理値割当を条件欄、後件部分（確認項目）の真理値割当を動作欄として埋めることで、デシジョンテーブルを完成させることができる（③）。得られたデシジョンテーブルには取りうる入力項目の組み合わせが網羅されるので、漏れのない入力項目の組み合わせでテストを実行できる。また、膨大になりがちな入力項目の組み合わせへの対策として、Don't Care と呼ぶ、真と偽のいずれの値をとっても論理式全体の真偽に影響しないことを意味する割当値を使い、テーブルサイズを縮小する。

表 3.1　図 3.6 から生成したデシジョンテーブル

		テストケース				
		前件真		前件偽		
	命題	1	2	3	4	5
条件	is(?, _, 沸騰モード中)	F	T	F	F	T
	is(?, _, 給湯 LOCK 中)	T	-	F	T	-
	押す (?, 給湯ボタン)	T	T	-	F	F
動作	給湯する (?, ?)	F	F	-	-	-
	鳴らす (?, ブザー)	T	T	-	-	-

表中の記号は、F: 真、T: 偽、-: Don't Care である。

提案アルゴリズムにより、図 3.6 から表 3.1 に示すデシジョンテーブルが生成できる。

3.6　セミ形式記述への変換アルゴリズムの評価と考察

3.6.1　評価内容

本節では、本論文で提案する日本語で書かれた仕様書からセミ形式記述への変換アルゴリズム、及びセミ形式記述からテストケースへの変換アルゴリズムの評価のために、ケーススタディによる

詳細な変換ステップの評価と、仕様書に変換アルゴリズムを適用した場合の Precision と Recall による定量評価を行う。

3.6.1.1 変換アルゴリズムのケーススタディ

ケーススタディでは、架空の電気ポットの仕様 3 文を用いて、次に示す項目を確認する。

1. 変換ルールの追加により変換の正確さを改善できること
2. 「入力項目→確認項目」の形式で記述されたセミ形式記述がデシジョンテーブル形式のテストケースへ自動変換されること

上記項目の確認のために、次に示す架空の電気ポットの 3 つの仕様文を対象に変換を実施した。

Sentence A: 入力項目・確認項目を 1 つずつ持つ仕様

日本語の仕様文:「給湯ボタンを押したら、お湯を排出する。」

対応するセミ形式の仕様文:「押す（?，給湯ボタン）–> 排出する（?，お湯）.」

Sentence B: 論理関係に曖昧さを含む仕様

日本語の仕様文:「沸騰モード中に給湯ボタンを押すか、再沸騰ボタンを押したら、ブザーを鳴らしてお湯を排出しない。」

対応するセミ形式の仕様文:「is（?，_，沸騰モード中）& （押す（?，給湯ボタン）| 押す（?，再沸騰ボタン））–> 鳴らす（?，ブザー）& !排出する（?，お湯）.」

Sentence C: ルールを追加しないと不正確な変換がなされる仕様

日本語の仕様文:「給湯ボタンを押すことで、ポットはお湯を排出する。」

対応するセミ形式の仕様文:「押す（?，給湯ボタン）–> 排出する（ポット，お湯）.」

上記機能の仕様を Juman++/KNP にて構文解析を行うと、図 3.11 のような係り受け構造となる。

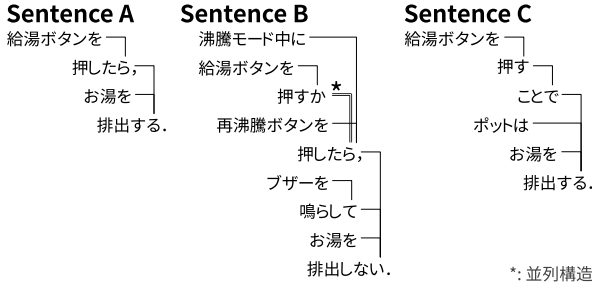


図 3.11 評価用の文の係り受け構造

3.3.2 小節のルールにより、Sentence A は次のステップで変換される。

1. 「排出する」が用言なので、文節のセミ形式記述変換ルールに従い、命題プリミティブ「排出する（?，?)」へ変換（Algorithm 1 line.4）
2. 「押したら」が用言なので、文節のセミ形式記述変換ルールに従い、命題プリミティブ「押す（?，?)」へ変換（Algorithm 1 line.4）

3. 「押したら」の子文節「給湯ボタンを」が体言なので、文節のセミ形式記述変換ルールに従い、句「給湯ボタンを」に変換（Algorithm 1 line.4）
4. 「給湯ボタンを」が句なので、セミ形式記述結合ルールに従い、句の結合オペレータを選択（Algorithm 1 line.12）
5. 句の結合オペレータにより、句「給湯ボタンを」を命題プリミティブ「押す（?，?)」と結合し、命題プリミティブ「押す（?，給湯ボタン)」を生成（Algorithm 1 line.24）
6. 命題プリミティブ「押す（?，給湯ボタン)」が命題なので、セミ形式記述結合ルールに従い、命題の結合オペレータを選択（Algorithm 1 line.14）
7. 「お湯を」が体言なので、文節のセミ形式記述変換ルールに従い、句「お湯を」に変換（Algorithm 1 line.4）
8. 「お湯を」が句なので、セミ形式記述結合ルールに従い、句の結合オペレータを選択（Algorithm 1 line.12）
9. 命題の結合オペレータよりも句の結合オペレータの方が優先度が高いので、句「お湯を」を命題プリミティブ「排出する（?，?)」と結合し、命題プリミティブ「排出する（?，お湯)」を生成（Algorithm 1 line.24）
10. 命題の結合オペレータにより、命題プリミティブ「押す（?，給湯ボタン)」を命題プリミティブ「排出する（?，お湯)」と結合し、命題プリミティブ「押す（?，給湯ボタン）–> 排出する（?，お湯)」を生成（Algorithm 1 line.24）

Algorithm 1 に従うと、Sentence B、Sentence C は、次のように変換される。

Sentence B:「is（?，_，沸騰モード中）& 押す（?，給湯ボタン）| 押す（?，再沸騰ボタン）–> 鳴らす（?，ブザー）& !排出する（?，お湯).」

Sentence C:「排出する（ポット，お湯，給湯ボタンを押すことで).」

Imply の形式で記述されている Sentence A、Sentence B のセミ形式記述をデシジョンテーブルに変換した結果を表 3.2、表 3.3 に示す。

表 3.2　Sentence A のセミ形式記述から生成したデシジョンテーブル

	テストケース	
	前件真	前件偽
命題	1	2
条件　押す（?，給湯ボタン）	T	F
動作　排出する（?，お湯）	T	-

表 3.3　Sentence B のセミ形式記述から生成したデシジョンテーブル

		テストケース				
		前件真			前件偽	
	命題	1	2	3	4	5
条件	is（?，_，沸騰モード中）	F	T	T	F	T
	押す（?，給湯ボタン）	-	F	T	-	F
	押す（?，再沸騰ボタン）	T	T	-	F	F
動作	鳴らす（?，ブザー）	T	T	T	-	-
	排出する（?，お湯）	F	F	F	-	-

3.6.1.2　セミ形式記述への変換アルゴリズムの定量評価

定量評価のために、電気ポットの仕様書 [47] の 1 章–7 章のうち、図表、数式を含む文を除く 98 文を対象にセミ形式記述への変換を実施した。本論文の著者 1 名による手作業での変換結果を正解データとし、変換結果を比較して、命題プリミティブの抽出・論理関係の同定のそれぞれについて、次の式で定まる Precision（式（3.1））と Recall（式（3.2））を求めた。

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.2}$$

命題プリミティブの抽出結果における TP（True Positive）、FP（False Positive）、FN（False Negative）は、98 文の仕様文をそれぞれ S_i（$1 \leq i \leq 98$）とし、変換アルゴリズムによる S_i の変換結果に含まれる命題プリミティブの集合を P_{a_i}、人手による S_i の変換結果に含まれる命題プリミティブの集合を P_{m_i} として、それぞれ次に当たるものを $1 \leq i \leq 98$ にわたって集計した。

- TP: $p \in P_{a_i} \wedge p \in P_{m_i}$ となる p の数
- FP: $p \in P_{a_i} \wedge p \notin P_{m_i}$ となる p の数
- FN: $p \notin P_{a_i} \wedge p \in P_{m_i}$ となる p の数

論理関係の同定における TP、FP、FN は、変換アルゴリズムによる S_i の変換結果に含まれる論理演算子（Not・And・Or・Imply）の集合を O_{a_i}、人手による S_i の変換結果に含まれる論理演算子の集合を O_{m_i}、ある命題 x に含まれる関係語の集合を W_x、x に単項演算子がついているかを表す関数を has_not(x)、$x \neq y$ なる命題 y との論理関係（And・Or・Imply）を $r_{x,y}$ で表すとき、それぞれ次に当たるものを $1 \leq i \leq 98$ にわたって集計した。

- TP: 次の項目の合計
  - $(x \in P_{a_i} \wedge t \in P_{m_i}) \wedge (W_p = W_t \wedge \text{has\_not}(x) \wedge \text{has\_not}(t))$ を満たす x の数

– $(x, y \in P_{a_i} \land x \neq y) \land (t, u \in P_{m_i} \land t \neq u) \land (W_p = W_t \land W_q = W_u) \land (r_{x,y} = r_{t,u} \land r_{x,y} \in O_{a_i} \land r_{t,u} \in O_{m_i})$ を満たす $r_{x,y}$ の数

• FP: 次の項目の合計

– $(x \in P_{a_i} \land t \in P_{m_i}) \land (W_p = W_t \land \text{has\_not}(x) \land \neg\, \text{has\_not}(t))$ を満たす x の数 $W_p = W_t$ を満たす x の数

– $(x, y \in P_{a_i} \land x \neq y) \land (t, u \in P_{m_i} \land t \neq u) \land (W_p = W_t \land W_q = W_u) \land (r_{x,y} \in O_{a_i} \land r_{t,u} \in O_{m_i} \land r_{x,y} \neq r_{t,u}) \lor (r_{x,y} \in O_{a_i} \land r_{t,u} \notin O_{m_i})$ を満たす $r_{x,y}$ の数

• FN: 次の項目の合計

– $(x \in P_{a_i} \land t \in P_{m_i}) \land (W_p = W_t \land \neg\, \text{has\_not}(x) \land \text{has\_not}(t))$ を満たす x の数

– $(x, y \in P_{a_i} \land x \neq y) \land (t, u \in P_{m_i} \land t \neq u) \land (W_p = W_t \land W_q = W_u) \land (r_{x,y} \in O_{a_i} \land r_{t,u} \in O_{m_i} \land r_{x,y} \neq r_{t,u}) \lor (r_{x,y} \notin O_{a_i} \land r_{t,u} \in O_{m_i})$ を満たす $r_{x,y}$ の数

表 3.4　セミ形式記述への変換の Precision と Recall

	TP	FP	FN	Precision	Recall
命題プリミティブの抽出	147	65	62	0.69	0.70
論理関係の同定	103	23	26	0.82	0.81

集計の結果、人手での変換結果に含まれる 212 個の命題プリミティブ、128 個の論理演算子についての変換の Precision と Recall は表 3.4 に示す通りとなった。

3.6.2　評価結果の考察

3.6.2.1　変換アルゴリズムのケーススタディの考察

本項では、3.6.1.1 項の評価結果について考察する。

まず、セミ形式記述により元の日本語文中の論理関係の曖昧さが除去されたことを示す。3.6.1.1 項の Sentence A をテストする際には、機能の入力項目として「給湯ボタンを押す」という操作を行い、確認結果として「お湯を排出する」ということを確認する必要がある。入力項目・確認項目をそれぞれ 1 つずつ持つ Sentence A は、論理演算子 Imply を使って、「押す（?，給湯ボタン）–> 排出する（?，お湯）.」のように変換された。変換されたセミ形式記述は、元の仕様文にある「給湯ボタンを押す」と「お湯を排出する」という入力項目・確認項目を反映し、それぞれ Imply の前件・後件に期待通りに記述された。入力項目・確認項目は、Imply の前件・後件を見ることで一意に把握できる。

ここから、入力項目・確認項目をそれぞれ 1 つずつ持つ最も単純な形式な仕様文がセミ形式記述で表現でき、論理関係の曖昧さなく Imply の前件・後件から入力項目・確認項目を把握可能なことを確認した。

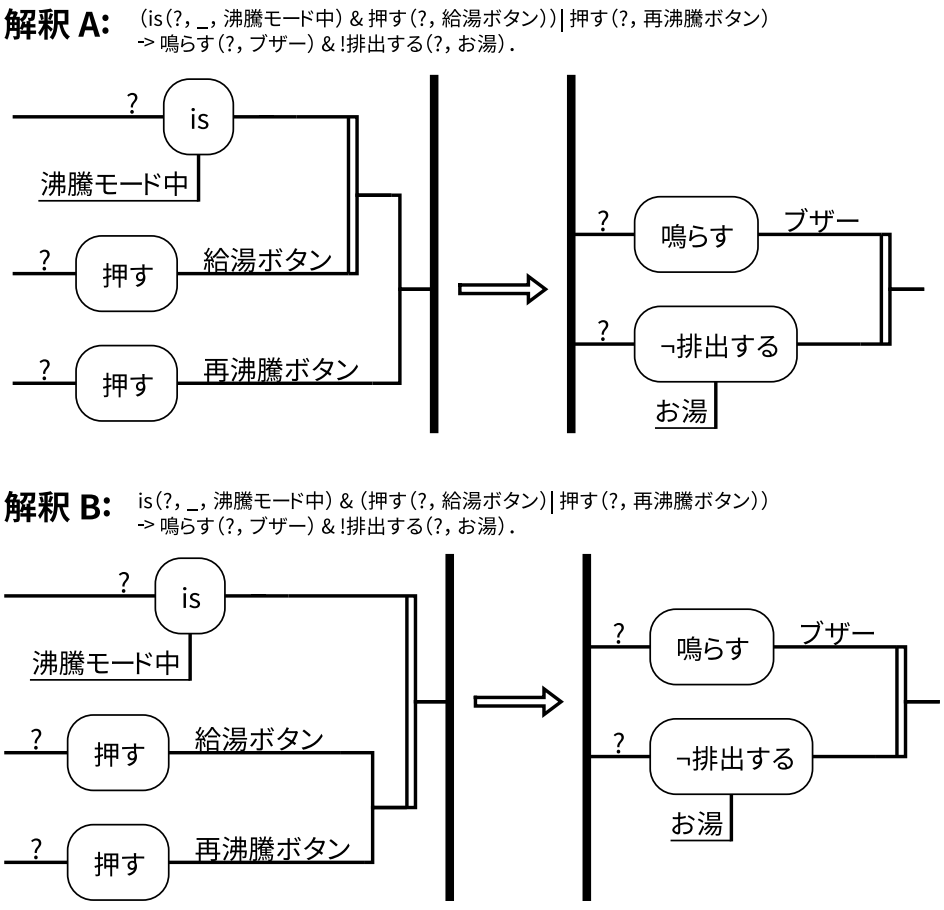


図 3.12　3.6.1.1 項の Sentence B の解釈

続いて論理関係に曖昧さを含んだ仕様文である 3.6.1.1 項の Sentence B について考察する。Sentence B では、付帯状況を表す「沸騰モード中」という表現が、「給湯ボタンを押すとき」と「再沸騰ボタンを押すとき」の両方に制約するのか、あるいは「給湯ボタンを押すとき」のみを制約するのかが曖昧になっている。解釈可能な論理関係を命題ネットワークで表現すると、図 3.12 の 2 通り存在する。

解釈 B からデシジョンテーブルを生成すると、表 3.5 になる。解釈 A から生成された表 3.3 の 1 列目のテストケースと表 3.5 は 3 列目のテストケース（太字部分）は、ともに入力項目の「is（?, _, 沸騰モード中)」が偽であるが、確認項目が異なっている。セミ形式記述では機能を Imply で表現し、入力項目が成立しないとき（前件偽のとき）の確認項目は真偽が不定であることを示すために don't care で表現する。Sentence B の入力項目が不成立の時に確認項目が成立しない場合、解釈 A の 1 列目では、確認項目が成立することを確認するテストケースとなるが、一方、解釈 B の 3 列目は確認項目が成立しないことを確認するテストケースとなる。解釈 A・解釈 B のいずれか誤った解釈のもとでテストケースを作成した場合、不具合を見落としてしまう。

図 3.12 では、解釈 A では「(is（?, _, 沸騰モード中）& 押す（?, 給湯ボタン))」、解釈 B では「(押す（?, 給湯ボタン）| 押す（?, 再沸騰ボタン))」と異なる命題プリミティブがペアになって

表 3.5　図 3.12 の解釈 B から生成したデシジョンテーブル

		テストケース			
		前件真		前件偽	
	命題	1	2	**3**	4
条件	is（?，_，沸騰モード中）	T	T	**F**	T
	押す（?，給湯ボタン）	F	T	**-**	F
	押す（?，再沸騰ボタン）	T	-	**-**	F
動作	鳴らす（?，ブザー）	T	T	**-**	-
	排出する（?，_，お湯）	F	F	**-**	-

いることが可視化されており、セミ形式記述上でも解釈ごとに異なる論理関係となっている。ここから、セミ形式記述で表現することで論理関係の曖昧さが除去され、一意に解釈できる仕様が記述されている。このセミ形式記述上で正しい解釈に修正し、テストケースを生成することで、論理関係の解釈の誤りによって誤ったテストケースを作成することを防げる。

表 3.2、表 3.3 のデシジョンテーブルは、3.5 節の方法により自動生成された。これにより、セミ形式記述上で一意に解釈可能となった機能の仕様から人為的な操作ミスによる誤りの混入なく、テストケースを生成できる。

次に、ルール追加により Sentence C の日本語の仕様文をセミ形式記述へ変換可能となることを示す。Sentence C のセミ形式記述への誤変換は、3.3.2 小節のセミ形式記述結合ルール中に、「用言 p+ことで」という表現について、p を命題プリミティブに変換するルールがないために起こった。「給湯ボタンを」は体言であり、かつ「押す」は用言であるため、両者は文節のセミ形式記述変換ルールにより、句「給湯ボタンを」と命題プリミティブ「押す（?，?)」となる。句「給湯ボタンを」はセミ形式記述結合ルールにより句の結合オペレータが選択されることから、句「給湯ボタンを」と命題プリミティブ「押す（?，?)」の結合の結果までは「押す（?，給湯ボタン)」という命題プリミティブに変換される。

一方、「押す（?，給湯ボタン)」の親文節「ことで」は体言のため句となっており、ここに命題プリミティブの「押す（?，給湯ボタン)」が命題の結合オペレータで結合されることで、「『押す（?，給湯ボタン)』＋『ことで』」の句となってしまう。この句が「ポットはお湯を排出する」の部分から生成された命題プリミティブ「排出する（ポット，お湯)」と結合されて、誤ったセミ形式記述「排出する（ポット、お湯、給湯ボタンを押すことで)」となってしまう。

上記の誤った変換をルール追加により改善可能なことを示す。「給湯ボタンを押すことで」を、「押す（?，給湯ボタン）–>」と変換するためには、「用言+ことで」の結合において、次を実現するルールが必要となる。

1. 「用言」部分から生成される命題プリミティブ p を維持すること
2. 「ことで」を二項演算子決定ルールで Imply として解釈すること

これはセミ形式記述結合ルールとセミ形式記述結合ルールに次のようなルールを追加することで実現できる。

- 追加のセミ形式記述結合ルール:

Definition:
Output:
1: **procedure** NEWOPERATOR(s_t, s_f)
2: **if** d_f が命題プリミティブであり、d_t が「ことで」という文節から構成される句である **then**
3: **return** 親文節スキップオペレータ（）

追加の二項演算子決定ルール:

1: **procedure** GETBIOPERATOR(s_t, s_f)
2: **if** B_f の末尾の文節が「ことで」である **then**
3: **return** “imply”

上記ルール追加により、変換アルゴリズムにより対応可能な表現を拡充できる。ここから、変換ルールの追加により、漸進的に変換の正確さを改善できることを示した。

以上、ルールベースの変換アルゴリズムにより、日本語の仕様文をセミ形式記述に変換することで日本語の仕様文の論理関係の曖昧さを除去できることを示した。また、変換されたセミ形式記述により、システム仕様からテストケースへの人為的な誤りの混入を防ぐ自動的なテストケース生成が実現されていることを示した。

3.6.2.2 セミ形式記述への変換アルゴリズムの定量評価の考察

3.6.1.2 項の結果について、提案するセミ形式記述への変換アルゴリズムにより、過半数の命題プリミティブ・論理関係を正しく変換できた（表 3.4）。本項では以降、変換誤りの原因について考察する。

自動変換の結果と人手による変換結果を比較すると、次の項目に起因する変換誤りが存在した。

1. 変換ルール・オペレータの不足による誤り
   (a) 2 文節からなる複合語を 1 つの句にまとめるルール・オペレータが存在しない
   (b) 論理関係の同定ルールが不足している
2. 論理関係の結合順序の誤り
3. Juman++/KNP による構文解析結果の誤り

1. の誤りは、本評価時点でのルール・オペレータの不足に基づくものであるため、次のようにルール・オペレータを追加することで解決できる。

1a. の例は、「システム全体と/して、/以下の/動作仕様を/満たさなければなりません。」という文が、「する（?，?，システム全体と）& 満たす（?，以下の動作仕様）.」と変換されたものである（「/」は文節区切りを表す）。正解データでは、「満たす（?，以下の動作仕様 システム全体として）.」のように、「として」1 まとまりで英単語の「as」に相当する意味を表現した。一方、変換アルゴリズムの結果では「と」と「して」まとめて複合語として扱うルール・オペレータがない

ため、「do with a whole system」における「do with」という意味に誤変換された。

この誤りは、次のようなオペレータを変換アルゴリズムに追加することによって解決できる。

```
1: procedure 複合語結合オペレータ F(s_t, s_f)
2:     d ← 句 ([d_f, d_t])                                  ▷ 句のインスタンス
3:     return (d, B_f ∪ B_t)
```

本オペレータを命題の結合オペレータよりも高い優先順位に設定することで、「して」と別の命題を結合する前に、複合語の句として結合できる。

このオペレータの追加により、次のようなルールをセミ形式記述結合ルールの先頭に追加することで、本例のような複合語に対応できる。

```
Output:
    • op: オペレータ
1: procedure NewOperator(s_t, s_f)
2:     if B_f 末尾の文節が「と」で終わる ∧ B_t 先頭の文節が「して」で始まる then
3:         return 複合語結合オペレータ ()
```

1b. の例は、「ロック中は、/給湯ボタンを/押しても/水は出ません。」という文が、「is（?，_，ロック中）-> 押す（?，給湯ボタン）& !出る（水，?).」と変換されたものである。正解データとしては、「is（?，_，ロック中）& 押す（?，給湯ボタン）-> !出る（水，?).」のように、「is（?，_，ロック中)」と「押す（?，給湯ボタン)」が Imply で結合され、「押す（?，給湯ボタン)」と「!出る（水，?)」が And で結合されるものであった。これらは、二項演算子決定ルールの不足によるものであり、「is（?，_，ロック中)」と「押す（?，給湯ボタン)」が Imply で結合された誤りは、「ロック中は」のように、「中」を含む文節のみから二項演算子を判断していたことが原因であり、「押す（?，給湯ボタン)」と「!出る（水，?)」が And で結合されたものは、「用言連用形 + も」という表現について Imply と解釈するルールがなく、二項演算子のデフォルト値の And が採用されたことが原因である。これらは、次のようなルールを二項演算子決定ルール先頭に追加することで解決できる。

```
1: procedure GetBioperator(s_t, s_f)
2:     b_f ← B_f の末尾の文節
3:     b_t ← B_t の末尾の文節
4:     if b_f が用言 + 副助詞「も」 then
5:         return “imply”
6:     if b_t が用言 + 副助詞「も」∧ b_f の末尾が「場合」、「時」、「際」、「中」のいずれかで終わるか、左記に助詞・句読点がついて終わる then
7:         return “and”
```

2. の例は、「温度制御可能な/水位ならば/沸騰状態に/移行し、/ポット内の/水を/加熱します。」という文が、「(is（?，_，温度制御可能な水位）-> 移行する（?，?，沸騰状態に)) & 加熱する（?，ポット内の水).」と変換されたものである。本例は論理演算子の結合順序が誤っており、正解データとしては、「is（?，_，温度制御可能な水位）-> 移行する（?，?，沸騰状態に）& 加熱する（?，ポット内の水).」と変換されるべきものであった。自然言語で書いた文章では論理関係の結合順序を厳密に定義できないため、変換結果には本例のような解釈の誤りが生じうる。この解消には人手による意味の解釈が必要なため、命題ネットワーク上でのレビュー時に修正する。

3. の誤りは、「ユーザが/設定した/タイマの/タイムアウト時、及び/沸騰状態終了時には、/ブザーを/3 回/鳴らします。」について、「is（?, _, タイムアウト時）& is（?, _, 設定したタイマの沸騰状態終了時）–> 鳴らす（ユーザ, ブザー, 3 回）.」と変換されたものである。正解データは、「is（?, _, タイムアウト時）& is（?, _, ユーザが設定したタイマの沸騰状態終了時）–> 鳴らす（?, ブザー, 3 回）.」と設定した。「ユーザ」は Juman++/KNP による構文解析の結果、「鳴らします」に係ると判断されるため、変換アルゴリズムによる変換の結果では、「ユーザ」が誤って「鳴らす（ユーザ, ブザー, 3 回）」の主体語とされてしまった。

本変換アルゴリズムでは、構文解析の結果を元にルールを構築するため、構文解析自体が誤っている場合、提案する変換アルゴリズムでは回避ができない。本評価で用いた 98 文の仕様文では、14 文に構文解析結果の誤りが存在した。構文解析結果の誤りのある文を除くと、命題プリミティブの抽出については Precision 0.80、Recall 0.82、論理関係の同定については Precision 0.82、Recall 0.86 でセミ形式記述への変換が実現された。構文解析の失敗による変換誤りについては変換結果のレビュー時に手動で修正する。

以上、提案するアルゴリズムにより、従来では暗黙的に行われていた自然言語仕様書から入力項目・確認項目の抽出、及び論理関係の同定作業をルール化し、明示化可能なことを示した。

3.7 欠陥の検出・指摘とテストケース生成のケーススタディ

3.7.1 方法

本論文で提案するテストケース設計プロセスとツールを用いて、欠陥を取り除いたテストケースが導出できることを確認する実験を行った。

実験では、現場の技術者から構成される 2 チームに、提案するテストケース設計プロセスに沿って、次のような手順でテストケース設計をさせた。

1. 実験の概要説明（15 分）
2. 実験対象のシステム仕様書表 3.7 の説明（15 分）
3. セミ形式化した仕様書表 3.8 の欠落部分の補完と仕様書のレビュー（20 分）
4. 命題ネットワーク化した仕様書図 3.14 の欠落部分（記述漏れ・論理関係）の補完と仕様書のレビュー（20 分）

現場でのテストケース設計におけるレビューを想定し、実験に参加するチームは経験の豊富な技術者と経験の浅い技術者で構成した。実験参加者のプロファイルを表 3.6 に示す。

テストケース設計プロセスの入力となる仕様書には、実験参加者の製品ドメインの知識の多寡によりレビューでの指摘事項が増減することを防ぐために、実験参加者の業務とは異なる製品ドメインの仕様書を採用した。本実験では、曖昧さを含む仕様書として公開されている話題沸騰ポット第 3 版 [48] を元に作成した、表 3.7 に示す 9 つの要求仕様の文を用いた。

また、修正結果の正答の仕様には、話題沸騰ポット第 3 版の曖昧さを修正したものとして公開さ

表 3.6　実験参加者プロファイル

		経験年数		
実験参加者名		総業務経験	設計・開発	テスト
Team A	A	3	3	0
	B	5	2	3
	C	19	1	15
	D	30	22	5
Team B	E	0.5	0.5	0
	F	3	0	3
	G	3	0	3
	H	3.5	3	0.5
	I	7	7	0
	J	15	15	0

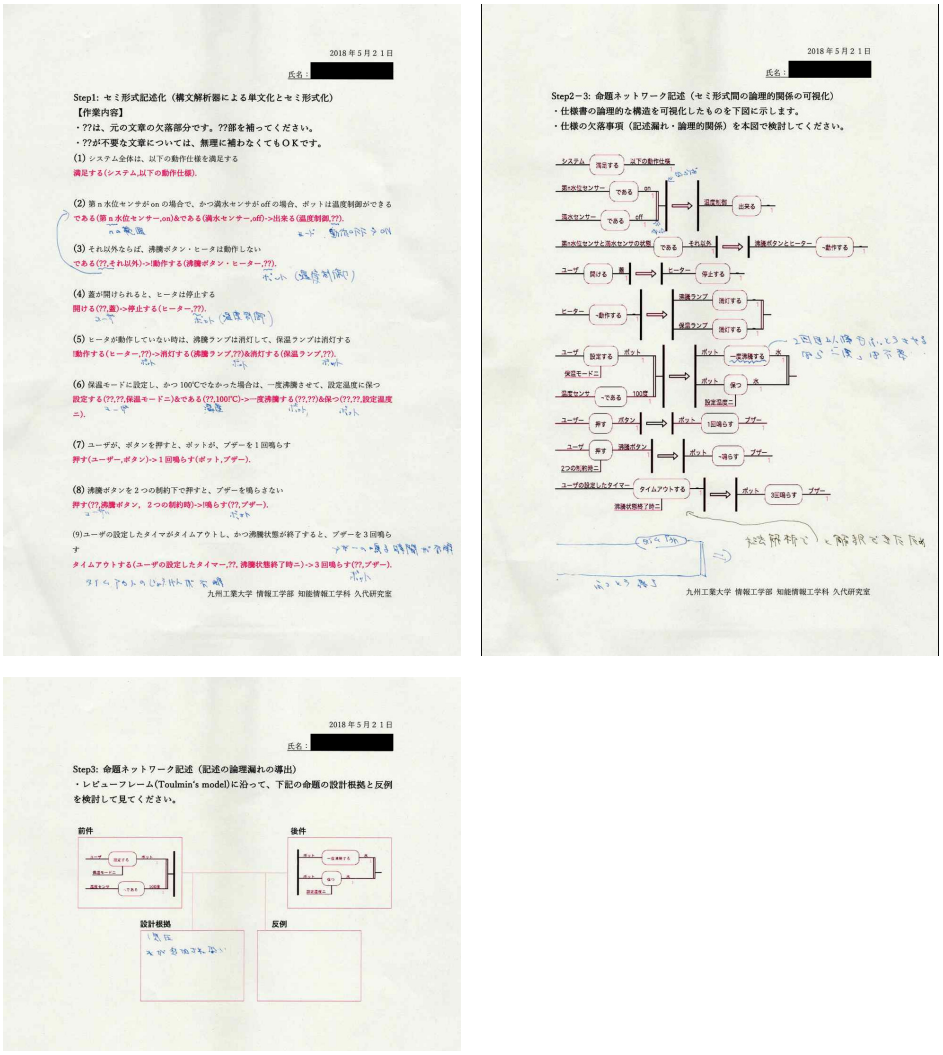
表 3.7　実験用の要求仕様書

No.	仕様文
1	システム全体は、以下の動作仕様を満足する。
2	第 n 水位センサが on の場合で、かつ満水センサが off の場合、ポットは、温度制御ができる。
3	それ以外ならば、沸騰ボタン・ヒータは動作しない。
4	蓋が開けられると、ヒータは停止します。
5	ヒータが動作していない時は、沸騰ランプは消灯して、保温ランプは消灯します。
6	保温モードに設定し、かつ 100℃でなかった場合は、一度沸騰させて、設定温度に保つ。
7a	ユーザが、ボタンを押すと、ユーザが、ブザーを 1 回鳴らします。
7b	ユーザが、ボタンを押すと、ポットが、ブザーを 1 回鳴らします。
8	沸騰ボタンを 2 つの制約下で押すと、ブザーを鳴らさない。
9	ユーザの設定したタイマがタイムアウトし、かつ沸騰状態が終了すると、ブザーを 3 回鳴らす。

Note: Team A は No. 7a、Team B は No. 7b で実験した。

れている話題沸騰ポット第 7 版の記述を元に、図 3.15 の中央部分に示すものを設定した。

3.7.2 結果と考察



Note: 氏名の部分は筆者が黒塗りで加工した。

図 3.13 実験参加者の回答シート

実験の結果、実験参加者それぞれ図 3.13 のようなレビュー結果を得た。以降、本テストケース設計プロセスの各ステップの結果と考察を述べる。

3.7.2.1 セミ形式記述による記述漏れレビュー支援

表 3.9 は、実験参加者によって作成された表 3.8 の記述漏れを補完した内容である。

セミ形式記述による記述漏れの補完は、表 3.9 の下線部で示すとおり、いずれのチームからも妥当な補完内容の指摘を得ることができた。表 3.9 には誤った補完内容も含まれるものの、レビュー

表 3.8　表 3.7 のセミ形式化された仕様

No.	セミ形式化された仕様文
1	満足する（システム，以下の動作仕様）.
2	である（第 n 水位センサー，on）& である（満水センサー，off）–> 出来る（温度制御，①）.
3	である（①，それ以外）–> !動作する（沸騰ボタン・ヒーター，②）.
4	開ける（①，蓋）–> 停止する（ヒーター，②）.
5	!動作する（ヒーター，①）–> 消灯する（沸騰ランプ，②）& 消灯する（保温ランプ，③）.
6	設定する（①，②，保温モードニ）& である（③，100!℃）–> 一度沸騰する（④，⑤）& 保つ（⑥，⑦，設定温度ニ）.
7a	押す（ユーザ，ボタン）–> 1 回鳴らす（ユーザ，ブザー）.
7b	押す（ユーザ，ボタン）–> 1 回鳴らす（ポット，ブザー）.
8	押す（①，沸騰ボタン，2 つの制約時ニ）–> !鳴らす（②，ブザー）.
9	タイムアウトする（ユーザの設定したタイマー，①，沸騰状態終了時ニ）–> 3 回鳴らす（②，ブザー）.

Note:　記述漏れの候補を参照の便宜のために丸数字で置き換えた。実験参加者には丸数字の代わりに「??」と書かれた仕様文についてレビューを実施させた。また、本ケーススタディにおける「である」は、3.3.1 小節で述べた「is」に相当し、かつ「である」の対象語は、主体語が取りうる値（3.3.1 小節の「is」の制約語）に相当する。

表 3.9　表 3.8 の補完内容

	補完内容		
補完箇所	模範解答	Team A	Team B
No. 2 ①	補完候補なし	補完候補なし、ポット内、水温、水、on	補完候補なし、動作 off→on、沸騰、保温
No. 3 ①	第 n 水位センサと満水センサの状態	水位センサと満水センサ、(2) の前件否定、第 n 水位センサ on & 満水センサ off、ユーザ	(2) の前件、第 n 水位センサ、満水センサ、温度制御
No. 3 ②	補完候補なし	補完候補なし、off	補完候補なし、ポット（温度制御）、off
No. 4 ①	ユーザ	ユーザ、人（外部）、ポット	ユーザ、ポット、補完候補なし
No. 4 ②	温度制御	温度制御、動作、off、補完候補なし	ポット（温度制御）、動作、沸騰モード、保温モード、off、補完候補なし
No. 5 ①	補完候補なし	補完候補なし、システム、off	補完候補なし、ポット、沸騰モード、保温モード、off
No. 5 ②	補完候補なし	補完候補なし、システム、off	補完候補なし、ポット、off
No. 5 ③	補完候補なし	補完候補なし、システム、off	補完候補なし、ポット、off
No. 6 ①	ユーザ	ユーザ、人	ユーザ、温度、補完候補なし
No. 6 ②	ポット	ポット、システム、補完候補なし	ポット、ポット内の水、補完候補なし
No. 6 ③	水温	水温、温度センサ?、システム	水の温度、水温度センサ、ポット内の水、水、どの温度?
No. 6 ④	ポット	ポット、システム、ヒーター、水	ポット、ヒーター、補完候補なし
No. 6 ⑤	水	水、ヒーター、on	水、ポット内の水、沸騰モード、on、補完候補なし
No. 6 ⑥	ポット	ポット、システム、補完候補なし	ポット、ヒーター、補完候補なし
No. 6 ⑦	水	水、温度制御、水温	水、保温モード、補完候補なし
No. 8 ①	ユーザ	ユーザ、システム	ユーザ、補完候補なし
No. 8 ②	ポット	ポット、システム	ポット、補完候補なし
No. 9 ①	補完候補なし	補完候補なし、システム?、on	補完候補なし、カウント満了
No. 9 ②	ポット	ポット、システム	ポット、補完候補なし

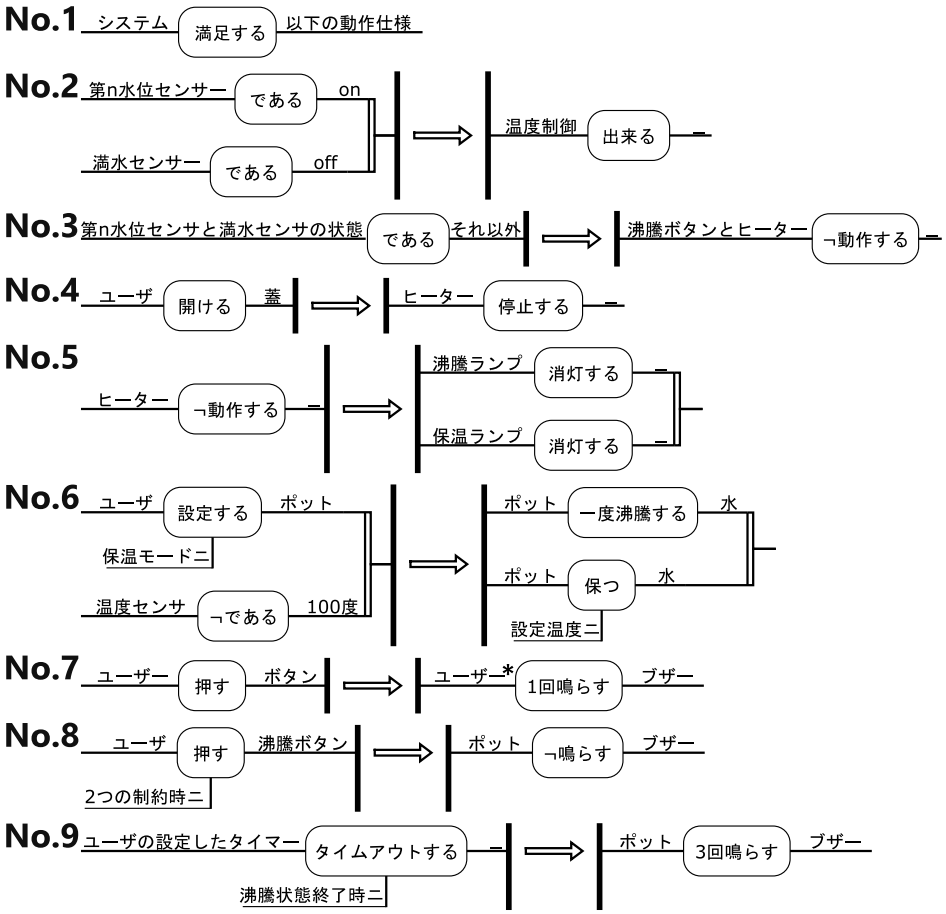
Note:　模範解答は、話題沸騰ポット第 7 版の仕様書を元に筆者らが正答として補完したものである。下線部は模範解答に照らして妥当な補完内容を示す。

は複数人で実施されること、及びチームとしては正答を導くことができたことを踏まえると、本セミ形式記述上でのレビューにより、記述漏れ（主体・対象漏れ）の補完が可能であると考える。

また、Team A からは主体・対象の欠落箇所（「??」箇所）に加えて、記載された語に対する誤りの指摘として、表 3.8 の No. 7a の後件（「1 回鳴らす（**ユーザ**，ブザー）」）の「ユーザ」を、「ポット」か「システム」への修正をすべきという指摘も得られた。本指摘は妥当なものであり、記載済みの箇所についても誤りが指摘されることを確認した。Team B では表 3.8 の No. 7b のように是正版の仕様書で実験したため、記載済みの記述に対する指摘はなかった。

以上により、セミ形式記述の表現上でのレビューにて記述漏れ（主体・対象漏れ）が修正されることを示した。

3.7.2.2 命題ネットワークによる論理関係のレビュー支援



∗: Team Bでは、〝ポット〞と記載されたもので実験した。

図 3.14　実験対象の命題ネットワーク

命題ネットワークを用いたレビューの実験では、表 3.8 の記述漏れを表 3.9 の模範解答で補完した図 3.14 を用いた。

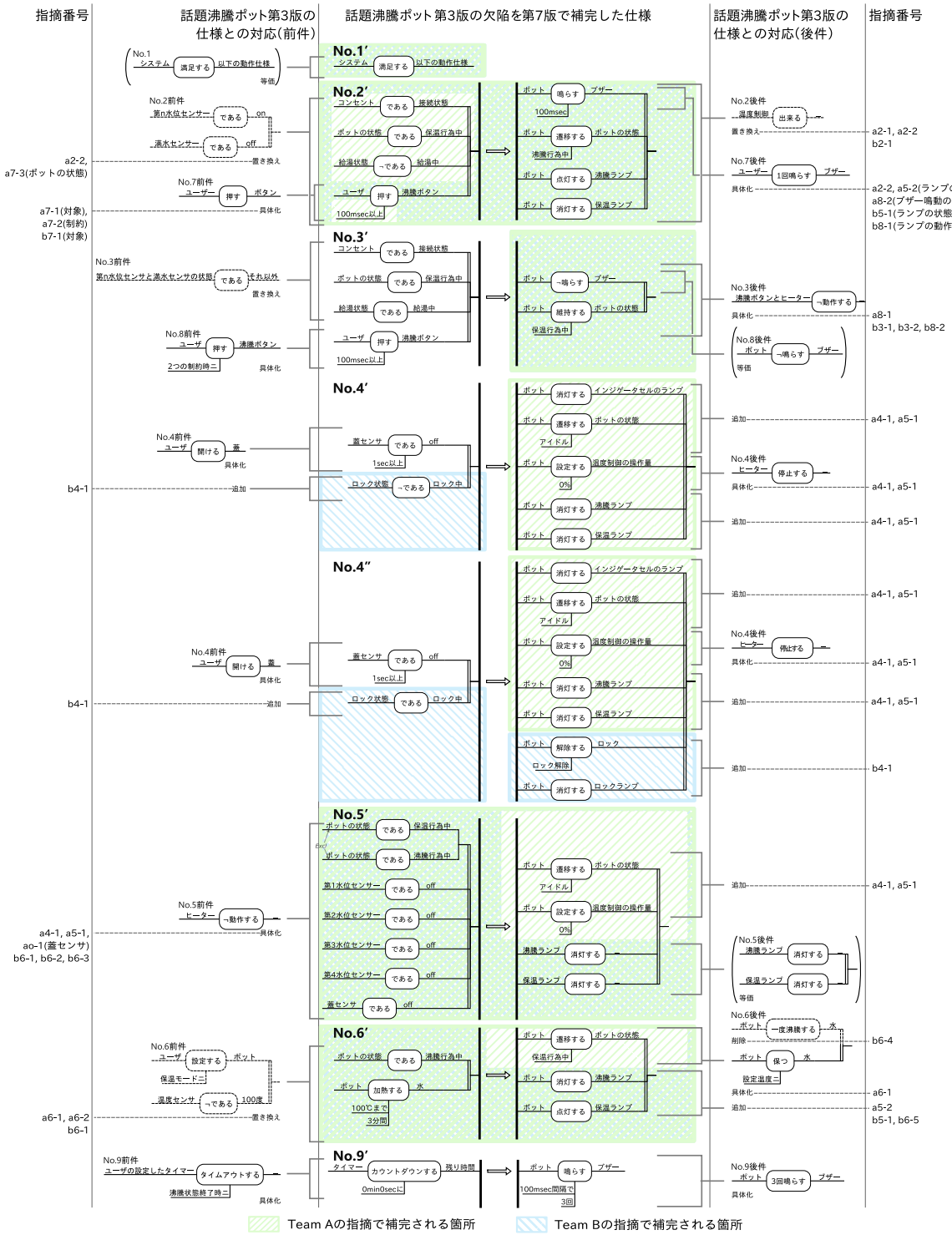


図 3.15　期待する修正版命題ネットワーク

表 3.10　図 3.14 の指摘内容

	指摘内容	
指摘箇所	Team A	Team B
No. 1	-	
No. 2	温度制御とは沸騰ボタンとヒーターが動作することなのか?(a2-1)、記載事項が不明である(a2-2)、満水時は温度制御できないのか?、水位 off かつ満水 on はありえるのか?、どの n 水位か?、「それ以外」は任意の 1 つでよいのか?、「出来る」に「ふた閉」という制約を追加せよ、No. 3 との間に Exclusive 制約を追加せよ	温度制御とは具体的にどういうことか?(b2-1)、No. 3 との間に One 制約を追加せよ
No. 3	ランプはどうなっているのか?、温度制御とは沸騰ボタンとヒーターが動作することなのか?、No. 2 との間に Exclusive 制約を追加せよ	沸騰ボタンが動作しないとはどういうことか?(b3-1)、保温はするのか?(b3-2)、温度制御はヒータがするのか?、「フタの状態開ける」を追加せよ、No. 2 との間に One 制約を追加せよ
No. 4	停止=動作しない?(a4-1)、	水を出すボタンのロックは?(b4-1)、ユーザが蓋を締めたらヒータは停止したままか、あるいは動作を始めるのか?
No. 5	停止=動作しない?(a5-1)ランプの点灯条件(a5-2)、	ヒータ動作中のランプ状態は?(b5-1)
No. 6	保温モードへの遷移方法はどのようか?(a6-1)、沸騰条件は何か（100℃に到達、100℃で何秒経過 など）?(a6-2)、ポットのモードは保温のみか?、正しくは設定温度が 100℃なのではないか?、制御方式はヒータしかないのか、どのように保つのか?、保つのは何分か?、	ヒーターを動作させる条件は?(b6-1) 、水位 or 満水センサの情報が必要なのではないか?(b6-2)、水がなくても一度沸騰するのか?(b6-3)、2回目以降も沸騰させるなら「一度」は不要では ないか?(b6-4)、ランプはどのようになる?(b6-5)、ヒータはどのように動作する?、保温モードに設定されるかつ 100 度であるときの動作は?、いつまで保つのか?
No. 7	どのボタンか?(a7-1)、ボタンを続けて押しているときは?(a7-2)、沸かし中には?(a7-3) 、沸騰を開始しないときはブザーは鳴らなくていいのでは?	何のボタンか?(b7-1)
No. 8	ブザー以外の動作が見えない(a8-1)、ブザーを鳴らす際の制約は?(a8-2)、ボタンのブザー鳴動中は?、	沸騰ボタン押下時のランプの動作は?(b8-1)、沸騰自体はスタートするのか?(b8-2)
No. 9	ブザー鳴動中のタイムアウトではどちらが優先か?、ヒータが動作しないを追加せよ、役割がわからない	「沸騰終了」を And で追加せよ
その他	蓋が開いていても温度制御できる?(ao-1)、ボタンの操作条件などは何か、安全対策は?、水は何か?、気圧はいくらか?	-

Note: 下線部は、欠陥のある仕様図 3.14 から修正済みの仕様図 3.15 を導く妥当な記述漏れ・論理関係の誤りの指摘である。下線部直後の番号は図 3.15 の指摘番号と対応する。記号「-」は指摘なしの意である。

図 3.14 に対する実験参加者による記述漏れ、論理関係の誤りの指摘事項は表 3.10 である。表 3.10 の指摘事項を修正すると、話題沸騰ポット第 7 版を元に設定した正答の仕様のうち、図 3.15 の網掛け部分が導かれる。

次に図 3.15 の網掛けを導く表 3.10 の指摘事項の例を挙げる。

1. 曖昧な対象: 図 3.15 No. 2' の「沸騰ボタン」を導く指摘
   表 3.10 の a7-1（「どのボタンか?」）、b7-1（「何のボタンか?」）という指摘に回答するためには、No. 7 の前件にて、ユーザが押す対象「ボタン」を具体的に示す必要がある。この指摘に対する回答「沸騰ボタン」が元の記述「ボタン」の曖昧さを取り除く。
2. 条件の漏れ: 図 3.15 No. 5' の前件を導く指摘
   表 3.10 の Team A の指摘 a5-1（「停止=動作しない?」）に対する回答は否であり、動作しないときの具体的な条件を示すことで、No. 5' の条件が補完される。Team B の指摘 b6-1（「ヒーターを動作させる条件は?」）で得られる条件を否定することで No. 5' の条件が得られる。また、Team B の指摘 b6-2（「水位 or 満水センサの情報が必要なのではないか?」）、b6-3（「水がなくても一度沸騰するのか?」）への回答から、No. 5' の条件には水位センサの条件が補完される。
3. 動作の漏れ: 図 3.15 No. 6' の後件を導く指摘
   ランプの点灯・消灯は、ポットの状態遷移に伴って起こる。Team A の指摘 a5-2（「ランプの点灯条件」）、Team B の指摘 b5-1（「ヒータ動作中のランプ状態は?」）、b6-5（「ランプはどのようになる?」）への回答として、このポットの状態遷移に伴って起こることが判明する。No. 6' は「アイドル」状態へ遷移するので、この遷移に伴うランプの点灯/消灯状態変化が補完できる。

また、Team A の指摘 a6-1（「保温モードへの遷移方法はどのようか?」）へ回答として No. 6’ の前件を求めると、元の仕様図 3.14 No. 6 の条件は誤り――ユーザの操作でポットを保温モードに設定する機能はない――であることが判明する。正しくは、保温モードは、ポットにより自動的に遷移する状態の一つであり、この動作を No. 6’ の後件に置くと、No. 6’ の動作「遷移する（ポット，ポットの状態，保温行為中)」が補完される。

図 3.15 の網掛けに示すとおり、Team A、Team B のいずれも命題ネットワーク上のレビューで論理関係の誤りを指摘できた。

以上により、命題ネットワーク上でのレビューにて論理関係の誤り（条件・動作の漏れ）が修正されることを示した。

3.7.2.3 セミ形式記述からのテストケース生成

話題沸騰ポット第 7 版より作成した図 3.15 中央の仕様からデシジョンテーブルを自動生成すると、起こりうる真偽の組み合わせを漏れなく含む、次の 8 件のテストケースが得られた。[7)]。

- No. 2’–No. 4’、No. 4”、No. 6’、No. 9’ のテストケース:
  前件の And が真の場合がそれぞれ 1 件 （計: 6 件）
- No. 5’ のテストケース:
  前件の And が真の場合が 2 件（この 2 件中、前件の Or 条件の 2 つのオペランド（「である（ポットの状態，保温行為中)」と「である（ポットの状態，沸騰行為中)」）は Exclusive 制約があるため、どちらか一方のみ真となった。その他の前件の And のオペランドは 2 件中全て真であった。）

テストケース設計プロセスのステップ 1–3（図 3.1）によって自然言語のシステム仕様書からセミ形式記述へ、セミ形式記述からテストケースへ自動的に変換できることを確認した。

3.8 欠陥の検出・修正のケーススタディとテストケース生成

3.8.1 方法

表 3.11 評価ケーススタディ参加者のプロファイル

	経験年数
ID	テスト設計及びテスト実行
A	13
B	8
C	4

7) 機能の仕様を示さない図 3.15 No. 1’ を除く。また、動作条件が不成立時の動作について規定されていないため、動作条件が成立する場合のみ集計した。

3.7 節では、テスト設計者のみの参加により、提案するテストケース設計プロセスにしたがって、仕様書の欠陥を検出・指摘が可能なことを確認した。本ケーススタディでは、ケーススタディ参加者をテスト設計者・システム設計者が参加するレビューを想定し、提案するテストケース設計プロセスの成果物に対して、仕様書欠陥の指摘に加え、欠陥の修正作業まで可能なことを確認する。

本ケーススタディに当たり、表 3.11 のプロファイルを持つテスト技術者 3 名からなるグループに、テストケース設計プロセスにしたがってテストケース設計を実施してもらった。

表 3.12　ケーススタディで用いた仕様文

No.	仕様文
1	第 n 水位センサが on で、かつ満水センサが off の場合、温度制御が可能になります。
2	それ以外の場合は、沸騰ボタン・ヒータは動作しません。
3	蓋が開けられると、ヒータは停止します。
4	沸騰ボタンは動作しません。
5	ヒータが動作していないときは、沸騰ランプ及び保温ランプは消灯します。
6	保温モードに設定した際、100℃でなかった場合は、必ず一度沸騰させた後、自然に冷やしながら設定温度に保つ動作をします。
7	タイマは最大 1 時間まで設定できます。
8	ユーザからボタン（タイマ・保温設定・沸騰・解除・給湯の 5 つ）が押された時、ブザーを 1 回鳴らします。
9	しかし、上記 2 つの制約時には、沸騰ボタンが押されてもブザーを鳴らさないこととします。
10	ユーザが設定したタイマのタイムアウト時、及び沸騰状態終了時には、ブザーを 3 回鳴らします。

表 3.13　セミ形式化した仕様文

No.	セミ形式の仕様文
1	is（第 n 水位センサ，?，on）& is（満水センサ，?，off）–> なる（温度制御，?，可能に）.
2	is（?，?，それ以外）–> !動作する（沸騰ボタン，?）& !動作する（ヒータ，?）.
3	開ける（?，蓋）–> 停止する（ヒータ，?）.
4	!動作する（沸騰ボタン，?）.
5	!動作している（ヒータ，?）–> 消灯する（?，沸騰ランプ）& 消灯する（?，保温ランプ）.
6	設定する（?，?，保温モードに）& !is（?，?，100℃）–> 沸騰する（?，?，必ず一度）& 保つ（?，?，設定温度に）.
7	設定できる（タイマ，?，最大 1 時間まで）.
8	押す（ユーザ，沸騰ボタン）–> 鳴らす（?，ブザー，1 回）.
9	is（?，?，上記 2 つの制約時に）& 押す（?，ボタン “タイマ・保温設定・沸騰・解除・給湯の 5 つ”）–> !鳴らす（?，ブザー）.
10	タイムアウトする（ユーザが設定したタイマ，?）& is（?，?，沸騰状態終了時）–> 鳴らす（?，ブザー，3 回）.

ケーススタディ対象の仕様には 3.7 節と同様に、ケーススタディ参加者の製品ドメインの知識の多寡によりレビューでの指摘事項が増減することを防ぐために、ケーススタディ参加者の業務とは異なる製品ドメインの仕様書を採用した。本ケーススタディでは、曖昧さを含む仕様書として公開されている話題沸騰ポット第 3 版 [48] から表 3.12 に示す 10 文の仕様文を用いた。

修正内容の検証するにはシステム設計者の参加が必須であるが、本ケーススタディでは、ケーススタディ参加者がテスト設計者・システム設計者を兼任するとみなして代替し、本テストケース設計プロセスの成果物上で、欠陥の指摘・修正が可能か、修正結果からテストケースが生成されるかに着目してケーススタディを実施した。

3.8.2 ケーススタディ結果と考察

表 3.14 レビューにより省略語が補われたセミ形式記述

No.	セミ形式の仕様文
1	is（第 n 水位センサ，_，on）& is（満水センサ，_，off）–> なる（温度制御，_，可能に）.
2	is（水位，_，それ以外）–> !動作する（沸騰ボタン，ブザー）& !動作する（ヒータ，ランプ）.
3	開ける（ユーザ，蓋）–> 停止する（ヒータ，ランプ）.
4	!動作する（沸騰ボタン，ランプ）.
5	!動作している（ヒータ，ランプ）–> 消灯する（*，沸騰ランプ）& 消灯する（*，保温ランプ）.
6	設定する（ユーザ，*，保温モードに）& !is（温度センサ，_，100℃）–> 沸騰する（ヒータ，水，必ず一度）& 保つ（ヒータ，水温，設定温度に）.
7	設定できる（タイマ，_，最大 1 時間まで）.
8	押す（ユーザ，ボタン “タイマ・保温設定・沸騰・解除・給湯の 5 つ”）–> 鳴らす（*，ブザー，1 回）.
9	is（*，_，上記 2 つの制約時に）& 押す（ユーザ，沸騰ボタン）–> !鳴らす（*，ブザー）.
10	タイムアウトする（ユーザが設定したタイマ，_）& is（*，_，沸騰状態終了時）–> 鳴らす（*，ブザー，3 回）.

上記「*」の記号は、「_」という回答を、本文中での参照のために置き換えたもの。

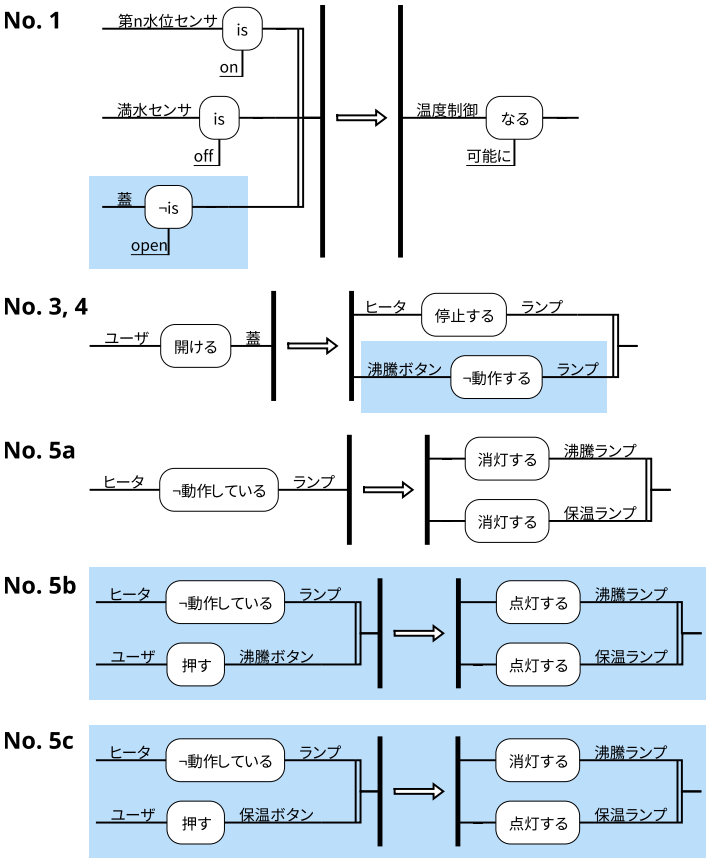
ケーススタディ参加者によるセミ形式記述の修正の結果は表 3.14 である。ケーススタディ参加者による補完内容と [48] から曖昧さを取り除いた仕様書として公開されている第 7 版 [48] の仕様書と比べると、表 3.14 中の下線部分が不適当な補完内容となっていた。下線部分のうち、「*」部分は、システムに相当する語であり、ケーススタディ参加者は、自明であるとして意図的に補完を行わなかった。

上記レビューの結果、参加者は「?」で示された 31 個の補完候補を全てレビューし、補完作業を行うことができた。ここから、ケーススタディ参加者は、命題プリミティブの主体・対象・制約が表現する内容を理解し、修正作業が可能なことを確認した。

表 3.14 の内容を命題ネットワークとして可視化し、論理関係の誤りを修正してもらった。レビューの結果、表 3.14 中の 22 個ある論理演算子について、31 個の論理演算子の追加・削除の修正が実施された。図 3.16 に修正結果の一部を示す。

網掛け部分に示す部分がレビューにて補完された箇所である。図 3.16 の No. 1 は、動作条件の不足を指摘・補完し、No. 3、4 では、動作条件の記述されていない表 3.14 の No. 4 に適当な動作条件を議論した結果、No. 3 と同等であると判断され、統合されたものである。No. 5a–No. 5b は、No. 5（表 3.14 の No. 5 に相当）の確認項目に記載の沸騰ランプと保温ランプを点灯するための動作がないことから追加された仕様である。No. 5b は沸騰ランプと保温ランプを点灯するための仕様として追加されたものであり、No. 5c は No. 5b とは異なるボタンを押したときのランプの点灯動作を表す仕様である。上記から、命題ネットワークを使って動作条件・確認項目の論理関係がレビューでき、誤りを指摘・修正可能なことが確認された。

レビュー前後のセミ形式記述からデシジョンテーブルを生成すると、表 3.15、表 3.16 に示すものが得られた。表 3.16 は命題ネットワーク上でのレビューによる修正を受けて、動作条件「is（蓋, _，open)」を含めたセミ形式記述から自動的に生成された。ここから、セミ形式記述上の動作条



※網掛け部分はレビューにより補完された箇所

図 3.16　ケーススタディ参加者により修正された命題ネットワーク

表 3.15　セミ形式記述のレビュー結果（表 3.14）の No. 1 から生成したデシジョンテーブル

		テストケース		
		前件真	前件偽	
	命題	1	2	3
条件	is（第 n 水位センサ，_，on）	T	F	T
	is（満水センサ，_，off）	T	-	F
動作	なる（温度制御，_，可能に）	T	-	-

表 3.16　命題ネットワークのレビュー結果（図 3.16）の No. 1 から生成したデシジョンテーブル

		テストケース			
		前件真	前件偽		
	命題	1	2	3	4
条件	is（第 n 水位センサ，_，on）	T	F	T	T
	is（満水センサ，_，off）	T	-	F	T
	is（蓋，_，open）	F	-	-	T
動作	なる（温度制御，_，可能に）	T	-	-	-

件・確認項目の修正結果から、人手によるミスなく網羅的な条件組み合わせのテストケース生成が実現されることを確認した。

以上、提案するテストケース設計プロセスにより、仕様書の欠陥の検出・修正と、テストケースの自動生成が実現されていることを確認した。

3.9 本章のまとめ

本章では、プロダクトライン開発におけるテストケース設計における課題解決に取り組んだ。

日本を代表する企業 2 社の技術者複数名へのヒアリングと観察の結果、従来のテストケース設計プロセスでは、次の 2 つの課題があることがわかった

1. テストケース設計工数が大きい点: 自然言語で書かれたシステム仕様書の欠陥の修正結果とシステム仕様書からテストケースへの変換結果がテストケースのレビューで合わせて検証されており、このレビューで修正・補完の誤りや仕様書の欠陥が発見された場合、テストケース設計への手戻りが発生し、テストケース設計の工数を大きくしていた。
2. テストケースのレビューで誤りを指摘・改善促すことが難しい点: システム仕様書の欠陥の修正とテストケースへの変換作業は各テスト設計者の頭の中で暗黙的なプロセスとして手作業で実施されており、テストケース中には、テスト設計者がどのような根拠でそのテストケースを出力したのかが現れないため、レビューによる誤りの修正が難しかった。

上記課題を解決するために、本研究では、次を実施した。

1. 複数企業へのヒアリング結果に基づき、システム仕様書からテストケース設計プロセスを再定義し、プロセス中のステップをアルゴリズム化してテストケース設計の属人性を削減
2. システム仕様書からテストの入力項目・確認項目の抽出と言った、定型的な作業の自動化してテストケース設計工数を削減
3. テストケースへの変換の中間成果物を用いてシステム仕様を可視化し、システム仕様書の欠陥のレビューを容易化

提案するテストケース設計プロセスでは、テストケースへの変換に先立ち、予めシステム仕様書中に含まれる欠陥の修正を実施することで、テストケース設計への手戻りの工数の削減を図る。また、アルゴリズムによる変換作業の自動化により、これまでシステム仕様書から入力項目・確認項目に当たる文を抜き出す作業に要していた新規製品の 520 ページ程度の仕様書を用いた全 700 時間テストケース設計のおよそ 26%、派生製品の 640 ページ程度の仕様書を用いた全 255 時間テストケース設計のおよそ 31%もの工数の削減を見込む。

また、提案するテストケース設計プロセスでは、システム仕様書からテストケースへの変換ステップがアルゴリズムとして実現されているため、システム仕様書の欠陥であることが判明すれば、変換されたテストケースに誤りを、システム仕様書の誤りなのか、変換ルールの誤りなのかを明らかにしてレビューできる。これにより、従来のテストケースのレビューの困難さを解決した。

提案するテストケース設計プロセスのケーススタディでは、システム仕様書から機能の入力項目・確認項目を抽出する作業が、ルールベースのアルゴリズムとして自動化されることを示した。また、テストケース設計プロセスのケーススタディでは、可視化されたシステム仕様について、欠陥の修正が実現され、システム仕様書からテストケースへの変換が実現されることを確認した。

以上結果から、自然言語仕様書からのテストケース設計工数を削減するテストケース設計プロセスが実現できたと考える。

第 4 章　モデル検査技術と実行テストを融合したテストケース実行環境

プロダクトライン開発では、組合わせにより多品種の製品が実現されている。派生製品開発の度に、Domain Artifact と Application Artifact の組合わせにおいて既存機能に不具合が起こらないことを確認するテストを実施する。近年の組込み機器には、ネットワークを介した通信機能を搭載することも多いため、メッセージによって非決定的な順序で起動される機能をテストするために、組み合わせた機能をさらに複数通りの実行順序で実行し、テストを実施する。従来のプロダクトライン開発では、派生製品開発の度に複数機能を組み合わせたテストケースの設計を実施し、しばしば手動でテストを実施するためテスト実行に工数を要する。

本章では、上記テスト工数削減のために、複数機能を決定的・非決定的な順序で実行するテストを、自動的かつ網羅的にテストする統合テスト環境 [49–51] を開発した。

4.1　統合テスト環境概要

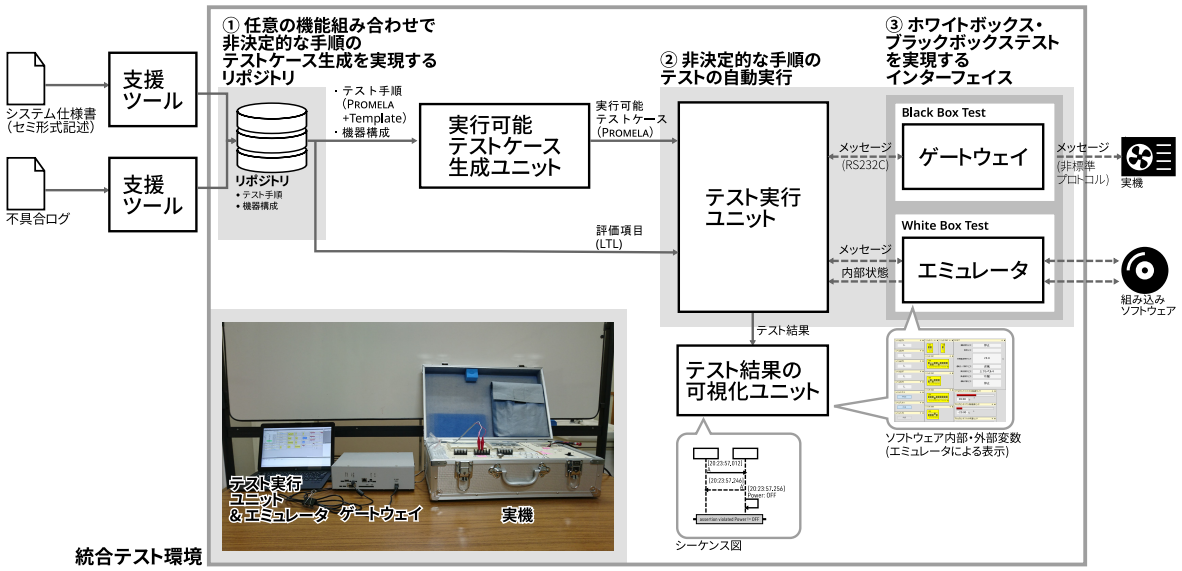


図 4.1　統合テスト環境構成

統合テスト環境の構成を図 4.1 に示す。統合テスト環境は、下記で構成される。

1. 任意の機能組み合わせで決定・非決定的なテストケースの生成を可能とするテストケースの部品化（図 4.1 の ①）
2. 非決定的な機能の実行順序の網羅的な実行と複数の評価項目の自動評価（図 4.1 の ②）

3. 組込みソフトウェアの自動評価を実現するインターフェイス（図 4.1 の ③）

1. は、任意の機能の組み合わせでテストを実施可能にするために、機能ごとのテストケースを部品化し、部品化したテストケースを任意に組み合わせて機能の決定・非決定的な実行順序のテストケースを生成する。テストケースの部品ははシステム仕様書と不具合事例を 2 種類の入力を元に作成する。システム仕様書を入力とするテストケースは機能ごと記述する。これにより、任意の機能の組み合わせのテストケース生成を可能にする。評価項目には各機能の期待値にあたる機器の状態や、機器から送信されるべきメッセージを書く。

不具合事例を入力とするテストケースは、不具合を再現するための操作手順をテストケースとして記述し、評価項目には不具合が発生しないことを示す機器の状態や、機器から送信されるべきメッセージ（応答メッセージなど）を書く。

2. は、前段の 1. のテストケースを入力に、モデル検査器 Spin [52] をテストの実行エンジンとして組込みソフトウェアを自動評価する。ネトとワークを介した通信機能を有する組込み機器では、メッセージ受信順序が非決定的なため、メッセージによって起動される機能は、複数の動作順序が起こり得る。2. では、Spin の機能により、非決定的なテストケースを含むテストケースを起こりうる全ての順序で自動評価する。

3. は、Spin から組込みソフトウェアを評価するためのインターフェイスである。ゲートウェイは、実機–Spin 間の通信を実現し、メッセージ交換を介したブラックボックステストを可能にする。エミュレータ [53] は、組込みソフトウェアの内部変数を操作・監視する機能を提供することで、ソフトウェア内部変数についてのホワイトボックステストを可能にする。

4.2 テストケースの部品化と決定・非決定的なテストの自動実行・評価

本研究では、モデル検査器 Spin [52] 用の記述言語を用いてテストケースの記述を行う。Spin では、テスト手順と評価項目の組でテストケースを構成し、Promela という言語でテスト手順を、Linear Temporal Logic (LTL) [54] という論理式で評価項目を記述する。

Promela 上では機能ごとのテスト手順を `proctype`（プロセスを定義する構文）を用いて記述する。Spin では `proctype` 間の実行順序は非決定的であるとして扱われ、複数の `proctype` が存在する時、`proctype` ごとのテストケースを起こりうる全ての順序で自動的にテストを実行する。

この仕組を用いることで、機能ごとに記述した Promela のテスト手順を部品として扱い、複数の機能の Promela のテスト手順を組み合わせて自動的に決定・非決定的なテスト手順を生成できる。具体的には、機能ごとの Promela テスト手順に含まれる `proctype` 中の記述を、一つの `proctype` に統合した Promela テスト手順を生成することで決定的なテスト手順を生成できる。また、各機能のテスト手順中の `proctype` ごと一つの Promela ファイルに統合することで非決定的なテスト手順を実現できる。

テストの評価項目を記述する LTL は、命題論理に `always`（(テスト実行中）常に。記号: □）や `eventually`（(テスト実行中の）いつか。記号: ◇)、`until`（(テスト実行中のある時点）まで。記

号:U）といった時系列的な制約を記述する modal operator を導入した論理である。modal operator を用いることで、テスト実行中全体に渡る評価項目、例えば「常にエラー状態 S にならない」は、「$\Box!S$」のように書け、ある時点に対する評価項目、例えば「いつか期待状態 S になる」は、「$\Diamond S$」のように書ける。また、順序を伴った評価項目、例えば「いつかある状態 S_a になったら、いずれ状態「S_b」になる」は、「$\Diamond S_a$–> $\Diamond S_b$」のように記述できる。

Spin は、上記 LTL で記述された評価項目を Promela のテスト手順と合わせて受け取ると、起こりうる全ての実行順序でテストを実行する過程で、評価項目が成り立つことを自動的に評価できる。また、LTL 式で書いた評価項目は論理式であるため、任意の評価項目を And で結合することで、結合された複数の評価項目を同時に評価できる。

この仕組みを用い、機能ごとの評価項目を And で結合することで、複数の機能を組み合わせたテストケース上で、各々の機能を同時に評価できる。

本研究では、上記テスト手順・評価項目の記述の仕組みと自動的なテスト実行の仕組みを用いることで、機能ごとにテストケースを独立して記述・部品化と、任意の機能の組み合わせで決定・非決定的なテストの自動実行・評価を実現する。

4.3 モデル検査技術を用いた実行テストを実現するインターフェイス

本項では、Spin を用いた組込みソフトウェアの実行テストを実現するためのインターフェイスについて述べる。

本研究で開発したインターフェイスは、ゲートウェイとエミュレータからなる。ゲートウェイは、PC–実機間の通信プロトコルの違いを吸収し、実機間に流れるメッセージを PC で解釈可能にするものである。Spin からゲートウェイを介したメッセージ交換を行うことで、Spin から実機上の組込みソフトウェアの操作を実現した。

実機上の組込みソフトウェアの操作にはメッセージ送信・受信待ちの 2 種類を用いる。メッセージ送信・受信待ちのそれぞれのステップは、Promela が提供する構文 `c_code`（C 言語プログラムをテストステップとして埋め込む構文）を用いて記述する。テスト実行時に、埋め込まれた C 言語のテストステップを呼び出すことで、Spin から実機を操作する。テスト結果の判定は実機からの応答を元に行い、異常な応答を受信したときや期待する応答が受信されないときにテスト失敗をレポートする。テストケース（Promela）は、例えば次のように記述する。

```
1  proctype A(int from; int to) {
2    c_code{
3      send_cmd("msg",
4        PA->from, PA->to);
5      wait_for_reply("msg",
6        PA->to, PA->from, 1000);
7    }
8    /* snip */
9  }
```

3–4 行目にてメッセージ `msg` をアドレス `from` から `to` に向けて送信するステップ[1)]が記述されてい

1) 「`PA->`」は、`c_code` 内で `proctype A` の変数を参照する構文。

る。5–6 行目では msg の応答を、アドレス to から from へ送信されるのを 1000 ms 待機するステップを記述している。この待機処理により統合テスト環境のテストステップと実機上の組込みソフトウェア内の状態変化の同期をとる。待機の上限の 1000 ms を超過したとき、テストが失敗する。

上述までがメッセージ交換を介して実機上の組込みソフトウェアをブラックボックステストする機能である。

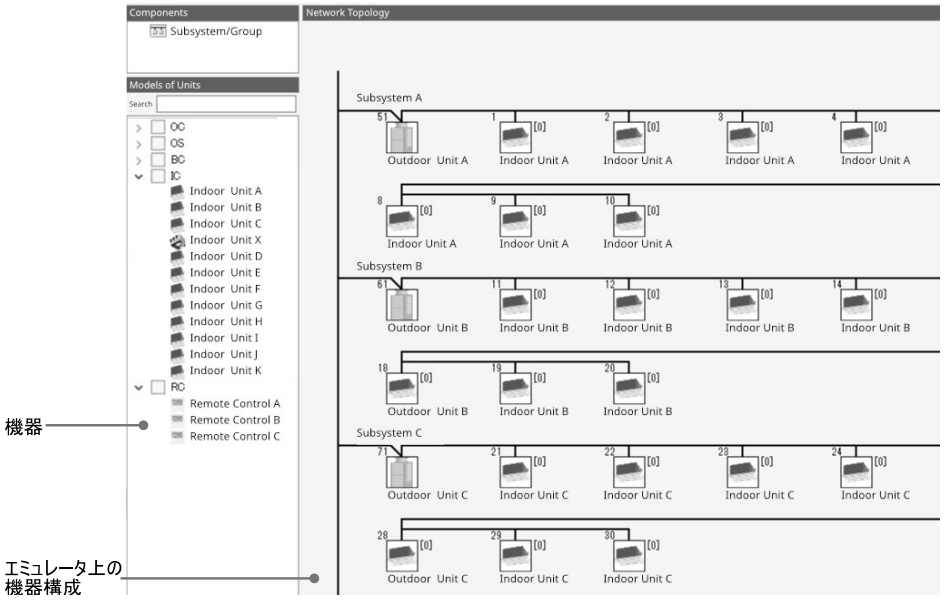


図 4.2　エミュレータ上の機器構成の例

本研究ではさらに、エミュレータ上の組込み機器ソフトウェアをテスト可能にすることで、機器のセットアップにかかる工数を削減、及び実機からは評価困難な組込みソフトウェアの内部変数に対する評価（ホワイトボックステスト）を実現する。エミュレータは、4.6.1 小節のビル用空調システムのハードウェア制約（CPU のクロック周波数など）やセンサなどの周辺装置の入力を模擬し、PC 上でビル用空調システムの組込みソフトウェアを動作させるツールである。エミュレータは、複数機器の組込みソフトウェアの同時に実行し、エミュレータ上だけ、あるいは実機とエミュレータ上の機器との組み合わせで空調システムを再現できる（図 4.2）。

本エミュレータは、エミュレータ上で動作する組込みソフトウェアのメモリやレジスタイメージから組込みソフトウェアの状態変数との対応付けを行う機能を有し、この機能を介して組込みソフトウェアの状態変数の内容を取得・操作できる。

本研究では、Spin からエミュレータを介して組込みソフトウェアの内部変数の操作・監視することで組込みソフトウェアを自動評価する。エミュレータを介して組込みソフトウェアの内部変数を評価することで、従来のハードウェアを用いたテストでは実施困難であったテスト——例えば LCD などの UI に表示されない状態や、メッセージから確認できない状態評価——が可能になる。

エミュレータ上の組込みソフトウェアの操作（メッセージの送信・受信待ち・状態値の設定・取得）の記述は、機上の組込みソフトウェアへの操作の記述と同様に、c_code により埋め込んだ C

言語関数により行う。テストケース（PROMELA）は、例えば次のように記述する。

```
1  proctype A(int addr) {
2    c_code{
3      set_state(PA->addr,"state","val");
4    }
5    /* snip */
6  }
```

3 行目でアドレス addr の機器の状態 state を値 val に設定するステップを記述している。

4.4　システム仕様書からのテストケース設計支援

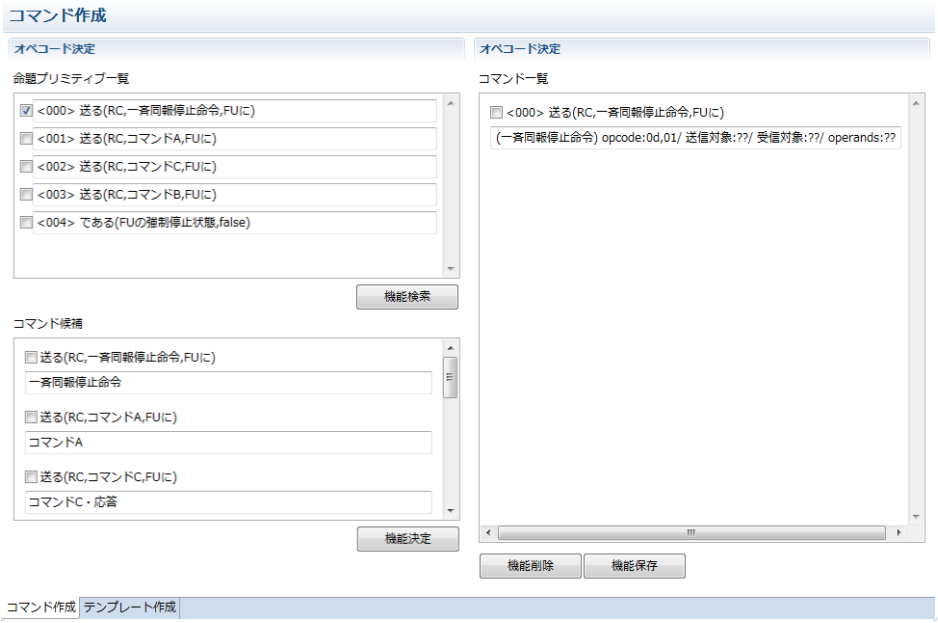


図 4.3　システム仕様書からのテストケース設計新ツール

本節では、システム仕様書からテストケースリポジトリにて資産化可能なテストケースの設計支援ツールについて述べる。システム仕様書からのテストケース設計では、第 3 章で述べたテストケース設計プロセスによりシステム仕様書から変換されたデシジョンテーブルを入力に図 4.3 のようなツールでインタラクティブにテストケースリポジトリへ入力可能な PROMELA 形式のテストケースを生成する。

図 4.3 を用いたテストケースの生成は次の手順にて行う。

1. 命題プリミティブから組込みソフトウェアを操作するためのコマンドへの変換
2. コマンドのパラメータの設定
3. テストケースの生成

1. は、図 4.3 の左上と左下の領域にて行う。左上の領域では、入力されたデシジョンテーブル中の命題プリミティブが列挙される。テスト設計者は、左上の領域から命題プリミティブを選択する

と、左下の領域に選択した命題プリミティブに対応する組込みソフトウェアを操作するためのコマンドの名称が一覧される。

コマンドの名称の一覧にあたり、命題プリミティブとコマンドとの対応付けは、5 つ組 (r, s, o, C, O) をひとつのレコードとして扱うデータベースとして実現されている（ただし、C、r、s、o はそれぞれ命題プリミティブの制約語のリスト、関係語、主体語、対象語、O はコマンドのリスト。$o \in O$ なる o は、コマンドの名称 n、コマンドのオペコード p からなる 2 つ組 (n, p) である）。オペコードの候補は、データベース上から、r、s、o、C の少なくともひとつが一致するレコード検索し、見つかったレコードの n を表示する。

本データベースのレコードが不足し、命題プリミティブに対応するオペコードが一覧に表示されない場合には、逐次データベースに 5 つ組 (r, s, o, C, O) のレコードをデータベースに追加し、本ツールがサポート可能な命題プリミティブ・コマンドの対応付けを拡充する。

一覧から必要なオペコードを選択すると、右の領域に選択したオペコードを含むコマンドの送受信機器・オペランドの編集が可能となる。テスト設計者は、命題プリミティブを参照してコマンドの送受信機器を設定し、デシジョンテーブルの真理値を参照して、テストで設定するオペランドを決定し、テストに必要なコマンドのリストを完成させる（2.）。図 4.3 のツールは完成したコマンドのリストに書かれた各コマンドをコマンドの送受信ステップへと置き換えることで、テストケースリポジトリにて資産化可能なテストケースを生成する（3.）。

```
[
  {
    "sa": "ic",------------------------ 初期化コマンドの送信元機器
    "da": "rc",------------------------ 初期化コマンドの送信先機器
    "sendCmd": {
      "opcode": "00,aa", ---------- 初期化コマンドのオペコード
      "operand": "00,01" ---------- 初期化コマンドのオペランド
    },
    "responseCmd": {
      "opcode": "10,aa", ---------- 初期化コマンドの応答のオペコード
      "operand": "00"--------------- 初期化コマンドの応答のオペランド
    }
  },
  {
```

図 4.4　初期化手順定義ファイル

テストの実行に際しては、テスト対象の機能を操作するためのテストケースに加えて、予めテスト対象の組込みソフトウェアを初期化する手順が必要となる。図 4.3 では、組込みソフトウェアの初期化手順は、図 4.4 に示すような形式で、定義し、任意のテストケース生成時に再利用することで、テストケースの作成を容易化する。

図 4.4 では、組込みソフトウェアの初期化を行うための送受信コマンドが JSON 形式のリストとして表現されており、リスト内の各要素が初期化用のコマンドに相当する。図 4.3 は、リスト内の要素ごとに初期化コマンドを送信するステップと、応答の受信を待機するステップに変換し、初期化手順を生成する。上記初期化手順は、テストケース生成時に初期化手順ファイルを読み込み、上述のコマンドの送受信ステップと合わせてテストケースに埋め込む。

加えて、デシジョンテーブルを元に、テスト実行ユニットによる自動評価を実現するためのテストの評価項目を記述する。テストの評価項目は、テスト実行ユニットに搭載された Spin による自動評価を実現するため、LTL で記述する。

命題論理には時系列の情報がないため、時相論理演算子は、デシジョンテーブル中のセミ形式記述を意味解釈して設定する。例えば、「受信する（リモコン、電源設定コマンド、室外機に、ON で）−> 設定する（室外機、電源、ON に）」というようなセミ形式記述について、「 □(c_expris_received(rc, oc, "電源設定 ON コマンド") −> ◇c_exprcompare_state(oc "電源", "==", "ON"))」という LTL 式によって、「室外機（oc）がリモコン（rc）から電源設定を ON にするコマンド（電源設定 ON コマンド」を受信したら、いつか室外機の電源が ON になるという事象が常に（必ず）起こる」というテストの評価項目を記述する（なお、c_expr は、オリジナルの Promela が持つ任意の C 言語コードにより真偽値の評価を行う構文である。compare_state は、アドレス oc の機器の組込みソフトウェアの内部変数 state を取得し、状態の値"ON"と一致しない（!=）かを比較した結果を真偽値で返す関数とし、is_received は、アドレス rc の機器から oc に送信されたメッセージ"電源設定 ON コマンド"を受信した時に真、そうでない時に偽を返す関数とする）。

4.5 不具合ログからのテストケース設計支援

市場にて不具合が発生した場合には、その不具合を事例として蓄積し、新機種の開発にて繰り返し再発がなされていないことを確認する必要がある。本節では、この過去不具合事例のテストケースの設計の支援のために、本研究で統合テスト環境の評価対象としたビル用空調システムにおける不具合発生時の通信ログから、不具合の要因となるコマンドの送信ステップの絞り込みと、テストケースリポジトリにて資産化可能なテストケースの生成を行うツール [55] について述べる。

本支援ツールは、図 4.5 に示すものである。右側の領域には、不具合発生時のコマンドの通信ログが表記されている。ログの一行は一つのコマンドに相当し、各コマンドには、コマンドの送信元・送信先アドレス、及びコマンドのオペコード・オペランドが含まれる。この通信ログ上でテストケースに含めるコマンドを絞り込み、絞り込んだコマンドから自動生成されたテストケースを統合テスト環境により自動実行することで、テストケースの再現に必要な最小限のテストステップを含むテストケースの設計を支援する。絞り込み支援としては次を実施する。

1. 指定時間内のコマンドのみに絞り込む
2. 指定アドレスの機器に対するコマンドのみに絞り込む

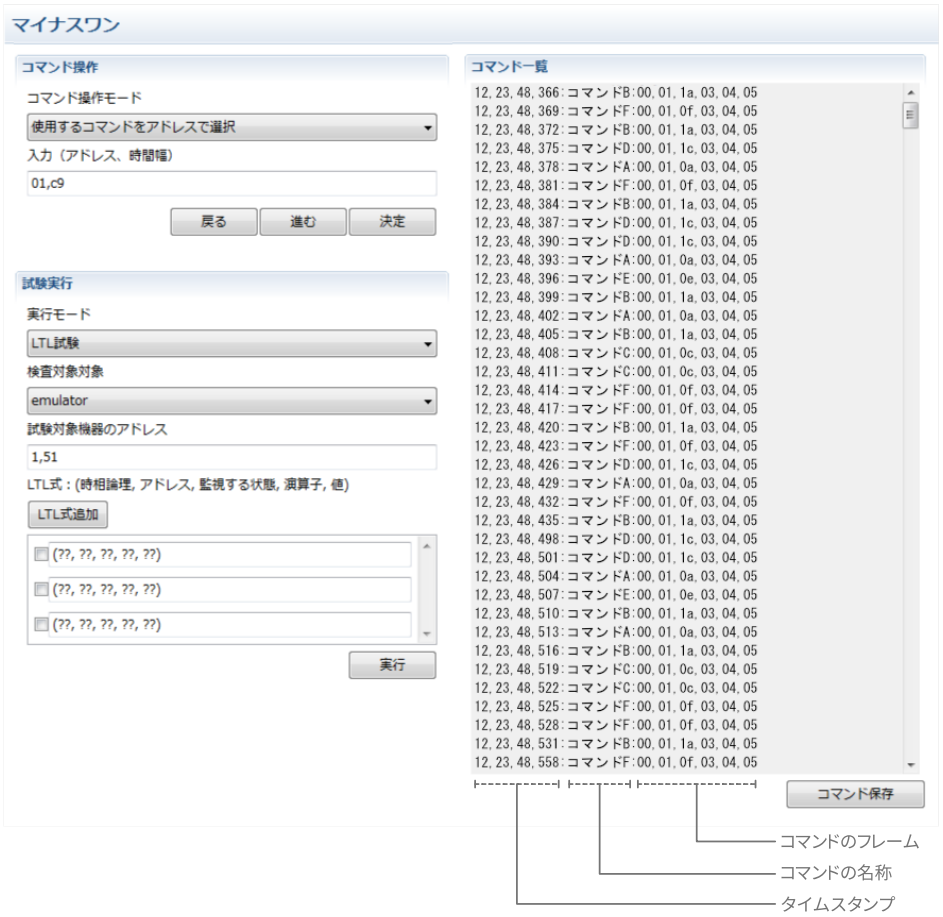


図 4.5　ログからのテストケース設計支援ツール

絞り込まれた各コマンドを一つのコマンド送信ステップに変換し、かつ 4.4 節と同様の同様の初期化手順定義ファイルを読み込むことで、テストケースリポジトリにて資産化可能なテストケースを生成する。

4.6　統合テスト環境の自動テストのケーススタディ

4.6.1 小節に示すビル用空調システムの過去不具合事例を対象に統合テスト環境による自動テストを実施し、次の 3 つの項目を確認した。

1. 部品化したテストケースから非決定的な手順のテストケースが生成されること（図 4.1 の ①）
2. 非決定的なテスト手順を含むテストケースが起こりうる全ての順序で実行されること（図 4.1 の ②）
3. テスト結果の自動評価が可能なこと

本ケーススタディでは、後工程で不具合が発生した表 4.1 の 3 件の事例を用いた。テスト対象の組込みソフトウェアは、本統合テスト環境が網羅的な順序でテストを実行可能なこと、不具合の検

表 4.1　不具合事例と評価項目

	事例 1	事例 2	事例 3
不具合の現象	3 つのメッセージで起動する機能 F が特定の順序でしか起動しなかった。	機能 A の後に機能 B を動作させると、機能 B による機器状態の変更が反映されなかった。	機能 C は機器のモードによって制御対象を異なる状態に変えた。ここで、モードの異なる 2 つの機器 D1、D2 から機能 C で機器 D3 を制御すると、機器 D3 の状態が不安定になった。
評価項目（LTL）	$\Diamond(S_1 = v_a)$	$\Diamond(S_2 = v_b)$	$\Box(S_3 \neq v_e)$
意味	（機能 F が）いつか状態 S_1 を値 v_a にする	（機能 B が）いつか状態 S_2 を値 v_b にする	（機器 D2 のモードで起動した機能 C によって）状態 S_3 は値 v_e にずっと変更されない

表 4.2　組込みソフトウェアの不具合の有無

	不具合の有無		
組込みソフトウェア動作環境	事例 1	事例 2	事例 3
実機	修正済み	-	-
エミュレータ	修正済み	修正済み	未修正

出が可能なことを確認するために、表 4.2 のように、不具合の修正済みのもの、未修正のものを含むように設定し、事例 1 を実機上の組込みソフトウェアを対象に、事例 1–事例 3 をエミュレータ上の組込みソフトウェアを対象にテストを実施した。

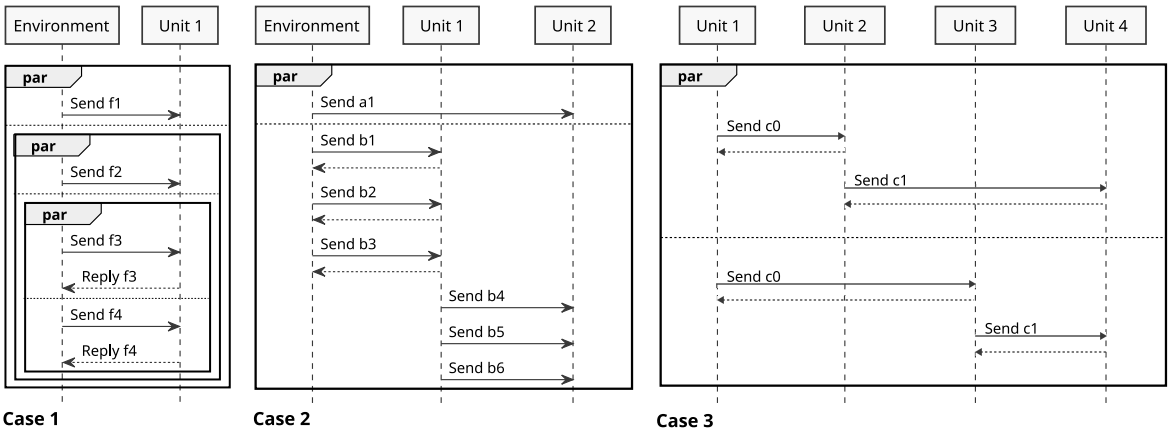


図 4.6　表 4.1 の事例 1–3 のテストケース

評価に用いるテストケースを図 4.6 に示す。テストケースは、図 4.1 の ① の方法により、機器構成・テストケースの組み合わせによって生成する。エミュレータによる自動テストでは、組込みソフトウェアの内部状態を参照し、不具合事例が再発していないことを確認する表 4.1 の最下段の LTL 式を評価項目に設定した。

4.6.1　ケーススタディ対象のビル用空調システム

表 4.3　ビル用空調システムを構成する機器

機種	1 年あたりのリリース台数
室内機	約 50 台
室外機	約 50 台
リモートコントローラ	約 5 台
システムコントローラ	約 1–2 台

本ケーススタディで評価対象とするビル用空調システムは表 4.3 のような複数機器で構成され、機器間でメッセージ交換を行って、協調動作する。機器間のメッセージ交換はユーザのリモコン操作や特定の気温・湿度への変化、機器ごとのタイマといった、設計時に順序を決定できない要素がトリガーとなる。このため、メッセージによって起動される機能の実行順序は非決定的になる。

こうした機能の実行順序に起因する不具合が起こらないことを確認するために、機能の実行順序を変えたテストを実施している。また同時に、接続機器の組み合わせや接続台数の違いによる不具合を防ぐために、同じ機能に対するテストでも複数の機器構成で複数回実施している。

本ビル用空調システムの開発にはプロダクトライン開発が適用されており、ネットワークを介したメッセージ交換の機能等が共通化され、コア資産として蓄積されている。回帰テストでは、上述のコア資産の機能と差分開発の機能の組み合わせのもとで、各機能の期待結果が得られること、異常状態が起きないことの確認が必要である。この結果、室外機 1 台あたり約 4000 件もの回帰テストが必要となる。

4.6.2 自動テスト実行の結果

表 4.4 自動テストの結果

事例	結果	期待結果	テストされた実行順序
事例 1	事例発生せず	事例発生せず	6 通り
事例 2	事例発生せず	事例発生せず	2 通り
事例 3	事例発生	事例発生	1 通り

テストの実行の結果、表 4.4 の成否判定結果を得た。

まず、部品化された機能ごとのテストケースから、非決定的な手順のテストケースが生成されたこと（4.6 節の 1.）を確認する。事例 2 のテストケースは、a1 からなる機能 A のテストケースと b1–b6 からなる機能 B のテストケースを統合することにより生成された。事例 3 も同様に部品化された機能ごとのテストケースよりテストケースが生成された。不具合を含まない組込みソフトウェアを対象にテストを行った事例 2 では生成したテストケースを用いて、機能 A と機能 B を非決定的な順序で実行するテストが実施できた。この結果より、4.6 節の 1. が実現されたことを確認した。

次に、非決定的なテスト手順を含むテストケースが起こりうる全ての順序で実行されること（4.6 節の 2.）を確認する。表 4.1 の事例 1 では、統合テスト環境から並列な 3 つのメッセージ f2–f4 が送信される（図 4.6）。このとき、起こりうる実行順序は、

$$3! = 6 \quad [\text{通り}]$$

である。統合テスト環境から送信された f1–f4 の送信順序は、実機・エミュレータ上のいずれの組込みソフトウェアを対象とした場合においてもこの数と一致する次の 6 通りあった。

1. f1 → f2 → f3 → f4
2. f1 → f2 → f4 → f3
3. f1 → f3 → f2 → f4
4. f1 → f3 → f4 → f2
5. f1 → f4 → f2 → f3
6. f1 → f4 → f3 → f2

上記送信順序から、非決定的な 3 つのメッセージ f2–f4 が起こりうる 6 通り全ての順序で送信されることを確認した。

表 4.1 の事例 2 では、機能 A と機能 B が並列に動作し、機能 A には 1 つのメッセージ a1 が、機能 B には 3 つのメッセージ b1–b3 が統合テスト環境から送信される。機能 A（a1）と機能 B（b1–b3）をまとめた単位での衝突を行ったため、このとき、起こりうる実行順序は、

$$\frac{(1+1)!}{1!1!} = 2 \quad [\text{通り}]$$

である。エミュレータ上の自動テストにおいて、統合テスト環境から送信された a1 と b1–b3 の送信順序は、この数と一致する次の 2 通りあった。

1. a1 → b1 → b2 → b3
2. b1 → b2 → b3 → a1

上記送信順序から、a1 と b1–b3 が起こりうる 2 通り全ての順序で送信されることを確認した。

上記より、（4.6 節の 2.）が実現されたことを確認した。

続いて、本統合テストにより、テスト結果を自動評価可能なこと（4.6 節の 3.）を確認する。

表 4.1 の事例 3 では、不具合修正前の組込みソフトウェアをエミュレータ上で動作させてテストを実施した。テストの結果、表 4.4 に示すとおりテストが失敗し、不具合が検出された。図 4.7 は不具合発生時のシーケンスを可視化したものである。統合テスト環境がエミュレータを介して評価した組込みソフトウェアの内部変数 s が、不具合である値 v に変化し、その結果 $\square(S_3 \neq v_e)$ という評価項目に違反し、不具合 $S_3 = v_e$ を検出したことを示している。

上記結果から、統合テスト環境により、テストケースが自動実行・評価され、（4.6 節の 3.）が実現されたことを確認した。

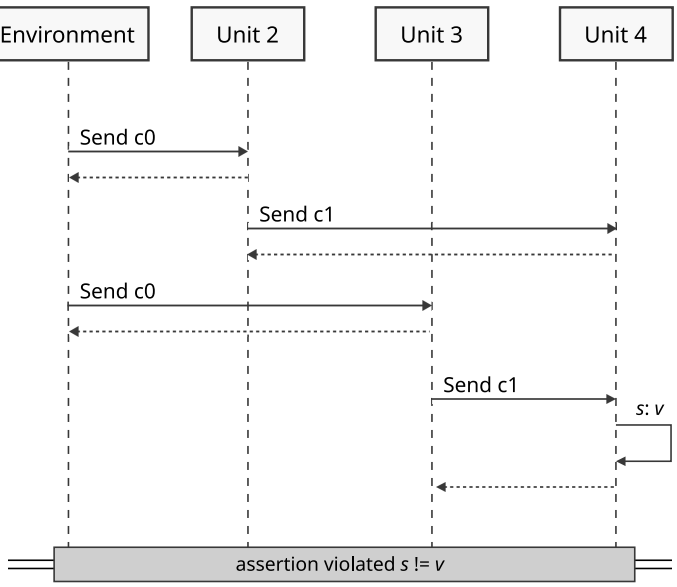


図 4.7　事例 3 の実行結果

4.6.3　テスト結果についての考察

4.6.2 小節の評価結果について考察する。

従来のテストケースは派生製品開発で新規に導入される機能の組み合わせのもとでテストケースを設計する。4.6.2 小節では、部品化された機能ごとのテストケースを任意に組み合わせて、複数の機能を非決定的な手順で実行するテストケースが生成されることを示した。本手法により、機能ごとに作成したテストケースの再利用を実現し、派生製品開発ごとに発生していたテストケースの作成工数が削減される。

本ケーススタディでは、本統合テスト環境により、評価が困難なテストが自動化されたことも確認した。事例 1 は、メッセージ単位で送信順序を制御しなければテストできないものであった。しかし、システムテストの段階では、メッセージ単位での制御が不可能であり、手動では評価困難であった。また、事例 2、3 は、評価に必要なソフトウェア内部の変数をメッセージから確認できないため、従来の統合テスト環境では自動評価できなかった。

本研究では、モデル検査器 Spin と実行テストを融合するインターフェイスの開発により、通信順序を自動的に網羅的に入れ替えてテストを実施し、またエミュレータを介して組込みソフトウェアの内部変数を評価することで従来、ハードウェアが内部変数の確認用インターフェイスを具備するかによらずテスト可能なことを示した。

事例 1–3 は次の非決定的なテストケースを含む典型的なテストケースであった。

- 単体機能のメッセージの送信順序の入れ替え（事例 1）
- 複数機能の実行順序の入れ替え（事例 2、3）

派生製品の開発時には、従来機能の単体機能の回帰テストと、新機能と従来機能を組み合わせた回帰テストが必要となる。手動では困難であったいずれの非決定的な手順のテストも本統合テスト環境で自動評価することで解決される。

以上より、工期の都合上、手動ではテスト困難であったプロダクトライン開発の回帰テストの課題を、自動テストによりテスト工数を削減し、解決する統合テスト環境を実現できた。

4.7　エミュレータと統合テスト環境を用いたテストの工数比較

統合テスト環境により、従来の手作業でのテストと比べてテスト工数が削減されることをケーススタディを通して示す。テスト対象のシステムには、後述の 4.6.1 小節で示すビル用空調システムを採用した。ケーススタディでは、ビル用空調システムの過去不具合の再発を確認するテストをビル用空調システムの開発・テストに習熟したエンジニアに実施してもらい、テストに必要な作業の観察と各作業の実行時間を計測した。ケーススタディを行う実験室には、後述する不具合事例表 4.1 の再現に必要なテスト対象機器を相互の接続を行わずに設置し、各機器の動作確認（ログのモニタやメッセージの仕様書の閲覧）に必要な PC を予め起動した状態で設置してテストを実施してもらった。

統合テスト環境がエミュレータを介して機器の内部変数を自動評価可能なことを示すために、同じ過去不具合の再発を確認するテストを統合テスト環境でも実施した。

4.7.1 ケーススタディの結果と考察

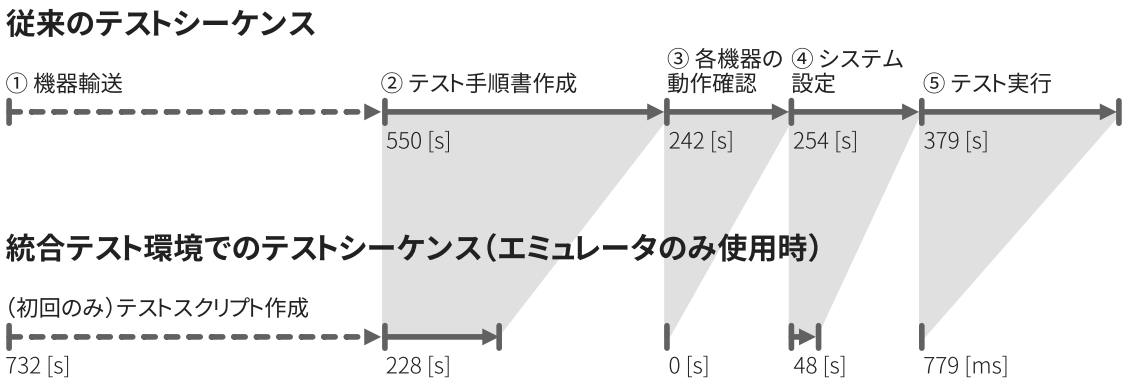


図 4.8　表 4.1 の不具合事例のテスト作業

図 4.8 の上側の手順でテストが実施され、ステップの実行には、ステップ間の矢印に示す時間がかかることが観察された。また、著者による統合テスト環境によるテストを実施したところ、図 4.8 の下側の手順のような時間がかかることが観察された。

統合テスト環境により、次の 3 点でテスト工数が削減される

1. テスト手順書の作成工数（図 4.8 ②）
2. テストのためのハードウェア準備工数（図 4.8 ①、③）
3. テストの実行工数（図 4.8 ⑤）

テスト手順書の作成工数の削減（1.）について、従来のテストシーケンスでは、②の「テスト手順書の作成」において、テスト対象の機器構成、テスト手順、及びテスト結果の確認手順を文書化する作業が実施された。本文書には、テスト対象の複数機能を組み合わせた状態でテスト手順が記載されるため、派生製品開発で新規の機能組み合わせが導入される度に既存の機能を含んでいたとしても、繰り返しテスト手順を設計し直す工数を要した。

一方、統合テスト環境では、機能ごとに部品化されたテストスクリプトを用いて、自動的に複数機能を決定的・非決定的に組み合わせたテストスクリプトを生成できるため、手動でテストケースを再設計し直す必要がない。このため、②では、テストケースを生成するための部品化されたテストスクリプトのファイル名の指定のみでテスト手順の設計が代替できるため、その分少ない工数で遂行できる。初回はテストスクリプト作成の工数が必要となるものの、プロダクトライン開発では同じ回帰テストを繰り返し実行するので、テストスクリプトを繰り返し再利用することで削減される②の工数により、長期的には挽回される。

ハードウェア準備工数の削減（2.）について、③の各機器の動作確認は、接続したハードウェアが故障なく正常に動作することを確認するステップであり、機器ごとに電源の投入時のログを確認

する作業が行われた。④のシステム設定は、機器同士を有線ネットワークに接続し、かつソフトウェア上で機器の接続設定を行うステップであり、物理配線及び、機器に搭載された専用のインターフェイス上で接続設定を行い、その後に、機器の連携動作の確認作業が行われた。

③及び④の物理配線作業はハードウェアを対象とする従来テスト手法に特有のものであり、本統合テスト環境にてエミュレータのみで実施するテストでは本工数を削減できる。また、物理配線作業は接続する機器台数に比例して工数が増加するため、最大機器構成のテストなど機器台数が多いほどテスト工数の削減に貢献できる。

また、本ケーススタディで計測した作業時間外にも、従来のテストシーケンスでは、テスト対象機器のハードウェア自体を実験室に用意するために、対象機器のハードウェアを輸送する①のステップが生じる。エミュレータを用いたテストではこれら作業を不要とする結果、このテストステップの工数が削減される。

テストの実行工数の削減（3.）について、⑤のテスト実行は、従来手法では、ハードウェアを操作して機器間の通信を発生させることでテストを実行し、テスト結果については、ハードウェアに搭載された機器状態を表示するインターフェイス（LCD 表示）及び通信ログから流れたコマンドのパラメータを確認することによって実施された。対して統合テスト環境では、ハードウェアの操作を要さず、自動的に通信コマンドを送受信し、かつ送受信されたコマンドから自動的にテストの成否を評価することで、従来手法よりも工数を削減できる。

上記では、従来のハードウェアを用いた手動でのテストシーケンスと、エミュレータのみを用いて機器構成を再現して行った統合テスト環境のテストシーケンスについて比較した。本エミュレータは、周辺デバイスのメモリ（VRAM など）のモニタなどは行わないため、ソフトウェア内部変数と LCD 表示の不一致など、ハードウェアの動作自体を評価するためには、エミュレータと機器のハードウェアを組み合わせたテストが必要となる。この場合、ハードウェアを用いたテストに固有の作業として、①、③、④での物理配線作業が必要となる。しかし、テスト対象機器と通信する対向機器はエミュレータで代行できるため、対向機器の輸送・機器の動作確認にかかるテスト工数は、エミュレータで模擬される機器の台数だけ削減される。また、②にかかる工数は提案手法により削減されるため、テストに必要な機器を全てハードウェアで用意し、手動でテストを実行する従来のテストシーケンスに比べて、テストの工数は削減される。

以上より、エミュレータを活用した本統合テスト環境により、従来のハードウェアを用いた手動での回帰テストの工数を削減されることが確認できた。

4.8 本章のまとめ

本章では、決定・非決定的なテストの実行にかかる工数を削減するために、第 3 章で設計されたテストケースを用いて自動的なテストの実行・評価するモデル検査技術と実行テストの技術を融合した統合テスト環境を開発した。統合テスト環境では、複数機能を用いた決定・非決定的なテストケースの設計にかかる工数・実行の両面を削減するために、機能ごとのテストケースを部品とし、任意の部品組み合わせから決定・非決定的に機能を実行するテストケースを生成することで、決

定・非決定的に機能を実行するテストケースの設計にかかる工数の削減を実現した。また、生成された決定・非決定的に機能を実行するテストケースを、モデル検査の技術を活用して自動的に実行・評価することで、テストの実行・評価にかかる工数の削減を実現した。

さらに、上記テスト実行にかかる工数を、開発の上流工程から下流工程まで一貫して削減するために、次の 2 種類のインターフェイスを開発した。

1. ハードウェア実装前の組込みソフトウェアをテストするためのエミュレータ
2. ハードウェア上の組込み機器ソフトウェアをテストするためのゲートウェイ

統合テスト環境について、プロダクトライン開発が適用されているビル用空調機器を対象とした自動テストのケーススタディを行った。ケーススタディの結果、機能ごとのテストケースの部品を用いて非決定的な手順で機能を実行するテストケースを生成し、生成されたテストケースを自動的かつ網羅的な実行順序で実行・評価可能なことを確認した。

また、従来のハードウェアを用いた手動によるテストシーケンスと、統合テスト環境によるテストシーケンスを比較したところ、統合テスト環境によるテストの自動実行・評価、及び部品化されたテストケースの再利用により、テスト実行にかかる工数が削減されることを確認した。

以上、提案する統合テスト環境により、プロダクトライン開発におけるテストの課題である、テスト実行にかかる工数の削減が実現できたと考える。

第 5 章　機能・シナリオを衝突させたテストケースの設計支援

第 4 章では、決定・非決定的な実行順序を網羅的に実行・評価するテスト環境について述べた。一方、非決定的な実行順序を網羅するテスト（以降順序入れ替えテストと呼ぶ）は取り得る実行順序の数が並列に動作する機能の数が増えるにつれて式（5.1）のように指数的に増加するため、構造的に並列に動作しうる全ての機能を対象にして実施することは現実的には困難である。

$$N_o = \frac{\left(\sum_{i=1}^{N_p} N_{s_i}\right)!}{\prod_{i=1}^{N_p} N_{s_i}!}. \tag{5.1}$$

（N_o: プロセス P_i が N_{s_i} のステップを持つとき（ただし $1 \leq i \leq N_p$））

従来では、機能の実行によるリソースの変更の影響を受けたり・与えたりする可能性のある機能の組み合わせに限定して順序入れ替えテストのテストケースを設計・実行していたが、左記のような組み合わせはシステム仕様書に明示されることは少なく、テスト設計者が暗黙的に決定していたため、テストに漏れが発生しうる課題があった。

本章では上記テスト対象の機能の組み合わせ決定の課題を解決するために、第 3 章で得られるセミ形式記述から機能の間にある影響関係・競合関係を抽出し、機能シナリオの実行順序と合わせて可視化することで、並列に実行させるべき機能・シナリオの選択支援手法について述べる。これにより、式（5.1）の N_{s_i} や N_p を制限することで、起こりうる実行順序の数を削減する手法について述べる。また可視化した機能・シナリオの実行順序から統合テスト環境で自動テスト可能なテスト手順に変換するアルゴリズムについても紹介し、本可視化図式で分析した機能・シナリオの自動評価を可能にする。

5.1　機能の実行順序と影響・競合関係の関係の図式表現

本研究では、機能間の優先順位の違反、機器の状態の不正な変更や、デッドロックといった機能の並列動作に起因する不具合を起こしうる機能の組み合わせを発見するために、機能が参照・変更する状態を元に、次の 2 種類の機能間の関係を自動的に算出する。

影響関係: 機能 A が機能 B より先に実行される、あるいは機能 A と B が並列に実行され、かつ機能 A が変更する状態を機能 B が参照するときの、機能 A と機能 B の関係

競合関係: 機能 A と機能 B が並列に実行され、かつ機能 C と機能 D が共通する状態を変更するときの、機能 C と機能 D の関係

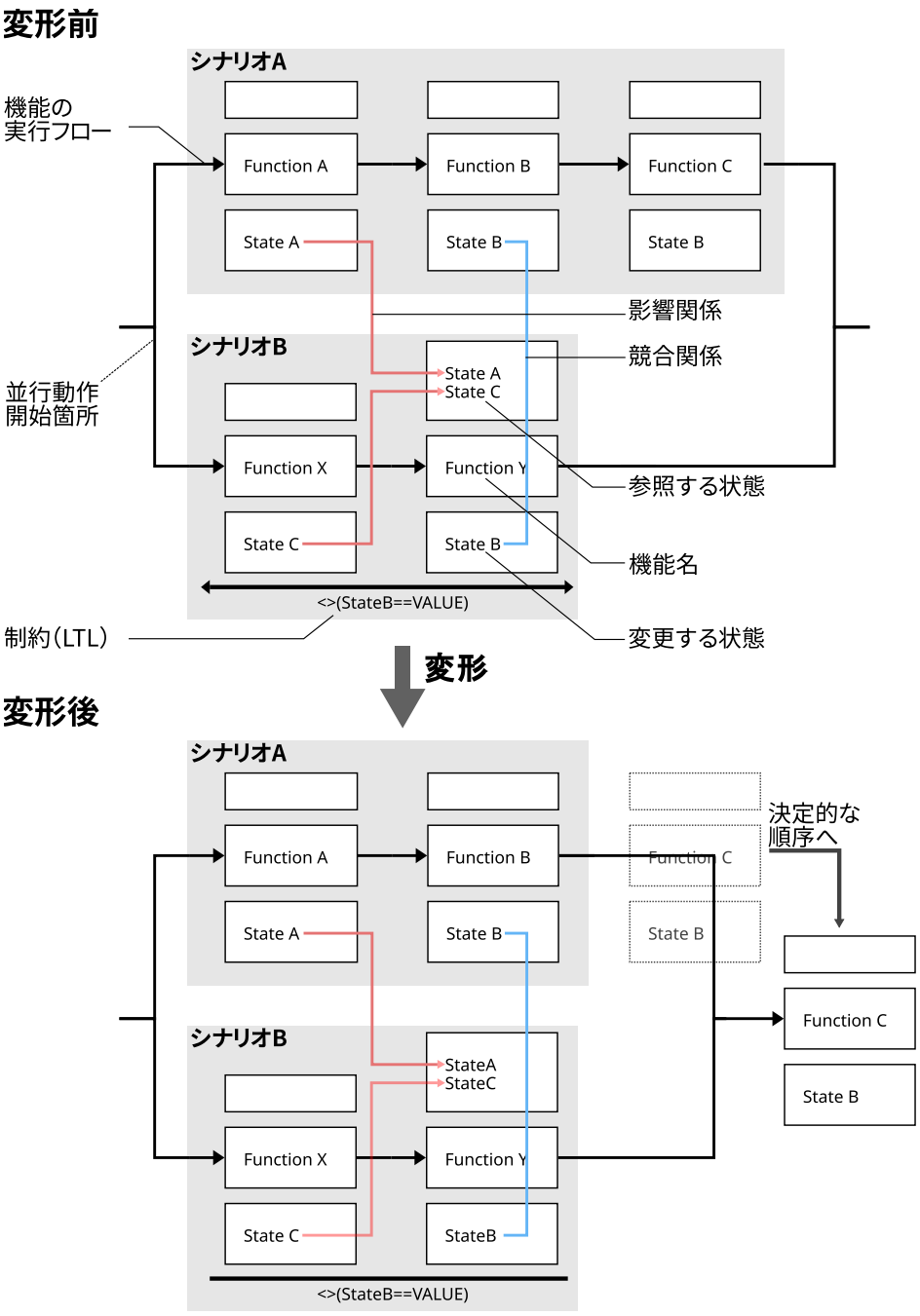


図 5.1　機能間の関係と実行順序の可視化例

順序入れ替えテスト対象の機能・シナリオの選択を支援するために開発した可視化図式は図 5.1 である。図 5.1 では、機能の実行順序と機能間の影響関係 [56] 及び機能間の競合関係を可視化する。なお、機能の実行順序についてはアクティビティ図や BPMN [57] でも表現可能であるが、前述のように本可視化図式ではテストする機能・シナリオの選択にあたり、機能が参照・変更する状態や機能間の影響・競合関係が重要と考えて、特にこれら情報を把握できるように強調して表現した図式となっている。

3 段組の矩形で 1 つの機能を表現し、上段に参照する状態、中段に機能の内容を表すセミ形式記述、下段に機能が変更する状態を記述する（なお、図 5.1 では、例示する図をシンプルにするためにセミ形式記述部分を Function X という形式で記述した)。黒い矢印は機能の実行順序を表し、枝分かれをする箇所で並列動作が開始され、矢印が交流する箇所で並列動作が終了する。機能の間の赤い矢印が影響関係であり、影響を与える機能から影響を受ける機能に対して矢印が引かれる。機能の間の青い線が競合関係を表す。機能の下に引かれている複数機能にまたがる矢印はその期間に機能が満たすべき制約条件を示しており、LTL で記述する。

機能間に影響・競合関係がない場合は、どの順序で機能を実行しても、お互いの実行が互いに影響を与えることはないと判断できる。テスト設計者がこの影響・競合関係にない機能がクリティカルな機能でなく、順序を入れ替えたテストが必要ないと判断する時、順序入れ替えテスト対象から除いたり、非決定的な実行手順を決定的な手順に変更したりする（図 5.1）ことで、式 (5.1) より N_p や N_{s_i} が削減され、順序入れ替えテストのテストケースの数を削減できる。

例えば、図 5.1 には、Function C には影響･競合関係にあるものがない。したがって、Function C を並列に動作する（非決定的な順序で動作する）記述から決定的な順序での記述へと変形することで、起こりうる実行順序は、$\frac{(2+3)!}{2!3!} = 10$[通り] から $\frac{(2+2)!}{2!2!} = 6$[通り] に削減できる。

5.2 機能の実行順序・影響関係の算出

機能の参照・変更される状態を機械的に抽出可能にするために、セミ形式記述を入力に影響・競合関係の算出を行う。セミ形式記述では、機能の入力項目 C と確認項目 A の間の論理関係を Imply（「–>」）を使って、C–> A と表現する。このため、Imply の前件・後件からそれぞれ機能の実行開始時に参照する状態、機能が実行時に変更する状態を取得できる。セミ形式記述では、参照･変更される機器の各状態は、命題プリミティブ「関係語（主体語、対象語、制約語)」のうち、関係語が is でないときは対象語・制約語の部分、関係語が is のときは主体の部分に記載するため、この部分を機械的に抽出することによって得られる。

抽出した機能の参照・変更する状態、及び機能の実行順序は図 5.2 の定義の YAML ファイルを用いて、図 5.3 のように記述する。機能の実行順序は、システム仕様書には記載されないため、テスト設計者が手動で設定する。機能間の影響・競合関係は、設定された機能の実行順序と抽出した参照する状態・変更する状態を元に、自動的に同定し、実行順序が前後・並列になっている機能のうち、同じ状態を変更・参照している機能同士（影響関係のある機能）や、同じ変数を変更する機能同士（競合関係のある機能）を自動的に可視化する。

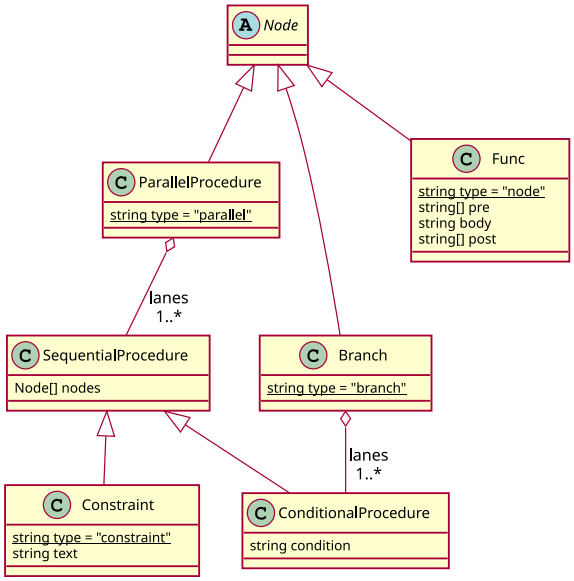


図 5.2　順序・影響関係可視化図式の入力ファイルの定義

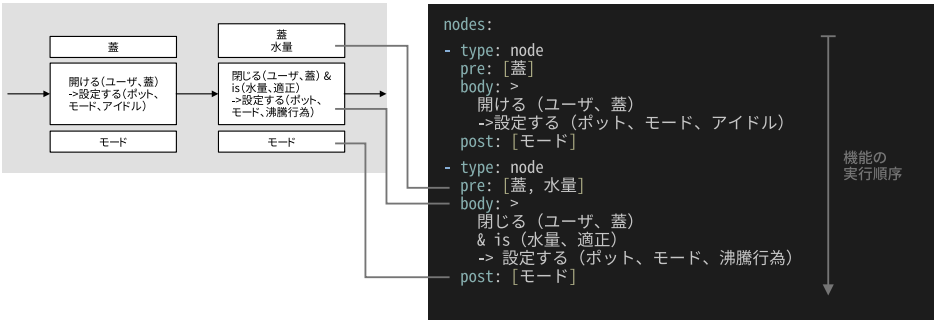


図 5.3　順序・影響関係可視化図式の入力ファイルの例

5.3 可視化図式によるテストケース生成

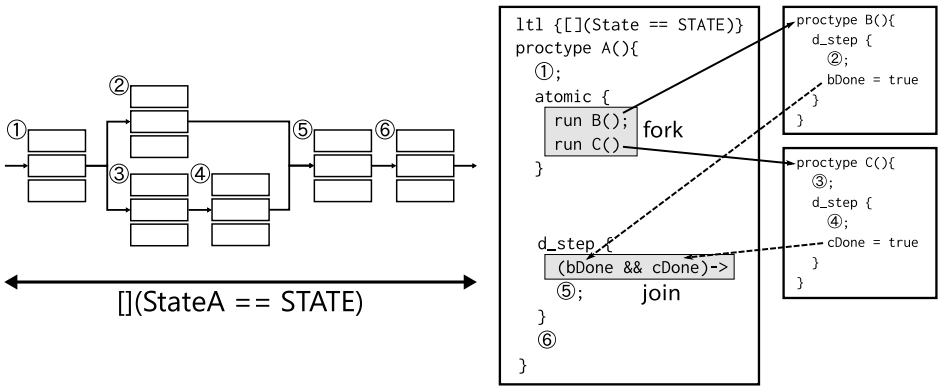


図 5.4 図式からのテストケース生成方法

本節では、本研究で提案する順序・影響関係の図式表現から Promela テストケースを生成する方法を述べる。テストケース記述言語に Promela を採用することで、統合テスト環境を用いて生成したテストケースの自動テストも可能にする。

図式からモデル検査用のテストケースを生成する手法には、CDL から生成するもの [58] や、UML の状態遷移図から生成するもの [59] があるが、本研究における可視化図式は、これら手法が表現可能な実行順序に加え、機能間の影響・競合関係についても表現することで、テストすべきシナリオの範囲設定にも貢献する。

図式から順序通りの実行と並列な実行を反映したテストケースを生成は、可視化図式を実行順序の早い順（左）から順番に見ていき、機能のノードか並列な分岐かに応じて次の 2 通りの方法で Promela に反映することで実現できる。

- **順序通り（分岐していない）ステップの並び:**
  1. 単一プロセス上にステップを図の順で並べる
- **並列な（分岐している）ステップの並び:**
  1. 分岐の発生箇所で子プロセスを生成する
  2. 子プロセスの終了まで false となる判定式を、子プロセス生成直後に置く
  3. 子プロセスの最終ステップに、プロセス終了の旨を示すフラグを立てる

Promela で実行の順序が保証されているものは、`proctype` によるプロセスの定義中にあるステップのシーケンスのみである。したがって、順序通りの実行が必要なステップについて全て一つの `proctype` に統合する（1）ことで、順序通りのステップの実行を実現する（図 5.4 の `proctype A`）。一方、プロセス同士は、統合テスト環境が搭載する Spin によって網羅的な順序で実行される。上記動作原理に基づき、並列な実行が指定されたステップを別々の `proctype` に定義することで、並列なステップの実行を実現する（図 5.4 の `proctype B` と `C`）。

定義を分けたプロセスを、並列にステップを実行すべき箇所にて子プロセスとして同時に生成する（1）ことで、任意箇所にてステップを並列にテストできる。2 は、生成した子プロセスの実行が終わる前に子プロセスを生成した元のプロセスが進行することを防ぐためのものである。SPIN は判定式が true になるまでプロセス中のステップを進行しないので、3 が行われる子プロセスの終了まで、後続処理が並列な処理実行中に先行しない（図 5.4 の `proctype A の (bDone && cDone)`）。

図 5.4 の下部にある `[](StateA == STATE)` の部分は、この可視化図式中で満たすべき状態の制約であり、モデル検査用テストケースへと変換した際に LTL) 式として実現され、SPIN による網羅的なテスト実行中に違反しないか評価される。これにより、どのような順序で機能が動作しても制約が成立することを確認できる。

5.4 可視化図式を用いたテスト対象シナリオ削減のケーススタディ

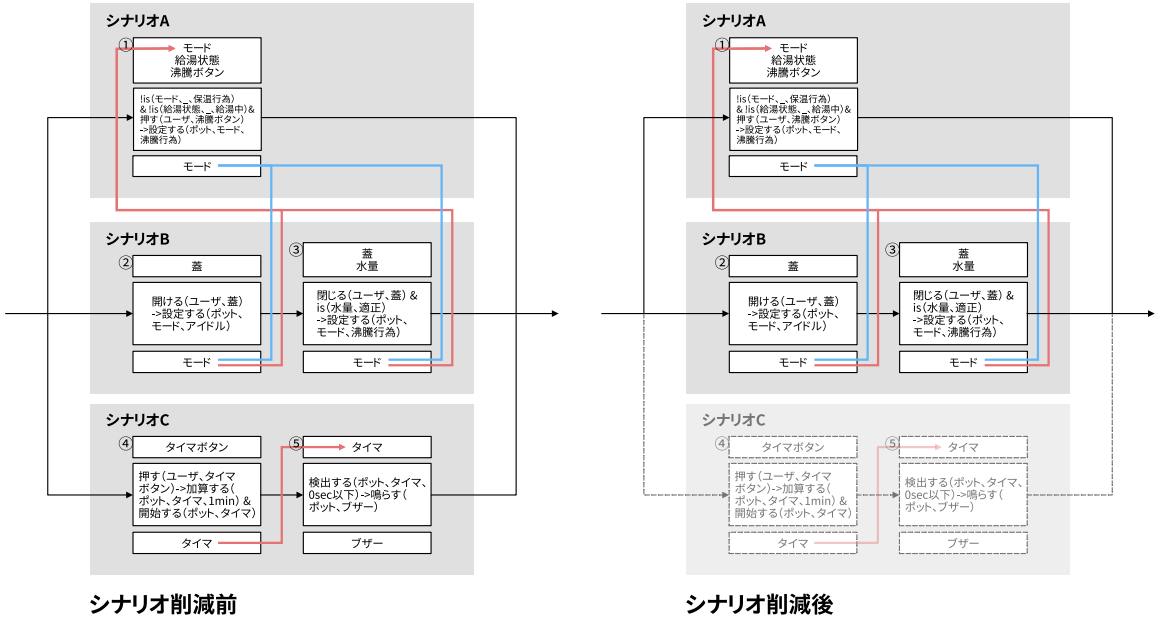


図 5.5　評価用可視化図式

本論文で提案する可視化図式を用いてテスト範囲を絞り込むことで、得られるテストケースの数を削減できることを確認する。方法として、可視化図式上から特定の状態に関連する機能・シナリオの調査を試行し、これに関連するシナリオのみに限定したテストケースを作成し、テストされた実行順序の数を比較する。ここでは評価のために話題沸騰ポット第 7 版 [60] を元に作成した次のセミ形式記述形式の機能について、図 5.5 の左側に 3 つのシナリオを可視化した。

シナリオ A（沸騰ボタンを押して沸騰行為にする）

- !is（モード，_，保温行為）& !is（給湯状態，_，給湯中）& 押す（ユーザ，沸騰ボタン）–> 設定する（ポット，モード，沸騰行為）

シナリオ B（蓋を閉じて沸騰行為をする）：

- 開ける（ユーザ，蓋）–> 設定する（ポット，モード，アイドル）
- 閉じる（ユーザ，蓋）& is（水量，適正）–> 設定する（ポット，モード，沸騰行為）

シナリオ C（キッチンタイマを使う）：

- 押す（ユーザ，タイマボタン）–> 加算する（ポット，タイマ，1min）& 開始する（ポット，タイマ）
- 検出する（ポット，タイマ，0sec 以下）–> 鳴らす（ポット，ブザー）

シナリオ A は、沸騰ボタンを押下することで、ポットの温度制御のモードを沸騰モードに変更するものである。本評価で使用した [60] の仕様書では、保温行為中で、給湯中でないときに沸騰ボタンによる温度制御モードの変更が可能である旨の記載がある。なお、今回の例では、沸騰ボタンを押すことによるアイドル状態から蓋センサが on になることによる沸騰行為状態への変化は、①を実行中に起こる、すなわち記載した可視化図式の範囲内で起こるものとする。

シナリオ B は、ポットの蓋を閉じることで沸騰行為状態になることを確認するシナリオで、「ポットの蓋を開ける」、「蓋を閉じて沸騰行為にする」ことを順序通りに実行するシナリオである。なお、今回の例では、蓋を開けるまえにポット中に適正とされるだけの水量が存在するものとする。また、蓋が閉じることによるアイドル状態から沸騰行為状態への変化は、③ を実行中に起こる、すなわち記載した可視化図式の範囲内で起こるものとする。

シナリオ C は、タイマボタンを押して設定した時間にブザーが鳴ることを確認するシナリオで、「キッチンタイマの時間の加算」、「キッチンタイマの時間が 0 秒になった際のブザー鳴動」の機能を順序どおりに実行するシナリオである。本ケーススタディでは「モード」に関係する機能を含むシナリオのみをテストするようにテスト対象のシナリオの選択を行う。

さらに、モデル検査器用のテストケースへの変換について、本仕様書には具体的な実装についての記述はないため、可視化図式中の各機能のノード内の各操作の実行は PROMELA 上では `printf` と置き、手動にて変換を実施した。また⑤ のタイマ減算処理についても手動にて実現した。

5.5 ケーススタディの結果

表 5.1 テストされた実行順序の数

テスト範囲	実行順序の数
全体版（図 5.5 の左側）	30
範囲限定版（図 5.5 の右側）	3

モードに関連する機能について、影響・競合関係にある機能を含むシナリオのみにテスト対象のシナリオを制限すると、図 5.5 の右側に示す破線の機能のノードがテスト範囲から除外された。テスト範囲絞り込み前の図式（図 5.5 の左側）、絞り込み後のテスト図式（図 5.5 の右側）をそれぞれ本論文で提案する方法に基づいて PROMELA テストケースに変換すると、図 5.6、図 5.7 のようなテスト手順の PROMELA テストケースが得られる。このテストケースについて、SPIN によってテストされた順序入れ替えの数を数えると、表 5.1 となった。

```
1  bool scenarioADone
2  bool scenarioBDone
3  bool scenarioCDone
4  proctype ScenarioA() {
5    d_step {
6      printf("1\n")
7      scenarioADone = true
8    }
9  }
10
11 proctype ScenarioB() {
12   printf("2\n")
13   d_step {
14     printf("3\n")
15     scenarioBDone = true
16   }
17 }
18
19 proctype ScenarioC() {
20   d_step {
21     printf("4\n")
22   }
23   d_step {
24     printf("5\n")
25     scenarioCDone = true
26   }
27 }
28
29 init {
30   d_step {
31     run ScenarioA(); run ScenarioB(); run ScenarioC()
32   }
33   (scenarioADone && scenarioBDone && scenarioCDone)
34 }
```

図 5.6　図 5.5 左側の Promela テストケース

5.6　考察

本ケーススタディでは、ポットの「モード」を介して影響関係のある機能を含むシナリオについてテスト対象のシナリオの選択を試行した。図 5.5 の左側を見ると、「モード」を介して影響関係にある機能（② → ①、② → ②、③ → ①）、及び競合関係にある機能（① → ② ① → ③）が明示されている。テスト設計者は図式に明示された影響・競合関係を元に、テスト対象の機能・シナリオを選択できる。

従来では、システム仕様書に影響・競合関係について明示されることが少なく、テスト設計者が暗黙的に影響・競合関係にある機能・シナリオを推測し、テスト対象として選択していた。暗黙的

```
1  bool scenarioADone
2  bool scenarioBDone
3  proctype ScenarioA() {
4    d_step {
5      printf("1\n")
6      scenarioADone = true
7    }
8  }
9
10 proctype ScenarioB() {
11   printf("2\n")
12   d_step {
13     printf("3\n")
14     scenarioBDone = true
15   }
16 }
17
18 init {
19   d_step {
20     run ScenarioA(); run ScenarioB()
21   }
22   (scenarioADone && scenarioBDone)
23 }
```

図 5.7　図 5.5 右側の Promela テストケース

なプロセスで行われる従来の機能・シナリオの選択方法では、選択対象の機能・シナリオの見落としの可能性があり、不具合が後工程に流出することがあった。

本手法では、機能間で共通に参照・変更されている状態を元に、影響・競合関係にある機能・シナリオ全体を可視化したことで、明示された影響・競合関係を根拠に、検討漏れなくテストする機能・シナリオの選択が可能となった。

次にテスト手順について、表 5.1 を見ると、本可視化図式によって優先度付けを行って限定したテスト範囲について生成したテストケースのほうがテストされた実行順序の数が少なくなっている。これは、図 5.5 中の並列に動く 3 つのシナリオからシナリオ C が除外されたことで順序の入れ替わりの候補が減ったことによるものである。

図 5.5 の左側の時点では、機能 ④–⑤ は ①–③ のどの時点でも実行される可能性があるために、これらが ④–⑤ との前後に実行される場合を実行順序の入れ替えたテストケースに含める必要があった。図 5.5 の右側ではこの入れ替わりの候補である ④–⑤ が存在していない。ここから、本可視化図式によるテスト範囲の制限により実行順序の入れ替えの数を削減できることがわかる。

以上、可視化図式上で影響・競合関係を元にしたテスト対象機能・シナリオの選択が可能であることを例示した。また、並列に動作しうるシナリオ群から、影響・競合関係にないシナリオを除くことで、起こりうる実行順序の数を削減できることを示した。

5.7　本章のまとめ

本章では、並列動作する機能を網羅的な順序で実行するテストについて、2 つの課題

1. 状態爆発によってテストの実行が困難なこと
2. システム仕様書中に機能間の影響関係・競合関係が明示されることが少なく、テスト対象の機能の限定が困難なこと

に取り組んだ。この課題を解決するために、機能間の影響・競合関係の可視化し、一つのテストケース中で並列に実行すべき機能の限定を支援することで、起こりうる実行順序の数を削減する手法を提案した。

提案手法では、第 3 章のセミ形式記述の仕様から各機能が参照・変更する状態を抽出し、この状態と、機能の実行順序を元に、影響関係・競合関係にある機能を可視化する図式を開発した。この可視化図式により、図式上に明示された関係を見ることで、一つのテストケース中で並列に実行すべき機能を発見可能にした。また、可視化図式に表現された決定・非決定的なテスト手順を第 4 章の統合テスト環境用のテストケースへ変換するアルゴリズムについても紹介した。これにより、テスト対象の機能の決定から自動評価までの一貫した支援を実現した。

ケーススタディとして、自然言語で書かれた仕様書から 3 種類のシナリオを図式化したところ、図式上で機能間の影響・競合関係が可視化でき、かつテスト対象の機能の選択が可能なことを確認した。また、本図式でテスト対象の機能を限定することで起こりうる実行順序の削減に寄与することを確認した。

以上から、本章で提案する非決定的な実行順序で行うテストの対象機能の選択支援により、状態爆発を回避し、現実的に実施可能な工期で統合テスト環境による自動評価が可能になると考える。

第 6 章　結論と今後の課題

本研究では、プロダクトライン開発を用いた組込み機器のシステムテストにおける、次の 2 つの課題の解決のために、テストケースの設計からテストの実行までにアプローチした。

1. 自然言語で書かれた欠陥を含むシステム仕様書テストケース設計において、テスト設計者が暗黙的なプロセスで欠陥の修正・テストケースへの変換を行うため、結果修正の誤りがあった場合に、手戻りが大きくなること。
2. 複数機能を衝突させたテストを派生製品の開発の中で繰り返し行うに当たり、テストケースの設計・実行に工数がかかること。

自然言語で書かれたシステム仕様書を元にしたテストケース設計の課題解決のために、本研究では、現場のテスト技術者へのヒアリングを元に、テストケース設計プロセスを再定義した。再定義したテストケース設計プロセスでは、欠陥が含まれる自然言語で書かれたシステム仕様書から、段階的にテストケースへ変換していき、変換の中間成果物をレビューすることで、欠陥が修正されたシステム仕様書を元にテストケースを設計する。中間成果物について明示的なレビューを実施することで、手戻りを小さくするとともに、システム仕様書からテストケースへの自動変換によるテストケース設計工数の削減、及びシステム仕様の可視化ツールによるレビューの容易化を図った。

提案するテストケース設計プロセスは、評価の結果、自然言語で書かれた仕様書を入力に、欠陥の修正を行いながら、最終的にテストケースの生成が可能なことを確認し、実行可能なテストケースとして実現できたことを確認した。

プロダクトライン開発を用いた組込み機器における回帰テスト工数が膨大である課題について、テストケース設計プロセスにより生成されたテストスクリプトを自動評価する統合テスト環境により、解決を試みた。統合テスト環境では、並列に動作しうる機能に対する決定・非決定的な順序でのテストをモデル検査技術の活用により、自動的・網羅的にテストする。開発した統合テスト環境では、実機上で動作する組込みソフトウェアに対し、通信順序を網羅的に入れ替えたブラックボックスのテスト、及びエミュレータ上で動作する組込みソフトウェアに対し、機器の内部状態を網羅的な順序で操作するホワイトボックスなテストの両方の視点でのテストを実現した。これにより、従来手法では困難であった特に組込みソフトウェアの内部状態を参照しなければ評価不可能なテストも可能となり、開発の上流から下流工程までをサポートするテスト環境を実現した。

開発した統合テスト環境の評価として、プロダクトライン開発を用いて開発されているビル用空調システムの過去不具合事例を統合テスト環境で自動テストした。テストの結果、統合テスト環境は、複数の機能の並列に動作させることによる非決定的なテスト手順を、ブラックボックス・ホワイトボックス両方の視点で自動的に実行可能なことを確認した。また、テストケースリポジトリの活用により、機能ごとに独立に書かれたテスト手順を組み合わせて非決定的なテスト手順のテスト

ケースが生成できること、生成するテストケースはテスト実行時の機器構成に合わせて柔軟に書き換え可能なこと、不具合事例を確認する評価項目を生成するテストケースに埋め込み、自動的に評価できることを確認した。

この結果から、統合テスト環境により、起こりうる機器構成・機能の実行順序が多いことからテスト実行の工数が多くテスト実行が困難であった課題について、自動的なテスト実行により解決されることを確認した。

加えて、複数機能を衝突させたテストにおける状態爆発に対処するために、テスト対象とする機能を限定し、起こりうるテスト実行の工数を削減する手法として、テストケース設計プロセスの成果物である形式化された仕様書から機能間の影響関係・競合関係を可視化することによって、テスト対象の機能の決定を支援する手法を提案した。提案手法では、並列動作する機能が他のどの機能の実行に影響を与えるかを明らかにするために、形式化された仕様書から、機能が参照・変更する状態を抽出し、共通する状態を参照・変更する機能を影響関係（ある機能が変更した状態を別の機能が参照する関係）と競合関係（ある機能が変更した状態を別の機能も変更する関係）として可視化するツールを開発した。テスト設計者は可視化されている図式を根拠に、影響・競合関係にある機能を発見できる。この図式表現により、並列動作する機能の順序を入れ替えたテストにおける、テスト対象の機能の選択が困難な課題の解決を図った。

また、可視化された図式に対応する Promela 形式のテストケースの生成アルゴリズムについても提案し、後述の統合テスト環境にて自動テストを行うことで、機能の実行順序を入れ替えたテストの実行自体の困難さの解決にもアプローチした。

機能間の実行順序と影響・競合関係を可視化する図式表現について、自然言語で書かれた仕様書から作成した 3 種類のシナリオを図式化したところ、機能間にある影響・競合関係が可視化できること、この影響・競合関係により、テスト対象の機能の選択が可能なことを確認した。また、本図式でテスト対象の機能を限定することで起こりうる実行順序が削減されることを確認した。この結果から、並列動作する機能の順序を入れ替えたテストにおいて、従来困難であったテスト対象の機能の選択支援が実現でき、テスト対象の機能を限定することで、起こりうる実行順序の削減が実現されることを確認した。

本研究で提案する以上手法により、プロダクトライン開発を用いた組込み機器におけるテストケースの設計から実行までテストの課題を解決した。これにより、プロダクトライン開発で従来取り組まれていた設計・実装工程の開発の効率化に、本研究で提案するテスト工程の効率化を加えることで、開発工程を効率化可能なプロダクトライン開発が実現可能となったと考える。

今後は、テストケース設計プロセスにおいて、自然言語仕様書からセミ形式記述への変換ルールの拡充を行い、変換の正確さの向上を図る。

現在、テストケースの設計・実行の一連を支援するテストケース設計プロセスの遂行支援ツール群、及び統合テスト環境について、実用化に向けたフィージビリティスタディを行っている。現場のシステムへの適用と現場の技術者からのフィードバックにより、より実践的な環境としてブラッシュアップを行っていく。

謝辞

私の指導教員である久代紀之先生に心より感謝申し上げます。学部時代より数えて 7 年間もの間、非常に熱心かつ親身にご指導を下さいました。研究室配属当初より研究の進め方から論文の書き方、わかりやすい研究発表の仕方と言った、研究に関わる何から何まで、大変親身にご指導いただきました。また、学会発表の渡航費から論文掲載費、研究用の計算機の貸与といった経済的な支援もしてくださいました。また、研究活動以外にも、私が技術者として成長するために、アプリケーション開発のアルバイトを斡旋してくださったり、私が人間的に成長するために、例えば人前で話をするのが得意でなかったことについて、久代紀之先生の担当する授業の一部を任せてくださったり、短期の海外留学を紹介してくださったりと、様々な機会を与えて下さいました。私が今日まで円滑かつ着実に、楽しく研究を進めることができたのは久代紀之先生のおかげです。また、先生のおかげでこの 7 年間、とても貴重な経験を積むことができました。感謝してもしきれません。心より深謝申し上げます。

私の副指導教官である、吉田隆一先生、古賀雅伸先生に心より感謝申し上げます。研究計画や研究・開発報告書のレビューを通じて、ご助言や激励の言葉を下さいました。自身の研究活動について見直すことができたとともに、頂いたお言葉に大変励まされました。

本論文のご査読、ご審査、ご助言を下さいました、副査の梅田政信先生、坂本比呂志先生、副指導教官でもある古賀雅伸先生に心より感謝申し上げます。予備審査・公聴会でいただいたご質疑やご助言のおかげで、公聴会の発表はもとより、本論文の完成度を上げることができました。

研究を進めるにあたって、実験機材や仕様書、テストケースのご提供、及び評価実験にご協力下さいました、三菱電機株式会社の皆様に心より感謝申し上げます。特に、黒岩丈瑠様、中野裕梨様、福田亜実様には、共同研究者として活発に議論をさせていただきました。私がこうして研究成果を博士論文にまとめることができたのは、皆様との密な連携のおかげです。

また、本学の社会人学生でもある黒岩丈瑠様には、私の学部時代に初めて研究というものに取り組んだ頃から研究活動について親身にご教示、ご助言、激励を下さいました。おかげさまで博士課程まで着実に研究を進めることができました。本当にありがとうございました。

仕様書やテストケースのご提供、及び評価実験にご協力下さいました、パーソル AVC テクノロジー株式会社様に心より感謝申し上げます。頂いた現場のドキュメントや実験へのご協力くださった皆様のおかげで、こうして研究成果をまとめることができました。

同じ研究室の学生である皆様に心より感謝申し上げます。共同研究者とは異なる外部からの視点でいただいた皆様の質疑・コメントのおかげで、着実に研究を進めることができました。

本研究は JSPS 科研費 16K00100 の一部支援を受けました。ご支援、心より感謝申し上げます。

最後に、私の生活をこれまで支えてくれた家族に感謝申し上げます。日々健康に研究活動に没頭できたのは、経済面、精神面ともに支えてくれる母や、その母を手伝い、支えてくれる妹のおかげ

です。

ここにお名前を上げることはいたしませんが、他にも既に卒業してしまった同研究室の学生や、講究発表や学会発表にてコメントをくださった先生方など、本当に多くの皆様のおかげでこの論文をまとめることができました。これまでお世話になった皆様に改めて御礼申し上げ、謝辞といたします。本当にありがとうございました。

参考文献

[1] 財務省. 輸出入額の推移（主要商品別）. https://www.customs.go.jp/toukei/suii/html/time.htm. Last accessed: Jan 20, 2020.

[2] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[3] Goetz Botterweck and Andreas Pleuss. *Evolution of Software Product Lines*, pp. 265–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[4] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[5] Eugene Diirr and J Van Katwijk. VDM++: A formal specification language for object oriented designs. In *Proceedings 6th Annual European Computer Conference, Compeuro*, pp. 214–219, 1992.

[6] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[7] Achim D Brucker, Frank Rittinger, and Burkhart Wolff. Hol-z 2.0. *Journal of Universal Computer Science*, Vol. 9, No. 2, pp. 152–172, 2003.

[8] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. Vdmtools: Advances in support for formal modeling in vdm. *SIGPLAN Not.*, Vol. 43, No. 2, pp. 3–11, February 2008.

[9] Clearsy. https://www.atelierb.eu/en/atelier-b-tools/. Last accessed: Jan 20, 2020.

[10] 張紅輝, 大西淳. 異なる視点で記述されたシナリオの統合支援. コンピュータ ソフトウェア, Vol. 20, No. 5, pp. 414–431, 2003.

[11] 高柳俊祐, 上條敦史, 石川勉. 日本語文から拡張型述語論理式への自動変換ツール: Conv. 人工知能学会論文誌, Vol. 27, No. 5, pp. 271–280, 2012.

[12] Benedikt Gleich, Oliver Creighton, and Leonid Kof. Ambiguity detection: Towards a tool explaining ambiguity sources. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pp. 218–232. Springer, Springer, 2010.

[13] Hui Yang, Anne De Roeck, Alistair Willis, and Bashar Nuseibeh. A methodology for automatic identification of nocuous ambiguity. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pp. 1218–1226. Association for Computational Linguistics, 2010.

[14] Erik Kamsties, Daniel M Berry, and Barbara Paech. Detecting ambiguities in requirements documents using inspections. In *Proceedings of the first workshop on inspection in software engineering (WISE'01)*, pp. 68–80, 2001.

[15] Colette Rolland and Camille Ben Achour. Guiding the construction of textual use case specifications. *Data & Knowledge Engineering*, Vol. 25, No. 1-2, pp. 125–160, 1998.

[16] 増田聡, 松尾谷徹, 津田和彦. テストケース作成自動化のための意味役割付与方法. コンピュータ ソフトウェア, Vol. 34, No. 2, pp. 16–27, 2017.

[17] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Nat2testscr: Test case generation from natural language requirements based on scr specifications. *Science of Computer Programming*, Vol. 95, pp. 275–297, 2014.

[18] Hironori Takeuchi, Taiga Nakamura, and Takahira Yamaguchi. Predicate argument structure analysis for use case description modeling. *IEICE TRANSACTIONS on Information and Systems*, Vol. 95, No. 7, pp. 1959–1968, 2012.

[19] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 321–330. ACM, 2011.

[20] Kim Lauenroth, Klaus Pohl, and Simon Töhning. Model checking of domain artifacts in product line engineering. *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 269–280, 2009.

[21] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering - GPCE '12*, GPCE '12, pp. 11–20, New York, New York, USA, 2012. ACM Press.

[22] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 372–375. IEEE Computer Society, 2011.

[23] Kim G Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pp. 299–306. ACM, 2005.

[24] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pp. 31–43, December 2003.

[25] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, Vol. 36, No. SI, pp. 75–88, 2002.

[26] René G de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 2, No. 4, pp. 382–393, 2000.

[27] 古川覚, 上田賀一, 中島震. 組込みシステム検査のための協調解析. コンピュータ ソフトウェア, Vol. 31, No. 3, pp. 307–317, 2014.

[28] Kei Homma, Satoru Izumi, Kaoru Takahashi, and Atsushi Togashi. Modeling, verification and testing of web applications using model checker. *IEICE transactions on information and systems*, Vol. 94, No. 5, pp. 989–999, 2011.

[29] Yongyan Zheng, Jiong Zhou, and Paul Krause. A model checking based test case generation framework for web services. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pp. 715–722. IEEE, 2007.

[30] José García-Fanjul, Claudio De La Riva, and Javier Tuya. Generation of conformance test suites for compositions of web services using model checking. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pp. 127–130. IEEE, 2006.

[31] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pp. 83–91. IEEE, 2001.

[32] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 2018.

[33] Sascha Lity, Thomas Morbach, Thomas Thüm, and Ina Schaefer. Applying incremental model slicing to product-line regression testing. In *International Conference on Software Reuse*, pp. 3–19. Springer, 2016.

[34] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software*, Vol. 147, pp. 46–63, 2019.

[35] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pp. 57–68. ACM, 2011.

[36] 久代紀之, 青山裕介, 村上響一. 仕様書からのテストケース設計プロセスの定義とテストケース生成支援ツール. 情報科学技術フォーラム 講演論文集, 第 17 巻, pp. 15–21, 2018. （選奨論文セッション）.

[37] 青山裕介, 黒岩丈瑠, 久代紀之. 自然言語仕様からの機能間の並列・順序動作の抽出と左記テスト環境. 情報科学技術フォーラム 講演論文集, 第 16 巻, pp. 35–40, 2017.

[38] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*, pp. 65–84. John Wiley & Sons, 2 edition, 2004. 長尾真 and 松尾正信（訳）. ソフトウェア・テストの技法, 近代科学社, 2006.

[39] 青山裕介, 黒岩丈瑠, 久代紀之. テストケース生成のためのシステム仕様書の論理記述変換アルゴリズム. 情報処理学会論文誌, Vol. 61, No. 3, 2020. （掲載予定）.

[40] Arseny Tolmachev, Daisuke Kawahara, and Sadao Kurohashi. Juman++: A morphological analysis toolkit for scriptio continua. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 54–59, 2018.

[41] Daisuke Kawahara and Sadao Kurohashi. A fully-lexicalized probabilistic model for japanese syntactic and case structure analysis. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational*

Linguistics, pp. 176–183. Association for Computational Linguistics, 2006.

[42] 河原大輔. KNP で付与される feature 一覧. `http://nlp.ist.i.kyoto-u.ac.jp/index.php?KNP`, 2013. Last accessed: May 31, 2019.

[43] Charles J. Fillmore. *The case for case*, pp. 1–88. Holt, Rinehart, and Winston, 1968.

[44] 国立国語研究所. 日本語における表層格と深層格の対応関係, 国立国語研究所報告, 第 113 巻. 三省堂, 1997.

[45] Boris Beizer. *Software testing techniques*, pp. 269–276. Dreamtech Press, second edition, 2003. 小野間彰 and 山浦恒央（訳）. ソフトウェアテスト技法, 日経 BP 出版センター, 1994.

[46] Chris Drake. Python library for electronic design automation (PyEDA). `https://media.readthedocs.org/pdf/pyeda/v0.28.0/pyeda.pdf`, 2011. Last accessed: May 23, 2018.

[47] 組込みソフトウェア管理者・技術者育成研究会（SESSAME）. 話題沸騰ポット第 6 版. Last accessed: May 23, 2018.

[48] 組込みソフトウェア管理者・技術者育成研究会（SESSAME）. 話題沸騰ポット第 3 版. Last accessed: May 23, 2018.

[49] Yusuke Aoyama, Takeru Kuroiwa, and Noriyuki Kushiro. Hybrid testing environment of execution testing and model checking for product line approach. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 693–694, dec 2018.

[50] Takeru Kuroiwa, Yusuke Aoyama, and Noriyuki Kushiro. Testing environment for CPS by cooperating model checking with execution testing. *Procedia Computer Science*, Vol. 96, pp. 1341–1350, 2016. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 20th International Conference KES-2016.

[51] T. Kuroiwa, Y. Aoyama, and N. Kushiro. A hybrid testing environment between execution test and model checking for IoT system. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–2, Jan 2019.

[52] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, Vol. 1003. Addison-Wesley Reading, 2004.

[53] Takeru Kuroiwa and Noriyuki Kushiro. Testing environment for embedded software product lines. In *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–7. IEEE, 2015.

[54] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*, pp. 31–33. MIT press, 1999.

[55] 十川雄司, 青山裕介, 黒岩丈瑠, 久代紀之. LTL 式による動作ログからの不具合要因特定支援ツールの構築. ウィンターワークショップ 2018・イン・宮島 論文集, Vol. 2018, pp. 16–17, 2018.

[56] 青山裕介, 黒岩丈瑠, 久代紀之. モデル検査の実行順序制約の図式表現と試験ケースの自動生成. 電子情報通信学会論文誌 D, Vol. J101-D, No. 3, pp. 502–511, 2018.

[57] Object Management Group. Business process model and notation (BPMN) version 2.0. `http:`

//www.omg.org/spec/BPMN/2.0/, 2011.

[58] Philippe Dhaussy, Jean-Charles Roger, and Frederic Boniol. Reducing state explosion with context modeling for model-checking. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pp. 130–137. IEEE, 2011.

[59] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal aspects of computing*, Vol. 11, No. 6, pp. 637–664, 1999.

[60] 組込みソフトウェア管理者・技術者育成研究会（SESSAME）. 話題沸騰ポット第 7 版. Last accessed: May 23, 2018.

成果一覧

審査のある原著論文: 3 報（うち 1 報は掲載予定）

[1] 青山裕介, 黒岩丈瑠, 久代紀之. テストケース生成のためのシステム仕様書の論理記述変換アルゴリズム. 情報処理学会論文誌, Vol. 61, No. 3, pp. 521–534, mar 2020.

[2] 黒岩丈瑠, 青山裕介, 久代紀之. エミュレーション技術の活用による IoT システムのテスト効率化. 電気学会論文誌, Vol. 140, No. 1, pp. 113–121, 2020.

[3] 青山裕介, 黒岩丈瑠, 久代紀之. モデル検査の実行順序制約の図式表現と試験ケースの自動生成. 電子情報通信学会論文誌 D, Vol. J101-D, No. 3, pp. 502–511, 2018.

国際会議発表論文: 10 報（うち 2 報は掲載予定）

[1] Yusuke Aoyama, Noriyuki Kushiro, and Takeru Kuroiwa. Test case generation algorithms and tools for specifications in natural language. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, jan 2020. **（掲載予定）**.

[2] Takeru Kuroiwa, Yusuke Aoyama, and Noriyuki Kushiro. Automatic testing environment for virtual network embedded systems. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, jan 2020. **（掲載予定）**.

[3] Noriyuki Kushiro, Koshiro Nishinaga, Yusuke Aoyama, and Toshihiro Mega. Differences of risk knowledge described in work procedures manual and that used in real field by field overseers. Vol. 159, pp. 1928–1937, 2019. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019.

[4] Kazuhiro Mega, Toshihiro nad Komatsu, Shigenori Kawasaki, Yusuke Aoyama, Masatada Kawatsu, and Noriyuki Kushiro. Method of creating prediction model and controlling method for realizing demand response. In *2019 International Conference on Smart Grid and Green Energy (SGGE 2019)*, jan 2019.

[5] Takeru Kuroiwa, Yusuke Aoyama, and Noriyuki Kushiro. A hybrid testing environment between execution test and model checking for IoT system. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–2. IEEE, jan 2019.

[6] Yusuke Aoyama, Takeru Kuroiwa, and Noriyuki Kushiro. Hybrid testing environment of execution testing and model checking for product line approach. In *2018 25th Asia-Pacific Software*

Engineering Conference (APSEC), pp. 693–694, dec 2018.

[7] Masafumi Ifuku, Noriyuki Kushiro, and Yusuke Aoyama. Requirements definition with extended goal graph. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 210–218, nov 2018.

[8] Noriyuki Kushiro, Yusuke Aoyama, and Yuji Fujita. Tool for extracting latent field overseers' knowledge for risk recognition on eyes and algorithm for structuring meta and domain knowledge from utterances. *Procedia Computer Science*, Vol. 126, pp. 2003–2012, 2018. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia.

[9] Noriyuki Kushiro, Yuji Fujita, and Yusuke Aoyama. Extracting field oversees' features in risk recognition from data of eyes and utterances. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 590–596, nov 2017.

[10] Takeru Kuroiwa, Yusuke Aoyama, and Noriyuki Kushiro. Testing environment for CPS by cooperating model checking with execution testing. *Procedia Computer Science*, Vol. 96, pp. 1341–1350, 2016. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 20th International Conference KES-2016.

査読のある国内会議論文: 2 報

[1] 青山裕介, 黒岩丈瑠, 久代紀之. Cps のためのモデル検査・実行テスト統合試験環境の構築. 情報科学技術フォーラム 講演論文集, 第 15 巻, pp. 43–50, 2016.

[2] 青山裕介, 黒岩丈瑠, 久代紀之. 既存ソフトウェア部品を用いたソフトウェア開発におけるソースコード理解支援ツール. 情報科学技術フォーラム 講演論文集, 第 14 巻, pp. 111–118, 2015.

その他学会発表: 14 報

[1] 青山裕介, 黒岩丈瑠, 久代紀之. テストケース自動生成のための自然言語の形式変換アルゴリズム. 情報科学技術フォーラム 講演論文集, 第 18 巻, pp. 33–38, 2019. （選奨論文セッション）.

[2] 藤原佑介, 青山裕介, 久代紀之. 勾配ブースティング木と並列処理を用いたビル電力需要予測. 第 81 回全国大会講演論文集, 第 2019 巻, pp. 639–640, feb 2019.

[3] 村上神龍, 村上響一, 青山裕介, 久代紀之. システム仕様書からのテストケース生成支援ツールの開発. 第 81 回全国大会講演論文集, 第 2019 巻, pp. 225–226, feb 2019.

[4] 村上響一, 青山裕介, 村上神龍, 久代紀之. 自然言語で記載された仕様書からのテストケース自動生成アルゴリズムの構築. 情報処理学会研究報告, 第 2019-SE-201 巻, pp. 1–8, feb 2019.

[5] 久代紀之, 井福政文, 青山裕介. 要件定義ツール（拡張ゴールグラフ）に基づく合意形成のため

のデータ選定（予備実験）. 電子情報通信学会研究報告, 第 118 巻, pp. 81–86, feb 2019.

[6] 久代紀之, 青山裕介, 村上響一. 仕様書からのテストケース設計プロセスの定義とテストケース生成支援ツール. 情報科学技術フォーラム 講演論文集, 第 17 巻, pp. 15–21, 2018. （選奨論文セッション）.

[7] 青山裕介, 黒岩丈瑠, 久代紀之. 順序入れ替えテストケースの蓄積を実現するテスト実行環境. 第 JSAI2018 巻, pp. 1–4, 2018.

[8] 久代紀之, 青山裕介, 江平達哉, 新庄篤尚. アイデア会議のコミュニケーション可視化システムとノーインタフェース家電コンセプト構築への適用. 研究報告知能システム（ICS）, 第 2018-ICS-191 巻, pp. 1–8, 2018.

[9] 村上響一, 青山裕介, 村上神龍, 久代紀之, 牧茂, 田畑一政, 神代勉, 中村潤. 自然言語仕様書からの試験ケース生成のための条件・動作の同定手法. 研究報告ソフトウェア工学（SE）, 第 2018-SE-198 巻, pp. 1–7, 2018.

[10] 久代紀之, 藤田裕司, 村上響一, 青山裕介. リスク認知における知ってる/知らない知識の表出化と可視化. 電子情報通信学会技術研究報告, Vol. 117, No. 440, pp. 27–32, 2018.

[11] 十川雄司, 青山裕介, 黒岩丈瑠, 久代紀之. LTL 式による動作ログからの不具合要因特定支援ツールの構築. ウィンターワークショップ 2018・イン・宮島 論文集, Vol. 2018, pp. 16–17, 2018.

[12] 青山裕介, 黒岩丈瑠, 久代紀之. 自然言語仕様からの機能間の並列・順序動作の抽出と左記テスト環境. 情報科学技術フォーラム 講演論文集, 第 16 巻, pp. 35–40, 2017.

[13] 中野裕梨, 黒岩丈瑠, 青山裕介, 久代紀之. 設備機器の網羅的な通信試験におけるモデル検査の活用. 情報科学技術フォーラム 講演論文集, 第 14 巻, pp. 449–450, 2015.

[14] 青山裕介, 黒岩丈瑠, 久代紀之. オープンソースソフトウェアを用いたシステムの開発支援ツール. 研究報告ソフトウェア工学 (SE), 第 2014 巻, pp. 1–8, 2014.