修 士 論 文

モデル検査用記述に基づいたテストケースの生成法

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

Nguyen Tam Thi Minh

2009 年 3 月

修 士 論 文

モデル検査用記述に基づいたテストケースの生成法

指導教官　青木利晃 特任准教授

審査委員主査　青木利晃 特任准教授
審査委員　島津明 教授
審査委員　鈴木正人 准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710023　Nguyen Tam Thi Minh

提出年月: 2009 年 2 月

概 要

本研究は、実装が設計モデルで検証した性質を満たすことを保証するため、W-**方法** [3] **を参考にして、設計モデルと実装の整合性をテストする枠組みを提案する。設計モデルは、**Promela **で記述され、モデル検査ツール** Spin **により正しさが検証されていることを前提とする。また、実装は、ホワイトボックスであり、内部状態が見えることを前提とする。設計モデルと実装の両方を有限オートマトン** (FSM) **に変換し、設計モデルと実装の整合性をテストすることは、２つの** FSM **の整合性をテストすることになる。そして、**Spin **で表現される探索機能に基づいて、設計モデルからテストケースを自動生成する手法を提案する。そのため、**FSM **は、**Promela **記述に基づいて定義される。提案した手法を評価するため、小規模な実験を行った。**

キーワード: Spin**、**Promela**、テストケース、**W-**方法、整合性**.

目 次

図 目 次

表 目 次

第1章　はじめに

ソフトウェア開発における「要求分析」、「設計」、「実装」の流れでは、各工程の質に注目しなければならない。現在、設計工程の質を保証するため、モデル検査などを用いて設計モデルの検証を行う研究が行われている。しかし、検証した設計モデルに基づいて実装する際、設計モデルで検証した性質が失われてしまう恐れがある。設計検証の結果を失わないように実装するためには、設計と実装の整合性をとる必要があると考えられる。この整合性を保証する一つの解決策として、設計モデルと実装が整合しているかテストを行うことが考えられる。このテストはテストケースを生成し、それに基づいて行われる。そこで、本研究では、設計モデルから整合性を保証するためのテストケースを生成し、それに基づいて実装をテストする手法を提案する。

関連研究としては、適合テスト方法とテストケース自動生成法がある。適合テスト方法には、W-方法 [3] がある。テストケース自動生成法には、モデルベーステスト（MBT 手法）[5] やモデル検査の反例に基づいた方法 [1][7] などがある。W-方法は、オートマトン理論に基づいて仕様と実装の整合性をテストする手法である。W-方法では、仕様と実装の両方が決定性有限オートマトン（FSM）でモデル化されることを前提している。よって、仕様と実装の整合性をテストすることは、２つの FSM の整合性をテストすることを意味する。また、W-方法で使われる FSM には、完全に規定され、ミニマルであり、全ての状態が到達可能な状態であるという前提がある。これらの前提は、W-方法の実用性をを制限する。MBT 手法とは、テスト対象のシステム（System Under Test または SUT）のモデルに基づいて、テストケースを導出する手法である。このモデルは、正しさが保証されていない。また、設計者は、モデルを作成するので、設計者の考えによっては異なるモデルが作成されたり、作業途中で誤りが入る可能性がある。そのため、テストを行って結果を評価するときに、テスト結果が良くなければ、テストケース生成アルゴリズムだけではなく、モデルも修正する場合がある。モデル検査の反例に基づいた方法は、変異分析（Mutation Analysis）の適用に基づいて、テストケースを生成する手法である。変異分析は、モデル検査のモデルとモデル検査で検証される性質の不一致を生成する手法である。一つの不一致をモデル検査で処理すると、反例が出力される。この反例は、初期状態から検証される性質に違反する状態までの到達可能状態シーケンスであり、テストケースとして使われる。しかし、モデルにおいて全ての到達可能な状態シーケンスを網羅する反例の集合を生成することは困難である。つまり、モデル検査の反例に基づいた方法により生成されるテストケースの集合は完全であると言えない。

そこで本研究では、実装が設計モデルで検証した性質が満たすことを保証するため、設

計モデルと実装の整合性をテストする枠組みを提案する。設計モデルはモデル検査により正しさが検証されていることを前提とする。そして、その正しさが実装後も成立していることを保証するためのテストケースを生成し、それに基づいて設計モデルと実装の整合性をテストする。本研究では、正しさが保証された設計モデルからテストケースを自動生成する。今回は、モデル検査ツール Spin（Simple Promela INterpreter）[6] を取り扱うため、設計モデルは Promela（PROcess MEta LAnguage）言語で記述されている。よって、Promela による記述からテストケースを自動生成する手法を提案する。

また、これまでに OSEK/VDX[1] 仕様に基づいた Real Time Operating System(RTOS) の Spin による検証実験 [9] が行われており、Promela 言語による RTOS の設計モデルの記述 [10] が存在する。実装としては、公開されている TOPPER[2]/OSEK という RTOS がある。そこで本研究では、RTOS の設計モデルと実装の整合性実験を想定している。

[1] OSEK は「Open systems and the corresponding interfaces for automotive electronics」という意味であり、VDX は「Vehicle Distributed eXecutive」の略称である。

[2] TOPPERS は「Toyohashi OPen Platform for Embedded Real-time Systems」の略称である。

第2章　モデル検査とソフトウェアテストの概要

ハードウェアおよびソフトウエアシステムの正しさを保証するため、４つの主要な手法がある。シミュレーション、テスト、演繹的検証、および、モデル検査である。これらの中にでは、モデル検査とテストは比較的実用性が高い。

2.1　モデル検査

モデル検査（Model Checking）とは、形式システムをアルゴリズム的に検証する手法である。ハードウェアやソフトウェアの設計から導出されたモデルが形式仕様を満たすかどうか検証する。形式仕様は、時相論理式で記述される。モデル検査はハードウェア設計に適用されることが最も多い。これは、ソフトウェアに非決定性があり、アルゴリズム的な手法だけでは、まだ完全ではなく、証明も反証もできない場合があるためである。モデル検査は当初、離散状態系の論理的正当性を調べるために開発されたが、リアルタイムシステムや限定された形式のハイブリッドシステムにも対応できるよう拡張されてきている。

形式的に、問題は次のように表すことができる。時相論理式の形式 p として記述される望む性質、および、初期状態 s が付いている構造 M を与え、$M, s \models p$ を確認する。M が有限なら、モデル検査はグラフ探索として捉えることができる。2.1.1 節では、本研究で利用するモデル検査ツール SPIN について説明する。

2.1.1　モデル検査ツール Spin

Spin はモデル検査を行うための公開されているソフトウエアツールであり、分散ソフトウエアシステムの形式的検証に使用されている。このツールは、1980 年からベル研究所で開発されている。Spin で使われる記述言語は Promela である。そして、検査対象のシステムは Promela で記述することよりモデル化され、Spin により検証される。モデルが取り得る状態遷移を網羅的に生成して、不正な状態 (例えば、デッドロック) などが発生しないかどうかを自動的に検証する。

2.1.2 記述言語 Promela

Promela は、実装のために使用されることを前提として設計された言語ではなく，システムの抽象化が行いやすいよう設計されている。そのため、Promela は計算ではなくプロセス同期および協調のモデル化に重点がおかれている。また、この言語はモデル検査の適用対象として、ハードウェア回路より、並行ソフトウェアシステムの記述に重点をおいている。Promela の基本的な構成要素は非同期プロセス、バッファーされるおよびバッファーされていないメッセージチャネル、同期ステートメント、構造化データである。これらの要素には、時間や時計の概念がなく、 浮動小数点数もない。計算機能についても最小限のものだけが提供されている。これらの制限事項は、平方根などの取扱いを難しくするが、プロセッサ網でのクライアントとサーバーの間の振舞いなどをモデル化することは比較的簡単になる。

2.2 ソフトウェアテスト

以上のように、モデル検査を用いれば、システムの正しさを検証できる。しかし、ソースコードのレベルではシステムの設計からモデルを抽出することは誤りがある可能性が高いので、実際にモデル検査を適用することが妨げられる。そのため、テストは、システムの品質を保証するいい方法である。テストは、テストデータとともにテスト環境でプログラムを実行することによって、プログラムの品質を査定するのに使用されている。この品質は、正しさだけではなく、信頼性、パフォーマンス、ロバスト性、および、有用性なども意味する。

2.2.1 テストの目標

テストは、品質保証の一部分として、バグの予防に焦点を合わせるべきである [2]。テストでバグを防がない限りで、バグを発見するべきである。最後に、テストはバグが容易に修正するできるように明瞭な診断結果を提供するべきである。

バグの予防は、テストの第一目標である。テストにより防がれたバグは、修正するべきコードがないので、検出され修正されるバグより良い。さらに、訂正が有効であることを確認する再試験は必要ではない。有用なテストは実装する前にバグを発見し、除去できる。さらに、理想的なテスト活動は全てのバグが見つけられ、修正されることである。

残念ながら、この理想は達成できない。人間がテスト活動を担当するので、必ずバグがある。テストがバグの予防という主要目的を達しない限り、テストは第二目標としてバグの発見に達しなければならない。バグは常に明らかではない。バグは期待された振舞いからの偏差で明示される。しかし、プログラムが不正確であることはバグを意味しない。そして、異なるバグは同じ明示を有し、一つのバグは多くの徴候を有することができる。

2.2.2 テスト方法論

テストの目的はプログラムの正しさの信頼を提供することである。プログラムの正当性を保証する唯一の方法はプログラムの全ての可能な入力を実行することであるが、これは不可能である。最も実行可能な代替案は、最大誤り数を見つけるのに十分な大きさがあるテストセットを生成することである。そうすれば、パスされるテストはプログラマにプログラムが仕様に合っているという確証を与える。

一般的に、テストセットには２種類ある。ブラックボックス（仕様ベース）テスト、および、ホワイトボックス（プログラムベース）テストである。

ブラックボックステスト
: ブラックボックステスト法は、行動テストとも呼ばれ、コード細部を分析せず、機能性を確認することによってプログラムの誤りを見つけるアプローチである。その方法の目的は「プログラムが仕様を満たすか」という質問に答えることである、あるいは、プログラムが仕様を満たさないかどうか見つけることである。行動テストは、構造の細部を取除き、高いレベルで抽象化してテストするので、ホワイトボックステストよりも簡単である。さらに、ブラックボックステストは、プログラムが間違った振舞いを実行しないかどうかテストできる。

ホワイトボックステスト
: ホワイトボックステスト法は、構造テストとして知られ、ソースコードのあらゆるラインが最低一回実行される。あるいは、あらゆる機能が個々にテストされるように要求するテストを設計する。ブラックボックステストは、システムの構造に関する知識を使用しないので、構造テストが生成するほど良いテストセットを生成できない。例えば、実装やプログラムの一部がある入力を無視するかもしれないので、誤るプログラムの部分があるかもしれない。これらの問題は行動テストに出て来ないかもしれない。

そこで、１つのテスト手法だけではなく、複数のテスト手法が適用されることが望ましい。このテスト手法は、グレイボックステストと呼ばれている。これらのテスト方法の一つ、または、複数を使用する異なるレベルの抽象化で様々なテスト技術がある。ソフトウエアシステムのテストには、典型的なものとして、ユニット/コンポーネントテスト、統合テスト、システムテストの３種がある。

ユニットテスト

最低レベルのテストは、ユニットテストと呼ばれる。ユニットとはテストできる最も小さな部分である。そのため、各機能、モジュール、および、クラスがそれぞれテストされる。ユニットテストは、単体が機能仕様を満足していないことを立証したり、構造が設計どおりでないことを立証するテストである。ユニットテストで、明らかになる誤りをユニットバグと呼ぶ。ユニットテストには、ユニットの実行を

制御し、誤りを観察する最も良い機能がある。しかし、それはシステム全体の振舞いの正しさに関する情報を与えない。

統合テスト

他の新しく開発された機能とともに別の機能をテストすることを統合テストと呼ぶ。統合とは、各コンポーネントがより大きなコンポーネントを作るために結合される処理という。統合テストとは、コンポーネントテストでバグが検出されなかった各コンポーネントを組み合わせ、それが正しくないことや矛盾していることを立証することである。統合テストは各コンポーネントの間で渡される変数だけではなく、グローバル変数も検査する。そのため、ユニットテストに対しては、ホワイトボックステストしか使えないが、統合テストに対しては、ホワイトボックステスト、ブラックボックステストの両方が使える。

システムテスト

システムテストは個々のコンポーネント、あるいは、コンポーネントと他のオブジェクトの相互作用によって発生する誤りを明らかにする。システムとは、一つの大きなコンポーネントを指す。システムテストとは、ユニットテスト、統合テストで検出できないバグ、すなわち、コンポーネント自身の不良、コンポーネント間の矛盾、コンポーネント同士のデータ変換不良以外の不良を摘出することを目的とする。システムテストには、性能、機密保護機能、信頼性、システム立上げ、システムエラー回復などのテストも含める。システムテストには、ブラックボックステストを使用することが多い。これは、ホワイトボックステストで要求される可能なパスの数が大きく、扱いが難しいためである。

一般的に、形式仕様は、実施非依存のように定義される。従って、形式仕様を使用したテストの自動生成では、容易に構造テストを行うことはできない。これは、ユニットテストがモデル検査により容易に扱われないことを意味する。そのため、システムレベルおよび統合レベルでの行動テストに着目する。

2.2.3 標準的なテストの選択技術

テストの自動選択技術の集合が「標準」として存在している。これらの技術については、長い時間研究されており、教科書 [2] にもまとめられている。それらのテスト技術の中でも最も重要なテスト技術は、パステスト、データフローテスト、および状態ベーステストである。

パステスト

パステストは、テストの基礎であり、構造モデルとしてプログラムの制御フローを用いるテスト技術の一つである。構造テスト方法の中で、パステストは最も古い。パステストの目的は、命令語と分岐命令を最低一回は実行するということである。なお、パスは、一連のステートメントや命令語で、合流点や判定を開始点として、別の、あるいは、同じ合流点や判定で終了する。

新しく開発するソフトウェアのユニットテストには、パステストが一番適している。パステストでは、プログラム（すなわち、ソースコード）の構造全部を知っている必要があるためである。パステストの結果の質は、ソフトウェアの大きさが大きくなるにつれて悪くなる。このような理由から、システムテストでパステストを使うことは、まれである。パステストは、プログラマーにとって最も基本的なテスト技術である。

パステストのカバレッジ基準では、全てのパス、全てのノート、または、全てのエッジに通しなければならない。理想的なパステストは、プログラムの全てのパスを網羅する。しかし、それは不可能である。これは、バグにより、不必要なパスが生じたり、必要なパスが抜けてしまうことがあるためである。例えば、20 回実行されるループから成っている小さいプログラムがあると仮定する。このループの中に、10 本のルートを割り当てる $case$ または if の命令語がある。このプログラムを通すパスの総数は 10^{20} になるため、バグの含まれる可能性が高い。

データフローテスト

データフローテストは、制御フローグラフを使って、データに発生する不正（データフロー変則）を検出するテスト技術である。具体的には、データフローテストはプログラムのデータ (すなわち、変数) のライフサイクルを追う。データ使用のパターンを捜すことによって、コードの危険地域が見つかる。データの使用には、四つの方法がある。すなわち、データは、定義され、条件文（述語文）で使用され、計算で使用され、解放されることである。

データフローテストでは、プログラムでの変数割り当ては、割り当て（定義）から変数の値が使用されるポイント（使用）までのサブパスを実行するテストの発生によって、テストされる。その「使用」には、計算における変数の使用（c-使用）と述語文における変数の使用 (p-使用) がある。c-使用は、変数が計算または出力文で使用されるたびに起こる。一方、p-使用は、変数が述語文で使用されるたびに起こる。データフロー技術は、制御フローグラフを使って、プログラムを表示する。制御フローグラフでは、各ノードは順番に実行されるステートメントに対応し、各エッジはステートメント間の制御の流れを表す。変数の定義と使用は、制御フローグラフのノードにに付けられる。そして、データフロー分析は、これらの定義と使用を使い、定義と使用（def-use）の関連を計算する。

さらに、データフローテストは、ホワイトボックステスト法の一つであり、テストの妥当性基準を提供する。この基準は、全ての定義（all-definitions）が全ての使用（all-uses）

にされることである。

イベントフローテスト

イベントフローテストは、特殊なデータのフローテストである。イベントフローは、システムの階層的な状態機械モデルを使用する。状態機械モデルは、オブジェクト指向分析および設計によく使用される概念である。このモデルは制御フローグラフが実行されるステートメントの可能なシーケンスを示す方法と同じようにイベントの可能なシーケンスを示す。しかし、データフローは変数の使用の定義に対する影響に注目するが、イベントフローは、後のイベントのシステムレベル・イベントに対する影響に注目する。

状態ベーステスト

状態ベーステストは有限状態機械 (FSM) および様々な拡張を使用する。FSM はシステムまたはオブジェクトの振舞いの仕様として使用される。また、これは、コードから得ることができる。FSM モデルは、パスカバレッジ基準を使用してテストできるが、複雑な技術を追加する必要がある。Chow[3] は、遷移のシーケンスを網羅するための技術を定義し、長さが異なるシーケンスを使用して見つけるエラーを示した。これは、移動エラー、余分な状態などである。

2.2.4 カバレッジ基準

カバレッジ基準は、ソフトウェアで行ったテストの質を測定する方法である。この測定は、プログラムでテストを行う直接的プロセスで行われる。これは、どのような部分が実行されたか、そして何よりも、どのような部分がまだ実行されていないかを示すデータを取り出す。つまり、カバレッジ基準は、どの程度実行されていない部分が残っているのかを示す。

ほとんどの開発者はこのプロセスを理解してその提案に同意し、100%[1]カバレッジを目標としている。しかし、100%カバレッジは素晴らしい目標ではあるが、100%の間違ったカバレッジは問題をもたらす場合がある。

テストカバレッジには以下の３種類がある。

パスカバレッジ

パスカバレッジは、プログラムの全ての制御フローパスを実行することである。普通は、プログラムの全ての入口/出口パスに限定する。この基準を満足すれば、100%カ

[1]普通は、「100%」という言葉はつけない。従って、「パスカバレッジを実現した」というときには、「100%パスカバレッジ」を実現したということを意味する。ステートメントカバレッジ、分岐カバレッジ、あるいは、そのはかのカバレッジについても同様である。

バレッジが成立するという。これをC0 と表す。しかし、これは、最も厳しい基準で、実現は一般的に不可能である。

ステートメントカバレッジ

ステートメントカバレッジは、少なくとも１回は、プログラムの全てのステートメントを実行することである。これが成立すると、100%ステートメントカバレッジを実現したという。この別表現には、等価の特徴をもった100%ノートカバレッジ（制御フローグラフのノートについて）がある。これをC1 と表す。これは、最も弱い基準であり、新規開発のプログラムに対するテストでは、ほとんど意味がない。

分岐カバレッジ

分岐カバレッジは、分岐命令の全ての遷移先を少なくとも１回は実行することである。これを満足すると、100%分岐カバレッジを実現したという。この別表現には、100%リンクカバレッジ（制御フローグラフの中のリンク）がある。構造化ソフトウェアでの分岐テスト、すなわち、分岐カバレッジテストは、ステートメントカバレッジを完全に包含する。分岐カバレッジはC2 で表す。

典型的なソフトウェア開発では、ステートメントまたは分岐のカバレッジを満足することに注目する。しかし、100%のステートメントまたは分岐カバレッジが成立しても、重大なバグが残る可能性がある。これは、ステートメントおよび分岐カバレッジでは、システムの論理が実行されたかどうかわからないからである。このように、実際のソフトウェアテストでは、ステートメントと分岐のカバレッジだけでは不十分である。そこで、もっと強い理解技術としてパスカバレッジに着目する。ただし、パスカバレッジが成立することは不可能であるので、パスカバレッジはバグを明らかにすることを助けることを意味する。

2.3　テストケースの基礎概念

ソフトウェアテストのうちでは、テストケースは、動的テストと見なされ、有効なアプローチである。IEEE Standard 610（1990 年）ではテストケースを次のとおり定義している。

(1) プログラムのあるパスを実行する、または、システムのある要求を満足するか確認するなどの特定の目的のために作成されるテスト入力、実行条件、期待される結果のセットである。

(2) （IEEE Std 829-1983）一つのテスト項目のための入力、予測される結果、および、実行条件を記述する文書化。

もっと簡単に言えば、テストケースとは、システムにどのような入力を与え、その結果としてどのような出力が得られるべきかを記述したものである。テストケースはプログラ

ムに尋ねる質問である。テストケースを実行するとき、プログラムがテストケースに合格するか、失敗するかどうかの情報が得られる。

2.3.1 テストケースの記述

テストケースの構造

テストケースの形式的な文書は 3 部で構成される。

- 情報
  情報はテストケースの概説から成る。情報は、テストケースの識別子、作者、版、名前、目的、簡単な説明、および、依存関係を含む。
- アクティビティー
  アクティビティーは実際のテストケースのアクティビティーから成る。アクティビティーは、テストケースの環境、テストケース初期化で実行されるべきアクティビティー、テストケースを行ったあとで実行されるべきアクティビティー、テストしている間で実行されるべき段階的なアクション、および、テストのために供給される入力データを含む。
- 結果
  結果は行われたテストケースの成果である。結果データは期待される結果および実結果の情報から成る。

テストケースの設計

テストケースはテストされる機能または技術を理解する設計者によって設計され、書かれるべきである。テストケースは次の情報を含むべきである。

- テストの目的
- ソフトウェア要求とハードウェア要求
- 特定の設定と構成必要条件
- テストを行う方法の記述
- 期待される結果とテストの成功基準

テストケースを設計することはテストスケジュールで時間がかかるが、不必要な再テスト、または、デバッギングを避けることができ、また少なくとも、そのような手間が減少されるので、手間を掛けるに値する。

表 2.1: 「Correct Login」というテストケースの例

Test Case ID	Correct Login
Functionality	Baseline
Use case reference	Authenticate User
Distribution	JBoss 3.2.3
Test Sequence	1.Execute Stitch Application 2.Enter user name = “ neel ” 3.Enter password = “ passwd ” 4.Click on Login button 5.Search auction page is displayed
Configuration/ Deployment Issues	Currently the system is only working in our development environment and has not been packaged for deployment.
Known Problems	

テストケースの例

テストケースの例を、表 2.1 に示す。

2.3.2 テストケース生成法

手動テストのコストを削減し、その信頼性を高めるため、研究者および実行者はテストケース生成の自動化に取り組んでいる。これまでに、いくつかのテストケース生成アプローチが提案されている。

モデルベーステスト

モデルベーステスト（Model Based Testing または MBT）とは、テストケースの一部または全部をテスト対象のシステム（System Under Test または SUT）の（通常、機能的側面を）モデル化したものから導出して行うソフトウェアテストの手法である。モデルは SUT の実現すべき振舞いを表現した抽象的なものである。そのようなモデルから導出されるテストケースはモデルと同レベルに抽象化された機能テストである。これは、抽象的なテストケースであり、直接実行することはできない。そのため、抽象的なテストケースを実行可能なテストケースに変換する必要がある。

MBT の主な有効性は、自動化の可能性にある。モデルが十分に形式的であれば、基本的にテストケースを自動的に抽出することができるはずである。一般的に、モデルは有限オートマトンに変換される。このオートマトンは SUT のとりうる構成を表現している。テストケースを選択するには、このオートマトン上で実行可能パスを探す。一つの実行可

能パスが 1 つのテストケースに対応することになる。テストケースの選択には、以下に示したような様々な手法が適用される。これらの技術は、MBT のテストケース選択法と考えられることもあれば、別のテストケース生成法とと考えられることもある。

定理証明によるテストケース生成

定理証明（Theorem proving）は本来、論理式の自動的な証明に使われるものであった。モデルベーステストでは、システムを一群の論理式でモデル化してシステムの振舞いを記述する。テストケースを選択するには、SUT を記述した論理式群を翻訳したものを同値なクラスで分類する。各クラスは特定のシステムの振る舞いを表現しており、テストケースと対応させることができる。簡単な分類法として論理和標準形を使った手法がある。システムの振る舞いを表す論理式を論理和標準形に変換するのである。

モデル検査によるテストケース生成

モデル検査は、モデルが仕様を満たすかどうか確認するための技術として開発された。モデルチェッカは、SUT のモデルと検証したい性質を与えると、その性質がモデル上で正しいかどうかを証明する過程で証拠と反例を検出する。証拠となるパスはその性質を満たす。一方、反例となるパスを実行すると、その性質に反した状態となる。これらのパスがテストケースとして使用できる。

記号的実行によるテストケース生成

記号的実行（Symbolic Execution）を MBT のフレームワークとして利用することがある。これは抽象モデルでの実行経路を追跡する手段である。基本的にプログラム実行を実際の値を使わずに、変数を記号として扱ってシミュレートする。各実行パスが可能なプログラム実行パスを表し、テストケースとして利用できる。その際に記号に値を設定してインスタンス化する。

第3章　適合テスト方法とテストケース自動生成法の調査

関連研究として適合テスト方法とテストケース自動生成法について調査を行った。適合テスト（Conformance testing）とは、システムの実装が仕様に合うかどうかテストすることである。適合テスト方法では、オートマトン理論に基づいた適合テスト方法（または、FSM に基づいた適合テスト）がよく知られ、W-方法、T-方法、U-方法、D-方法 [4] などがある。これらの中で、W-方法 [3] が最も古く、ほかの FSM に基づいた適合テスト方法の基礎となった。一方、テストケース自動生成法については、モデルベーステストおよびモデル検査の反例に基づいたテストケース生成法の研究が行われている。

3.1　モデルベーステスト

モデルベーステスト（MBT）は SUT のモデルに基づいて、テストケース生成、テスト結果の評価などのテストタスクを行うアプローチを示す一般用語である。MBT は、電話交換機などのハードウェアテストにルートアプリケーションがあるが、最近では色々なソフトウェアドメインに広がっている。

3.1.1　モデル

SUT のモデルは、明示的モデルである。明示的モデルの使用はテストケースを得るプロセスが非構造化で、再現できなく、文書化されない観察から動機を与えられる。この考えはシステムのテストされる振舞いを明示的に符号化するアーティファクトが存在することである。明らかに、SUT のモデルはそれ自身に妥当性がなければならない。妥当性ということは、システムが正しいかどうかを検証することではなく、正しいシステムを造ることである。モデルの正当性を保証することは相互活動であり、要求の一貫性および完全性が吟味されることを意味する。

MBT では、モデルの妥当性は、モデルは SUT より簡単であるし、または少なくとも、SUT よりチェックしやすく、変更しやすく、維持しやすいことを意味する。つまり、モデルは、状態としない振舞いを省くことができ、SUT の抽象化である。そして、モデルは、意味のあるテストケースの生成の基礎となければならない。そこで、「抽象化」という用語は、細部の省略および細部のカプセル化の両方とも表示する。

3.1.2 プロセスと用語

MBT の一般的なプロセスは、モデルの作成、テスト選択基準の定義、テストケースの生成、および、SUT でのテストを含む（図 3.1）。このプロセスは、次のように進む。

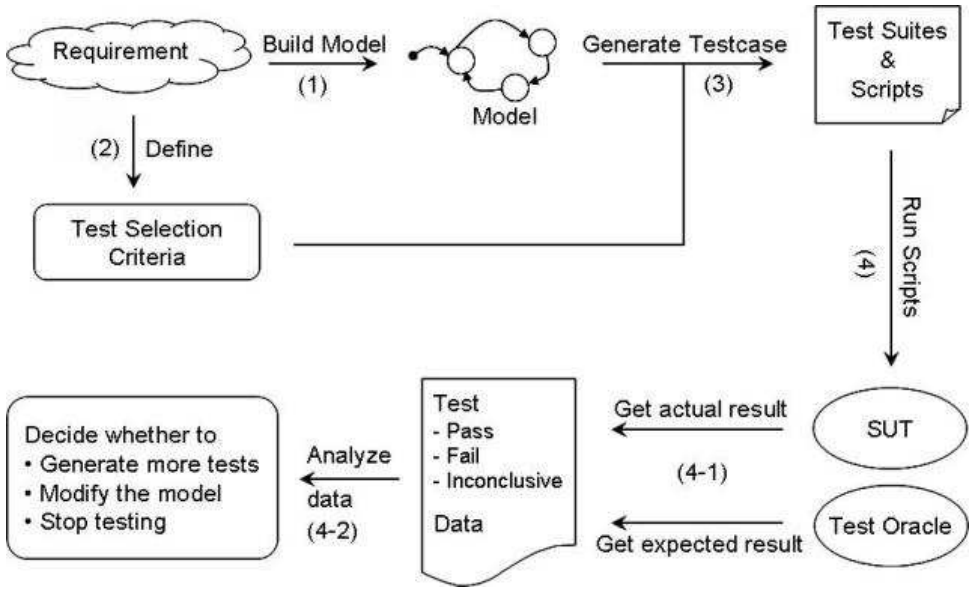

図 3.1: MBT の流れ

ステップ１. SUT のモデルは要求、または、仕様書に基づいて作られる。このモデルは、対象とする振舞いを符号化し、様々な抽象化のレベルに存在することができる。実際、様々な SUT のモデルがあり、それぞれはソフトウェアの振舞いの様々な様相を記述する。例えば、有限状態機械（FSM）、状態図（State chart）、統一モデリング言語（Unified Modeling Language または UML）、または、マルコフ連鎖（Markov chain）などである。

ステップ２. テスト選択基準が定義される。一般的に、「良いテストケース」を定義することは困難である。良いテストケースとは、受諾可能なコストで誤りを検出できるものである。テスト選択基準はシステムの与える機能 (要求によるテスト選択基準)、または、モデルの構造 (状態カバレッジ、遷移カバレッジ、def-use カバレッジ) と関連付けることができる。

ステップ３. モデルおよびテスト選択基準があれば、テストケースを生成する。テストケース生成法は、第 2 章に示したように、定理証明によるテストケース生成、モデル検査によるテストケース生成、記号的実行によるテストケース生成法などがある。

一方、タイミングに関連して、MBT のテストケース生成法は、オンラインテスト（on-line testing）とオフラインテスト（off-line testing）に分けられる。

オンラインテストを使えば、テストケース生成アルゴリズムは、SUT の実際の出力に反応できる。SUT が非決定的ならば、これは必要である。これは、オンラインテストの生成ツールは、SUT が取ったパスがモデルのパスと同じであるかどうか見えるためである。

オフラインテストは、実行する前にテストケースが完全に発生されることを意味する。非決定性モデルからのオフラインテスト生成はもっと難しく、シーケンスよりはむしろ木またはグラフによってテストケースを生成することを含む。オフラインテストの利点は、適用する場合は、実用的な点である。発生させたテストケースは、既存のテスト管理ツール上で管理され、実行することができる。テストプロセスはあまり変わらない要求がある。テストケースの集合を生成してから、何回も SUT でこれを実行することができる。また、テストケース生成、および、テスト実行は、異なる機械、異なる時刻、または、異なる環境で行うことができる。そして、テストを実行する前に、生成したテストケースの集合を最小化することが可能である。また、テストケース生成プロセスがテスト実行より早ければ、テストケース生成のフェーズに確かに明らかな利点があると言われる。

ステップ４. テストケースが生成されると、テストケースでテストを行う。このステップは、２つの段階を含む。

ステップ４－１. テストケースの実行は、テストケースの入力を SUT に適用し、SUT の出力を記録する活動である。この出力は、実際の出力となる。

ステップ４－２. 全てのテストケースを実行してから、判定を行う。判定は、テストケースの期待された出力と実際の出力を比較する作業である。成功、失敗、または、決定的でない判定がある。実際の出力が期待された出力と整合すれば、テストが成功したという。期待された出力が実際の出力と整合していなければ、テストが失敗したという。判定が下せないときは、決定的でない判定という。そのあと、全ての判定データを分析し、評価する。評価は、もっとテストケースを生成する、モデルを修正する、または、テストを終えるなどである。

3.2 モデル検査の反例に基づいたテストケース生成法

モデルチェッカは、モデルが性質に違反するかどうか確認する。モデルチェッカは、全ての到達可能な状態を追跡する。モデルチェッカが違反を検出しなければ、モデルは性質を満足する。しかし、モデルチェッカは性質に違反する到達可能な状態を見つければ、反例を出す。この反例は、初期状態から性質に違反する状態までの到達可能状態シーケンスである。その一方、モデル検査によるテストケース生成法では、モデルチェッカは、期待される出力、および、反例を生成するために使用される。また、一つの反例は一つのテストケースとして使われる。よって、テストケースを生成することは、反例を生成することになる。

この方法では、反例を生成するために、変異分析（Mutation Analysis）を利用する。変異分析は、ソースコードの小さな変更を含むソフトウェアテストの方法である。変異分析は、変異演算子（Mutation Operators）の集合を定義する。それぞれの変異演算は、小さな構文変更のパターンである。変異分析では、変異演算子に基づいて、ユーザーの典型的な誤り (例えば、間違った演算子と変数名の使用) をまねたり、貴重なテストの作成 (例えば、ゼロ除算) を強制したりする。変異分析の目的は、有効なテストを生成するか、または、実行中に決してアクセスされない誤りを見つけることを助けることである。

3.2.1 プロセス

モデル検査の反例に基づいたテストケース生成法の全面的なフローを図 3.2 に示す。

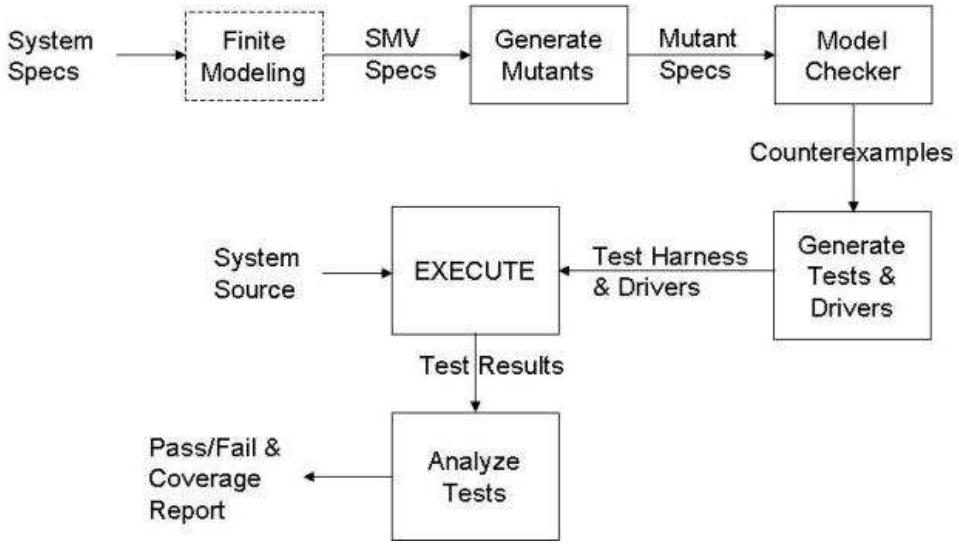

図 3.2: モデル検査の反例に基づいたテストケース生成法の流れ

- 初めに、システム仕様書から、有限モデリング（finite modeling）またはツールを使って、モデル検査に適した仕様に変換する。従来のモデル検査の反例に基づいたテストケース生成法では、SMV（Symbolic Model Verifier）というモデル検査ツールが使われていたので、モデル検査に適している仕様は、SMV に適している仕様であり、CTL（Computational Tree Logic）で表現される。このステップの後、全ての処理を自動化することができる。
- 変異演算子を仕様に適用して、変異仕様を作成する。変異仕様は、モデルが充足できない仕様である。モデル検査は一つの変異仕様を処理すると、一つの反例を出力する。反例を作成すると、反例の集合を最小化する。反例は入力と期待値の両方を含み、テストケースに自動的に変換することができる。

- 最後に、テストケースをソースコードに適用する。プログラムの実行結果を分析し、テストが成功したかを判定する。また、生成したテストケースのテストカバレッジを評価する。

3.2.2 変異演算子

モデル検査の反例に基づいたテストケース生成法において、変異演算子の定義は、重要なタスクの一つである。変異演算子は、モデルと仕様の不整合を検出するので、モデルの変更、および、仕様の変更の２つカテゴリーに分けられる。

モデルの変更. 仕様が正しいと仮定し、モデルの誤りをまねるように、モデルを変更する。すなわち、モデルの変更によるテストケースを実行するとき、正しい実装は、反例が出力する結果と異なる結果を出力するべきである。これは失敗テスト（Failing Test）と言われる。

仕様の変更. モデルが正しいと仮定し、仕様変異を生成する。このカテゴリーでは、反例が正しいモデルから出るので、正しい実装は反例が示した結果のシーケンスを追うべきである。これはパステスト（Passing Test）と言われる。

3.3 W-方法

W-方法は、FSM に基づいて仕様と実装の整合性をテストする手法の一つであり、制御構造にしか適用されない。W-方法では、仕様だけではなく、実装も、決定性 FSM でモデル化されることを前提とする。よって、仕様と実装の整合性をテストすることは、２つの FSM の整合性をテストすることを意味する。実装 FSM のある振舞いは、仕様の指定された振舞いと異なっていれば、実施は誤りを含んでいる。

W-方法は、三つの主要なステップを含む（図 3.3）。

1) 仕様と実装をモデル化する。ここで、仕様は、FSM S（Specification FSM）、実装は、FSM M（Implementation FSM）に変換される。
2) FSM S に基づいて、テストケースを生成する。
3) テストケースを実行すると、FSM S の出力（期待される出力）と FSM M の出力（実際の出力）を得る。これらの出力を分析し、FSM S は、FSM M と整合しているかどうか判定する。

3.3.1 オートマトン理論

W-方法で検討されるソフトウェアシステムには、２つの基本要素がある。刺激と操作である。刺激は、ソフトウェアの環境からの入力である。操作はソフトウェアによって引

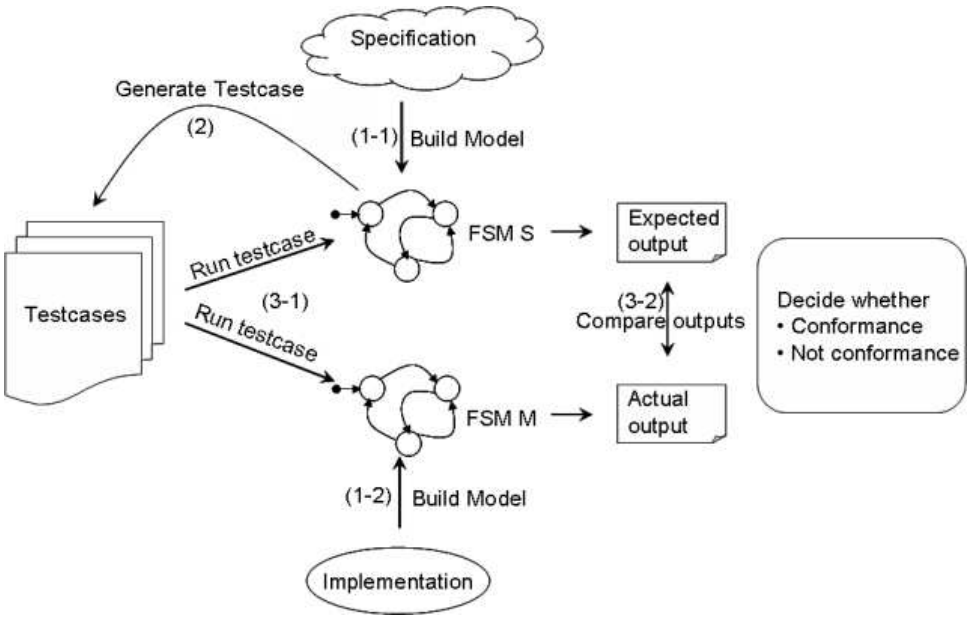

図 3.3: W-**方法のプロセス**

き起こされ、刺激に応答する出力である。例えば、字句解析、データ処理、リアルタイムプロセス制御などは、そのソフトウェアシステムの例である。そういうソフトウェアシステムを FSM **でモデル化するために、ミーリ・マシン（**Mealy Machine**）という** FSM **を用いる。**

ミーリ・マシンとは、出力が現在状態と入力によって決定される FSM **である。つまり、状態遷移図で描くと遷移エッジには入力記号と出力信号が付記される。図** 3.4 **にミーリ・マシンの例を示す。例えば、入力** '#' **を受けて状態** 1 **から状態** 2 **に遷移する際に、**'A' **が出力される（エッジには**'#/A' **と表示される）。形式的に、ミーリ・マシンは、**$(S, I, O, \delta, \lambda, s_0)$ **の６組として定義される。だだし、**

- S **は、状態の集合である。**
- I **は、入力記号の集合である。**
- O **は、出力記号の集合である。**
- δ **は、遷移関数である。** $(\delta : S \times I \rightarrow S)$
- λ **は、出力関数である。**$(\lambda : S \times I \rightarrow O)$
- s_0 **は、初期状態である。**$(s_0 \in S)$

そして、次の表記法と定義を利用する。

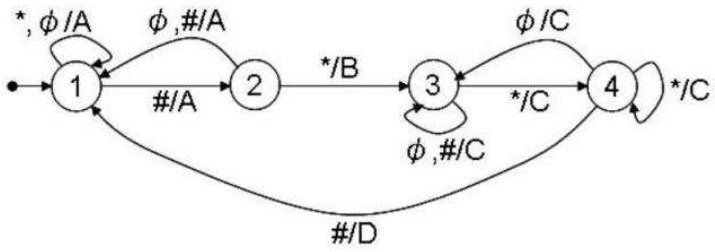

図 3.4: **ミーリ・マシンの例**

$M \mid s$ $\equiv$ M **の状態** s **のときである。**
$M \mid s < \alpha >$ $\equiv$ $M \mid s$ **のときに、入力シーケンス** α **を入力すると出力される出力シーケンスである。**

***completely specified* の定義.** *FSM* M **では、任意の状態** s **と入力** x **に対して、遷移が存在すれば、** M **は完全に規定される。つまり、** δ **と** λ **は全関数である。**

ミニマルの定義. *FSM* M **の状態数は、** M **と等しい全ての** *FSM* **の状態数以下であれば、** M **はミニマルである。**

状態識別の定義. *FSM* M **の状態** s_i **と** s_j **に対してある入力シーケンス** α **が存在し、** $M \mid s_i < \alpha > \neq M \mid s_j < \alpha >$ **を満たせば、** s_i **と** s_j **は識別される。**

適合の定義. 任意の入力シーケンスに対して、2つの *FSM* S **と** M **の初期状態を識別しなければ、** M **は** S **と適合している。**

W-**方法で** FSM **は、**1) **完全に規定され（**completely specified**）、**2) **ミニマル（**minimal**）であり、**3) **初期状態が決まり、**4) **全ての状態は到達可能であることを前提とする。**

W-**方法の実装は、ブラックボックスであり、内部状態が見えないため、環境からの刺激に対応する操作しか観察できない。そのため、仕様と実装の整合性をテストするには、特定の入力シーケンスに対応する出力シーケンスを用いて実装の状態が仕様の特定の状態と識別しているかどうか確認する。すなわち、仕様と実装の初期状態を識別するかどうか確認する。具体的には、**W-**方法で生成されるテストケースは、入力シーケンスである。テストケースを実行することは、生成した入力シーケンスに対応する出力シーケンス（期待される出力と実際出力）を得て、これらを分析し、実装の初期状態が仕様の初期状態と一致するかどうかを判定する。**

3.3.2 テストシーケンス生成

W-方法のテストケースは、入力シーケンスであるので、W-方法は、テストシーケンス生成法と呼ばれる。テストシーケンスのセットはシーケンス集合 P と集合 W の連結である。

P の作成. P は、FSM S の全ての遷移を網羅する入力シーケンスの集合である。P を作る方法の一つは、FSM S のテスト木（Testing Tree）を生成する方法である。このテスト木の全ての部分パスは P を構成する。一つの部分パスは、連続的枝のシーケンスである。これは、テスト木のルートから終端節点または非終端節点までのパスである。テスト木の各枝には入力記号が表示される。そのため、P は、入力シーケンスの集合である。また、空シーケンスも P の要素である。次に、FSM S からテスト木 T を生成する手続きを示す。

1) T のルートのラベルは、FSM S の初期状態である。これは、T のレベル 1 である。
2) T のレベル k を生成したと仮定する。T のレベル $(k+1)$ は左から右へ第 k レベルの節点を検討して、生成される。第 $(k+1)$ レベルの節点のラベルが第 j レベル $(j <= k)$ の非終端節点のラベルと同じならば、終端節点である。そうでなければ、第 $(k+1)$ レベルの節点のラベルを A_i とする。任意の入力記号 x に対して、$\delta(A_i, x) = A_j$ が存在すれば、節点 A_i に一つの枝と子ノードを付ける。付けた枝のラベルは x、子ノードのラベルは A_j である。

FSM S では、有限の状態が存在するので、上記のプロセスは常に終わる。また、このプロセスでは、幅優先探索によってテスト木 T を生成する。

集合 W. W は FSM S のキャラクタリゼーションの集合（characterization set）である。W は、FSM S の任意の２つの状態を識別する入力シーケンスの集合である。形式的に、集合 W は以下の通り定義される。

W-集合の定義. *FSM S* の $W = \{\alpha_1, \alpha_2, ..., \alpha_n\}$、ただし、$\alpha_i \in I_S \quad (i = \overline{1,n})$、つまり $W \subset I_S$
FSM S の任意の２つの状態 s_i と s_j に対して、$(\lambda(s_i, \alpha_1), \lambda(s_i, \alpha_2), ..., \lambda(s_i, \alpha_n)) \neq (\lambda(s_j, \alpha_1), \lambda(s_j, \alpha_2), ..., \lambda(s_j, \alpha_n))$ である。

集合 W が存在することは、W-方法の前提条件である。オートマトン理論によって、FSM がミニマルであれば、任意の２つの状態を識別できる。W-方法の前提では、FSM がミニマルであると仮定するので、集合 W は常に存在する。

そして、P と W の連結でテストケースの集合を生成する。すなわち、テストケースの集合は、$P \cdot W$ 集合である。例として、図 3.5(a) は、図 3.4 の FSM のテスト木であり、図 3.5(b) の集合はその FSM の集合 W である。また、テストシーケンスの集合を図 3.5(c) に示す。

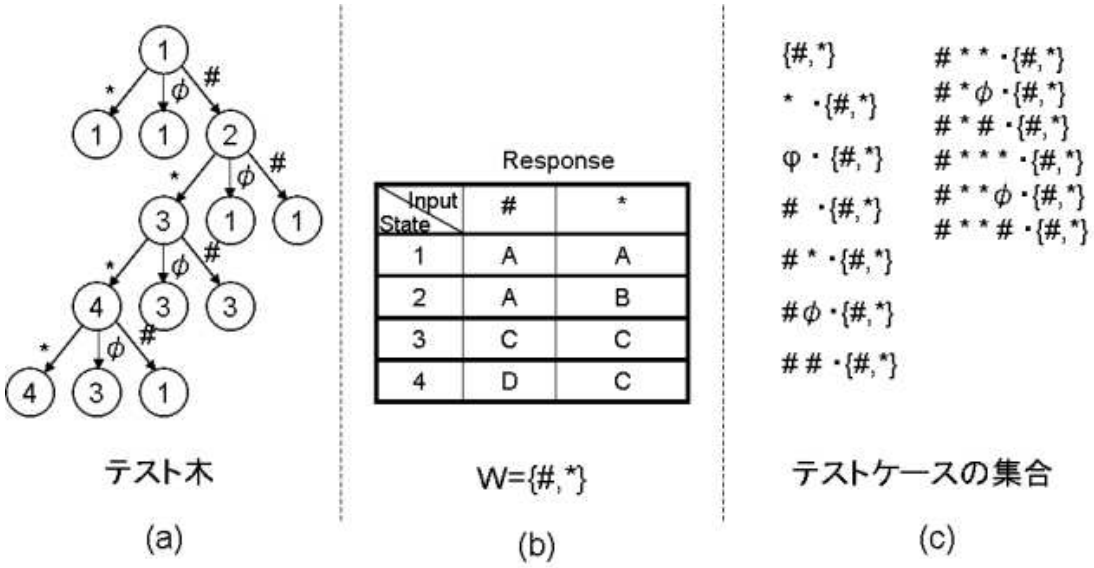

図 3.5: (a) **テスト木**　(b) **集合** W **への状態応答**　(c) **テストシーケンスの集合**

3.3.3　テストカバレッジ

生成したテストケースの質を評価するために、W-**方法ではシステムのエラーを三つのエラークラスに分けている。操作エラー（**Operation Error**）、遷移エラー（**Transfer Error**）、および、余分状態・欠状態エラー（**Extra/Missing State Error**）である。図** 3.6 **でこれらのエラーの例を示す。**A **と** A' **は、ミニマル** FSM **であるとする。**A **は、「基準」として正しい版であり、**A' **は、**A **と比較するものである。**

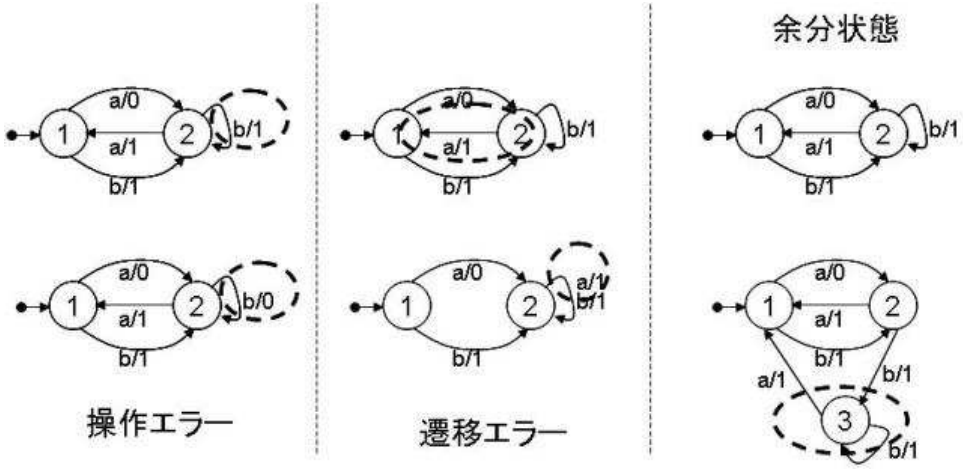

図 3.6: **配列決定のエラーの例**

1) **操作エラー**. A' **が** A **と適合しない場合、出力関数** λ **だけを整理すると** A' **は** A **と適合するようになる。そのとき、** A' **には、操作エラーがあると言われる。**

2) **遷移エラー**. A' **が** A **と適合しない場合、遷移関数** δ **だけを整理すると、** A' **は** A **と適合するようになる。そのとき、** A' **には、遷移エラーがあると言われる。**

3) **余分状態・欠状態エラー**. A' **が** A **と適合しない場合に、** A' **の状態数が** A **の状態数より多いか少ないとき、** A' **には、余分状態・欠状態エラーがあると言われる。**

これらのエラーは、配列決定のエラーと呼ばれる。W-**方法のテストケースの前半はテスト木のパスであるため、操作エラーを見つけられ、後半は集合** W **の要素なので、遷移エラーを見つけることができる。しかし、**W-**方法では、余分状態・欠状態エラーを完全に見つけるとは言えない。**

3.4 従来手法の問題点

テストケース生成には、以上に示した各方法のような利点がある。しかし、これらの手法には、問題がまだ残る。

MBT **はテスト実施者の技術に依存する。テスト実施者は、モデルおよびモデルの基礎的数学と理論に詳しくなければならない。**MBT **のモデル構成は、労働集約的な仕事であり、大きな障害となる可能性がある。テスト実施者の考えによって異なるモデルが生成され、誤りが入る場合もある。すなわち、**MBT **のモデルは、正しさが保証されてない。そのため、テストケースでテストを行う結果を分析する際に、テスト結果が良くなければ、テストケース生成アルゴリズムだけではなく、モデルも再考する必要がある。テストケース生成アルゴリズムか、モデルか、またはその両方ともを修正する場合がある。**

モデル検査の反例に基づいたテストケース生成法は、変異演算子の集合に依存する。定義される変異演算子の集合によって、生成したテストケースの質は異なる。実際に、モデルの全ての到達可能な状態シーケンスを網羅する反例の集合を生成するように変異演算子の集合を定義することは困難である。つまり、モデル検査の反例より生成されるテストケースは不完全である。

W-**方法は、適合テストの良いアイディアを提供している。システムの仕様と実装の両方を** FSM **に変換することを前提とする。そのため、仕様と実装の整合テストは、２つの** FSM **の整合性をテストすることを意味する。**W-**方法は、操作エラーと遷移エラーを見つけられるが、余分状態・欠状態エラーを見つけられない場合がある。そして、システムの仕様または実装を表示する** FSM **は、全ての前提条件を満足しなければならない。すなわち、**FSM **は、完全に規定され、ミニマルであり（つまり、集合** W **が存在すること）、初期状態を決め、全ての状態に到達できなければならない。しかし、全ての条件を満足する** FSM **を生成することは、簡単ではなく、**W-**方法の主な障害となる。**

第4章　Promela記述に基づいたテストケース生成法

本研究の目的は実装が設計モデルで検証した性質を満たすことを保証するために、テストケースを用いて実装と設計モデルの整合性をテストする枠組みを提案することである。この整合性は、$\forall p(D \models p \leftrightarrow M \models p)$ が成立すること同値である。ただし、D はモデル検査の設計モデル、M は実装をモデル化したもの、p は、設計モデルで検証された性質である。また、設計モデルはモデル検査により正しさが検証され、実装は検証した設計モデルに基づいて行われることを前提とする。すなわち、本研究は、設計モデルからテストケースを自動生成し、このテストケースに基づいて設計モデルと実装の整合性をテストする手法を提案する。

本提案手法は、MBT 手法の問題を解消する。すなわち、設計モデルからテストケースを形式的に生成するので、一つの設計モデルは、一つのテストケースの集合にしか対応しない。そして、設計モデルの正しさが検証されているので、テスト結果が良くなければ、テストケース生成アルゴリズムだけを再検討すれば良い。また、本提案手法は W-方法を参考にし、W-方法の問題を解決方法を提供する。W-方法では、実装の内部状態の数を見積もることができなければ、仕様と実装の整合性を保証することができない。そこで、本研究は、実装がホワイトボックスであり、内部状態が見えることを前提とする。よって、W-方法で使用されるテストケース生成アルゴリズムを変更する必要がある。

今回は、モデル検査ツール Spin を使用した設計モデルの検証を対象とするため、本研究の目的は、Promela で記述される設計モデル（以下、単に「Promela 記述」）からテストケースを自動的に生成し、このテストケースに基づいて Promela 記述と実装の整合性をテストする手法を提案することになる。そのため、Spin の探索機能を用いて、Promela 記述からテストケースを自動生成する手法を提案する。そこで、W-方法で使用される FSM を再定義する必要がある。

本章では、これら一連の Promela 記述に基づいたテストケースの自動生成法の流れについて説明する。

4.1　FSMの再定義：オートマトン理論

本研究では制御システム（Control System）のみに着目する。制御システムには２つの基本要素がある。刺激と操作である。刺激は、システムの環境から発生されるイベント

（入力）である。操作は刺激に応答する内部システムでの処理（出力）である。言い換えれば、制御システムは、内部システムがもらった刺激によって動く。本研究は W-方法で使われる FSM を参考にして制御システムを表現する。

Promela 記述からのテストケース生成を自動化するため、W-方法での FSM を再定義する。すなわち、FSM の状態を Promela で定義される状態のように再定義する。

Promela で定義される状態. Spin の理論では、Promela で定義される状態は、Promela での変数の集合、および、PC（プログラム・カウンター）の組合わせの状態である。しかし、Promela 記述と実装の整合性をテストする際に、Promela での全ての状態ではなく、Promela での制御状態のみに注目する。一方、Promela でのラベルは、内部プロセスにおける独特な制御状態を特定する。そのため、Promela で定義される（変数の集合、ラベル）組の状態は、Promela 記述と実装の整合性をテストすることに必要な状態である。

よって、FSM の状態を（ラベル、変数の集合）の組の状態と再定義する。つまり、FSM の状態は、ラベルと全ての変数値の組合わせである。そのため、FSM では、状態集合 S の代わりにラベル集合 L と変数集合 V が使用される。すなわち、FSM の状態集合 S は、$L \times V$ 集合で表現される。ただし、V に含まれる変数 v は、（名前、値域）の組で表現される。

FSM の再定義. Promela 記述と実装から変換される FSM は、以下のように定義される。

FSM の再定義. *FSM* M は、ミーリマシンであり、$(L, V, I, O, \delta, \lambda, s_0)$ の７組で記述される。だだし、

- L はラベルの集合、V は変数の集合である。
  S は M の状態集合とすると、$S = L \times Dom(v_1) \times Dom(v_2) \times ... \times Dom(v_n)$ である。
  - $Dom(v_i)$ は、$V \ni v_i$ の値域 $(i = \overline{1, n})$ である。
  - $\forall v \in V$ に対して、$v = (v\_name, v\_domain)$
    ただし、v_name：v の名前、v_domain：v の値域
  - $\forall s \in S$ に対して、$s =< l, a_1, a_2, ..., a_n >$
    ただし、$l \in L$、a_i は v_i の値、$v_i \in V \quad (i = \overline{1, n})$
- I は、入力記号の集合である。
- O は、出力記号の集合である。
- δ は、遷移関数である。 $(\delta : S \times I \rightarrow S)$
- λ は、出力関数である。$(\lambda : S \times I \rightarrow O)$
- s_0 は、初期状態である。$(s_0 \in S)$

実装の内部状態が見えるので、２つの状態の識別するときに、出力関数 λ だけではなく、遷移関数 δ も用いる。そのため、テストケースと FSM の整合性を再定義する。また、後の解説で使われる表記法についてもここでを定義しておく。

$\lambda < s, \alpha >$ $\equiv$ 状態 s には、入力シーケンス α を入力すると出力される出力シーケンスである。

$\delta < s, \alpha >$ $\equiv$ 状態 s には、入力シーケンス α を入力すると届く状態シーケンスである。

テストケースの定義.
FSM D から生成されるテストケースは、$< s_0, (x_1, y_1, s_1), (x_2, y_2, s_2), ..., (x_n, y_n, s_n) >$ で定義される。ただし、s_0 は D の初期状態、$x_i \in I_D$、$y_i \in O_D$、$s_i \in S(D)$ $(i = \overline{0, n})$ である。また、このテストケースは、以下の条件を満たす。

1) $y_i = \lambda(s_i, x_i)$ $(i = \overline{1, n})$

2) $s_{i+1} = \delta(s_i, x_i)$ $(i = \overline{1, n})$

状態識別の定義. *FSM* M の状態 s_i と s_j に対して、ある入力シーケンス α が存在して、

- $\lambda < s_i, \alpha > \neq \lambda < s_j, \alpha >$、または
- $\delta < s_i, \alpha > \neq \delta < s_j, \alpha >$

を満たせば、s_i と s_j を識別すると言う。

適合の定義. 任意の入力シーケンスに対して、２つの *FSM* S と M の初期状態を識別しなければ、M は S と適合している。

4.2 Promela 記述と実装の整合性をテストする枠組み

W-方法で提供された適合テストのアイディアを参考にして、Promela 記述と実装の整合性をテストする枠組み（図 4.1）を提案する。この枠組みは、３つの重要なステップを含む。

ステップ１. Promela 記述を FSM で表現する。Promela 記述を表現する FSM を D とする。このステップは、Promela 記述を FSM で表現する規則を提供する。また、今回は、本研究は内部プロセスが一つしかない Promela 記述のみに着目する。今後は、複数の内部プロセスを含む Promela 記述についても検討を行いたいと考えている。

図 4.1: Promela **記述に基づいたテストケース生成法の枠組み**

ステップ２．実装を FSM **でモデル化する。実装を表現する** FSM **を** M **とする。本研究は、実装がホワイトボックスであり、内部状態が見えることを前提とする。また、この実装は、Ｃコードで表現される。実装の内部状態を監視するため、テストドライバ（**Test Driver**）を用いる。テストドライバは、**SUT **のプログラム（ソースコード）を呼び出すためのプログラム（ソースコード）である。すなわち、実装するときに、実装の内部状態を抽出するテストドライバを実装（Ｃプログラム）に埋め込む。実装を実行すると、実装の内部状態のデータが出力される。このデータを** FSM **に変換する。実装から唯一の** FSM **に変換するために、内部状態データの取り出し方について基準化する。また、実装で生成したテストケースを実行するため、実装の出力を標準化する必要がある。つまり、実装を** FSM **でモデル化する規則、および、テストドライバの書き方を基準化する。**

ステップ３．２つの FSM M **と** D **の整合性をテストする。**M **と** D **の整合性は、**D **で満たされた任意の性質** p **に対して、**$(D \models p \leftrightarrow M \models p)$ **である。つまり、**$D = M$ **である。そこで、本研究は、**$I_M = I_D$ **と** $S(M) = S(D)$ **を前提とし、**W**-方法を参考にして** $M = D$ **をテストする手法を提案する。ただし、**I_M **は** M **の入力記号の集合、**I_D **は** D **の入力記号の集合、**$S(M)$ **は** M **の状態集合、**$S(D)$ **は** D **の状態集合である。これらの前提は、ソフトウェアが設計モデルに基づいて実装されていると仮定している。これらの前提が成立するため、実装することは、**Promela **記述に基づいて、**$I_M = I_D$ **と** $S(M) = S(D)$ **があるように行われる。**

提案した枠組みは、提案手法の基礎理論である。これから、この基礎理論の３つのステップについて詳しく説明する。4.3 **節は「**Promela **記述から** FSM **への変換」、**4.4 **節は「実装から** FSM **への変換」、**4.5 **節は「２つの** FSM **の整合テスト」について具体的に説明する。また、この基礎理論に基づき、**Spin **で使用された探索機能を用いて** Promela **記述**

からテストケースを自動生成することができる。この自動生成の技術については 4.6 節で説明する。

4.3 Promela 記述を FSM で表現

Promela 記述を表現する FSM D は、$(L_D, V_D, I_D, O_D, \delta_D, \lambda_D, s_{0D})$ の 7 組で記述される。FSM は状態と遷移で特徴付けられるので、Promela 記述上で FSM の状態と FSM の遷移を定義する。これらの定義に基づいて、Promela 記述から FSM に変換する規則を提供する。

4.3.1 Promela 記述上の FSM の状態

FSM の状態は、(ラベル, 全ての変数の値) の組で表現される。よって、Promela 記述上で FSM の状態を定義するため、Promela 記述上で FSM のラベルと FSM の変数を定義する必要がある。

Promla 記述上の FSM のラベル. FSM の一つのラベルは、Promela 記述での一つのラベルに対応する。すなわち、任意のラベル l に対して、l が Promela 記述に存在すれば、l は FSM のラベルである。形式的に、
$\forall l(l \in L_{Promela} \Leftrightarrow l \in L_D)$、ただし、

$L_{Promela}$：Promela 記述でのラベル集合

つまり、$L_D = L_{Promela}$

Promela 記述上の FSM の変数. FSM の一つの変数は、Promela 記述での一つのグローバル変数に対応する。すなわち、任意の変数 v に対して、v が Promela 記述でのグローバル変数であれば、v は FSM の変数である。形式的に、
$\forall v(v \in V_{Promela} \Leftrightarrow v \in V_D)$、ただし

$V_{Promela}$：グローバル Promela 記述での変数集合

$V_{Promela} \ni v$ は、(v の名前, v の値域) で表現される。

つまり、$V_D = V_{Promela}$

そのとき、FSM の状態 s は、$< l, a_1, a_2, ..., a_n >$ で表現される。ただし、$l \in L_D$、a_i は v_i の値、$v_i \in V_D \quad (i = \overline{1, n})$ である。
FSM の状態集合を S とすると、$S(D) = L_D \times V_D$ である。

4.3.2 Promela 記述上の FSM の遷移

制御システムは内部システムが環境からもらったイベントによって動くので、システムの各遷移は内部システムがもらったイベントを処理することに対応する。そのため、FSM の遷移は、Promela 記述の内部プロセスがもらったメッセージを処理することに対応する。FSM の遷移は、開始状態、終了状態、入力記号、出力記号から構成される。よって、Promela 記述上で FSM の遷移を定義するため、Promela 記述上で開始状態、次の状態、入力記号、出力記号を定義する必要がある。

例えば、ある Promela 記述において、環境プロセスが内部プロセスにメッセージ x を送り、内部プロセスが x をもらった後、アクション y を実行したとする。このとき、y は、x に対応する処理である。また、内部プロセスがメッセージ x をもらう直前の状態は、$s_1 < l_1, a_1, a_2, ..., a_n >$ である。内部プロセスがもらったメッセージ x を処理した直後の状態は、$s_2 < l_2, a'_1, a'_2, ..., a'_n >$ である。この例を用いて、Promla 記述上の FSM の遷移を定義する。

Promela 記述上の FSM の遷移の開始状態

FSM の遷移の開始状態は、内部プロセスがメッセージをもらう直前の FSM の状態である。以上の例では、FSM の遷移の開始状態は、$s_1 < l_1, a_1, a_2, ..., a_n >$ である。

Promela 記述上の FSM の遷移の終点状態

FSM の遷移の終点状態は、内部プロセスがもらったメッセージを処理した直後の FSM の状態である。以上の例では、FSM の遷移の終点状態は、$s_2 < l_2, a'_1, a'_2, ..., a'_n >$ である。

Promela 記述上の FSM の遷移の入力記号

FSM の遷移の入力記号は、内部プロセスがもらったメッセージである。以上の例では、FSM の遷移の入力記号は、x である。なお、$x \in I_D$ がある。
形式的に、$\forall x(x \in I_{Promela} \Leftrightarrow x \in I_D)$、ただし

$I_{Promela}$：内部プロセスがもらったメッセージの集合

つまり、$I_D = I_{Promela}$

Promela 記述上の FSM の遷移の出力記号

FSM の遷移の出力記号は、内部プロセスがもらったメッセージを処理するアクションである。以上の例では、FSM の遷移の出力記号は、y である。なお、$y \in O_D$ がある。
形式的に、$\forall y(y \in O_{Promela} \Leftrightarrow y \in O_D)$、ただし

$O_{Promela}$：内部プロセスがもらったメッセージを処理するアクションの集合

つまり、$O_D = O_{Promela}$

そのときは、FSM の遷移は、$(s_1, x/y, s_2)$ で表現される。

4.3.3 Promela 記述を FSM で表現する規則

4.3.1 節と 4.3.2 節に基づいて、Promela 記述から FSM への変換規則を提供する。
Promela 記述から変換される FSM $D = (L_D, V_D, I_D, O_D, \delta_D, \lambda_D, s_{0D})$

- **ラベル集合** $L_D = L_{Promela}$
  $\forall l(l \in L_{Promela} \Leftrightarrow l \in L_D)$、ただし、

  $L_{Promela}$：Promela 記述でのラベル集合

- **変数集合** $V_D = V_{Promela}$
  $\forall v(v \in V_{Promela} \Leftrightarrow v \in V_D)$、ただし

  $V_{Promela}$：グローバル Promela 記述での変数集合

  $V_{Promela} \ni v$ は、(v の名前, v の値域) で表現される。

- **入力記号の集合** $I_D = I_{Promela}$
  $\forall x(x \in I_{Promela} \Leftrightarrow x \in I_D)$、ただし

  $I_{Promela}$：内部プロセスがもらったメッセージの集合

- **出力記号の集合** $O_D = O_{Promela}$
  $\forall y(y \in O_{Promela} \Leftrightarrow y \in O_D)$、ただし

  $O_{Promela}$：内部プロセスがもらったメッセージを処理するアクションの集合

- **遷移関数** $(\delta : S(D) \times I_D \rightarrow S(D))$

- **出力関数** $(\lambda : S(D) \times I_D \rightarrow O_D)$

- s_{0D} は、Promela 記述での初期状態である。

4.4 実装を FSM でモデル化

実装から変換される FSM M は、$(L_M, V_M, I_M, O_M, \delta_M, \lambda_M, s_{0M})$ の 7 組で記述される。Promela 記述から FSM への変換に対して、Promela 記述が存在するが、実装から FSM への変換に関しては、実装を用意する必要がある。そのために、Promela 記述に基づいて実装すると同時に、実装の内部状態を抽出するテストドライバを埋め込む。そのあと、実装を実行し、実装の内部状態のデータを引き出す。このデータを FSM M に変換する。また、$I_M = I_D$ と $S(M) = S(D)$ の前提が成立するように実装する必要がある。実装から FSM への変換を基準化するため、3 つのタスクが必要である。

1) $I_M = I_D$ と $S(M) = S(D)$ の前提が成立するように実装する規則を提供する。

2) **実装から** FSM **に変換する規則を提供する。**

3) **テストドライバの書き方を基準化する。**

4.4.1 実装の規則

実装は、制御システムを表現するＣプログラムであるので、イベントを取り扱うことによって実行される。また、実装はシステムの制御状態を指定する変数を含む。環境からあるイベントを受けると、現在の制御状態によって、このイベントを取り扱う。これは、実装の規則の一つである。具体的に、以下のように実装する。

```
while(1) {
   catch event E from enviroment;

   switch(E){
      case e1:
         handleEvente1();
         break;
      case e2:
         handleEvente2();
         break;
      ...
      case en:
         handleEventen();
         break;
   }
}
```

```
handleEventei() {

//labelID = control state
switch(labelID){
   case i_label:
      ...
      break;
   case label1:
      ...
      break;
   ...
   case labelm:
      ...
      break;
   }
}
```

$I_M = I_D$ **前提が成立するため 、実装上で** FSM M **の入力記号を** FSM D **の入力記号のように定義する必要がある。**FSM D **の入力記号は、内部プロセスがもらったメッセージであるので、**FSM M **の入力記号は、内部プロセスがもらったメッセージに対応する。内部プロセスがもらったメッセージは、実装で処理されるイベントに対応するので、**FSM M **の入力記号は、実装で処理されるイベントである。形式的に、**

$\forall x(x \in I_{Imp} \Leftrightarrow x \in I_M)$**、ただし、**$I_{Imp}$**：実装で処理されるイベントの集合**

つまり、$I_M = I_{Imp}$

そこで、$I_M = I_D$ **があるため、**$I_{Imp} = I_{Promela}$ **がある。すなわち、実装で処理されるイベントは、**Promela **記述の内部プロセスがもらったメッセージに対応する。**Promela **記述の内部プロセスがあるメッセージをもらえば、実装はそのメッセージ**

に対応するイベントを処理する必要がある。また、実装は、Promela 記述の内部プロセスがもらったメッセージに対応するイベント以外のイベントを処理しない。これは、実装の規則の一つである。

$S(M) = S(D)$ 前提が成立するため 、実装上で FSM M の状態を FSM D の状態のように定義する必要がある。FSM D の状態は、Promela 記述の（ラベル、全ての変数値）で表現されるので、FSM M の状態は、Promela 記述の（ラベル、全ての変数値）に対応する。

- Promela 記述のラベルは、実装の制御状態を指定する変数の値に対応する。そのため、FSM M のラベルは、実装の制御状態を指定する変数の値に対応する。つまり、実装の制御状態は、Promela の制御状態に対応する。実装の制御状態の数は、Promela の制御状態の数と同じである。形式的に、

  $\forall l(l \in L_{Imp} \Leftrightarrow l \in L_M)$、ただし、$L_{Imp}$：実装の制御状態を指定する変数の値の集合

  つまり、$L_M = L_{Imp}$
- Promela 記述の特別な変数以外の全ての変数は、実装の変数に対応する。Promela 記述の特別な変数は、`chan` または `mtype` 型の変数などである。これらの変数は、制御状態の遷移に影響しない。そのため、「Promela 記述の変数」とは、制御状態の遷移に影響する Promela 記述の変数を意味する。一方、実装のある変数は Promela 記述のどんな変数にも対応しない場合がある。この変数は、制御状態の遷移に影響しない。そのため、実装するとき、2 つの種類に変数を分ける必要がある。制御状態の遷移に影響する変数と制御状態の遷移に影響しない変数である。すなわち、Promela 記述の変数は、制御状態の遷移に影響する実装の変数に対応する。よって、FSM M の変数は、制御状態の遷移に影響する実装の変数である。形式的に、

  $\forall v(v \in V_{Imp} \Leftrightarrow v \in V_M)$、ただし、

  V_{Imp}：制御状態の遷移に影響する実装の変数の集合

  つまり、$V_M = V_{Imp}$

そこで、$S(M) = S(D)$ は、$L_{Imp} = L_{Promela}$ と $V_{Imp} = V_{Promela}$ を意味する。すなわち、Promela 記述の内部プロセスにあるラベルが存在すれば、このラベルに対応する実装の制御状態がある。つまり、実装の制御状態を指定する変数の値としてそのラベルを使用する必要がある。また、実装の制御状態を指定する変数へ Promela 記述のラベルに対応する値以外の値を代入することができない。これは、実装の規則の一つである。一方、実装は、制御状態の遷移に影響する変数として Promela 記述の全ての変数を使用しなければならない。また、Promela 記述の変数に対応する変数以外の実装の変数は、制御状態の遷移に影響しない変数である。これは、実装の規則の一つである。

実装するときの４つの規則をまとめると以下のようになる。

1) イベントを取り扱うことによってシステムを実装する。環境からあるイベントを受けると、現在の制御状態によって、このイベントを取り扱う。

2) Promela 記述の内部プロセスにあるラベルが存在すれば、このラベルに対応する実装の制御状態がある。実装では、システムの制御状態を指定する変数を使う。この変数の値は、Promela 記述の内部プロセスにラベルに対応する値である。また、この変数へ Promela 記述のラベルに対応する値以外の値を代入することができない。

3) 実装は、環境からのイベントとして Promela 記述の内部プロセスがメッセージを処理する。また、実装は、Promela 記述の内部プロセスがもらったメッセージに対応するイベント以外のイベントを処理しない。

4) 実装は、制御状態の遷移に影響する変数として Promela 記述の全ての変数を使用しなければならない。また、Promela 記述の変数に対応する変数以外の実装の変数は、制御状態の遷移に影響しない変数である。

本研究では、実装の基本的な規則を提供するが、実際には色々な実装のパターンがある。実装の具体的なパターンによって、$I_M = I_D$ と $S(M) = S(D)$ の前提が成立するため、ほかの実装規則を追加する必要がある。また、$S(M) = S(D)$ の前提を状態の対応関係に変換する場合がある。第 5 章で、いくつかの実装パターンを実験する。

4.4.2 実装を FSM でモデル化する規則

Promela 記述から FSM への変換規則（4.3.1 節と 4.3.2 節）に基づいて、実装から FSM への変換規則を提供する。

実装上の FSM の状態. 実装上の FSM の状態は、実装上で FSM のラベルと FSM の変数に基づいて定義される。

- FSM の一つのラベルは、実装の制御状態を指定する変数の一つの値に対応する。すなわち、任意の変数値 l に対して、l が実装の制御状態を指定する変数の値であれば、l は FSM のラベルである。形式的に、
  $\forall l(l \in L_{Imp} \Leftrightarrow l \in L_M)$
  つまり、$L_M = L_{Imp}$
- FSM の一つの変数は、実装の制御状態の遷移に影響する変数に対応する。すなわち、任意の変数 v に対して、v が実装の制御状態の遷移に影響する変数であれば、v は FSM の変数である。形式的に、
  $\forall v(v \in V_{Imp} \Leftrightarrow v \in V_M)$、ただし
  $V_{Imp} \ni v$ は、(v の名前, v の値域) で表現される。

つまり、$V_M = V_{Imp}$

そのとき、FSM の状態 s は、$< l, a_1, a_2, ..., a_n >$ で表現される。ただし、$l \in L_M$、a_i は v_i の値、$v_i \in V_M \quad (i = \overline{1,n})$ である。

FSM の状態集合を S とすると、$S(M) = L_M \times Dom(v_1) \times Dom(v_2) \times ... \times Dom(v_n)$ である。ただし、$Dom(v_i)$ は、$V_M \ni v_i$ の値域 $(i = \overline{1,n})$ である。

実装上の FSM の遷移. FSM の遷移は、実装がイベントを取り扱うことに対応する。FSM の遷移は、開始状態、終了状態、入力記号、出力記号から構成される。そのため、実装上で FSM の遷移を定義するため、実装上で開始状態、次の状態、入力記号、出力記号を定義する必要がある。

例えば、ある実装（C プログラム）で、ユーザー（または、システムの環境）がイベント x を発生し、C プログラムが x を受けた後、アクション y を実行したとする。このとき、y は、x に対応する処理である。また、C プログラムがイベント x を受ける直前の状態は、$s_1 < l_1, a_1, a_2, ..., a_n >$ である。C プログラムがイベント x を処理した直後の状態は、$s_2 < l_2, a'_1, a'_2, ..., a'_n >$ である。この例を用いて、実装上の FSM の遷移を定義する。

- FSM の遷移の開始状態は、C プログラムがイベント x を受ける直前の FSM の状態である。以上の例では、FSM の遷移の開始状態は、$s_1 < l_1, a_1, a_2, ..., a_n >$ である。
- FSM の遷移の終点状態は、C プログラムがイベント x を処理した直後の FSM の状態である。以上の例では、FSM の遷移の終了状態は、$s_2 < l_2, a'_1, a'_2, ..., a'_n >$ である。
- FSM の遷移の入力記号は、C プログラムが受けたイベントである。以上の例では、FSM の遷移の入力記号は、x である。なお、$x \in I_M$ がある。
  形式的に、$\forall x(x \in I_{Imp} \Leftrightarrow x \in I_M)$
  つまり、$I_M = I_{Imp}$
- FSM の遷移の出力記号は、C プログラムがイベントを処理するアクションである。以上の例では、FSM の遷移の出力記号は、y である。なお、$y \in O_M$ がある。
  形式的に、$\forall y(y \in O_{Imp} \Leftrightarrow y \in O_M)$、ただし
  O_{Imp}：C プログラムがイベントを処理するアクションの集合
  つまり、$O_M = O_{Imp}$

そのときは、FSM の遷移は、$(s_1, x/y, s_2)$ で表現される。

実装から FSM に変換する規則をまとめると以下のようになる。
実装から変換される FSM $M = (L_M, V_M, I_M, O_M, \delta_M, \lambda_M, s_{0M})$

- **ラベル集合** $L_M = L_{Imp}$
  $\forall l(l \in L_{Imp} \Leftrightarrow l \in L_M)$**、ただし、**

  L_{Imp}**：実装の制御状態を指定する変数の値の集合**

- **変数集合** $V_M = V_{Imp}$
  $\forall v(v \in V_{Imp} \Leftrightarrow v \in V_M)$**、ただし**

  V_{Imp}**：制御状態の遷移に影響する実装の変数の集合**

  $V_{Imp} \ni v$ **は、(**v **の名前**, v **の値域) で表現される。**

- **入力記号の集合** $I_M = I_{Imp}$
  $\forall x(x \in I_{Imp} \Leftrightarrow x \in I_M)$**、ただし**

  I_{Imp}**：実装で処理されるイベントの集合**

- **出力記号の集合** $O_M = O_{Imp}$
  $\forall y(y \in O_{Imp} \Leftrightarrow y \in O_Imp)$**、ただし**

  O_{Imp}**：C プログラムがイベントを処理するアクションの集合**

- **遷移関数** $(\delta : S(M) \times I_M \rightarrow S(M))$

- **出力関数** $(\lambda : S(M) \times I_M \rightarrow O_M)$

- s_{0M} **は、実装の初期状態である。**

4.4.3 テストドライバの基準化

テストドライバは、テストケースのシナリオによって実装を実行するとき、FSM M **の表現を出力する。**FSM M **の出力する内容をテストの実際結果という。この実際結果とテストケースの期待される結果と比べ、実装と** Promela **記述の整合性を判定する。テストケースでこの整合をテストするためには、テストケースの表現と** FSM M **の出力する表現の一貫性が必要である。本節は、テストケースの表現を定義する。テストケースの表現に基づいて、実装から** FSM M **の表現を出力するテストドライバの書き方を基準化する。**

テストケースの表現の定義. 一つのテストケースは、$<$ **初期状態**, (**入力記号**, **出力記号**, **次の状態**)* $>$ **で表現される。ただし、**

- **初期状態の表現は、**`STATE<i_label,i_1,i_2,...,i_n>`**である。ただし、**

  `i_label` **は、初期状態のラベルである。**

  `i_k` **は変数** v_k **の初期値、**$v_k \in V_D \quad (k = \overline{1,n})$ **である。**

- **入力記号の表現は、**`Input: content of input`**である。**
- **出力記号の表現は、**`Output: content of output`**である。**
- **状態の表現は、**`STATE<label,a_1,a_2,...,a_n>`**である。**

  `label` **は、状態のラベルである。**

  `a_k` **は変数** v_k **の値、** $v_k \in V_D \quad (k = \overline{1,n})$ **である。**

また、テストケースで表現される (入力記号, 出力記号, 次の状態) の組は、テストケースの一つのステップである。

テストケースの各ステップで、FSM M **の出力される表現は、**FSM M **の（入力記号、出力記号、状態）の組である。この表現は、テストケースのステップの表現と同じである。**

FSM M **の状態の表現**

FSM M **の状態は** FSM M **のラベルと変数値の集合で定義されるので、**FSM M **の状態の表現は、**FSM M **のラベルの表現と** FSM M **の変数の表現の組合わせである。また、実装がイベントを処理すると、**FSM M **の状態の表現を出力する。そのため、**FSM M **の状態の表現を出力するテストドライバは、イベントを処理するＣコード直後に埋め込まれる。**

- FSM M **のラベルの表現は、テストケースの状態のラベルの表現と同じである。**FSM M **のラベルは、実装の制御状態を指定する変数の値、または、**Promela **のラベルに対応する。そのため、**FSM M **のラベルの表現は、**Promela **のラベルで表現し、実装の制御状態を指定する変数の値に対応する文字列である。そのため、テストドライバは、文字列の配列を宣言し、この配列に** Promela **記述のラベルの集合を保存する。また、この配列のインデックスは、実装の制御状態を指定する変数の値に対応する。詳細は、以下の通りである。**

```
//correspondence of values to labels in Promela
#define i_label 0
#define label1  1
#define label2  2
...
#define labelm  m

//array of Promela labels
#define LABELNUMBER (m+1)
char StateLabel[m+1][256] = {"i_label","label1","label2",...,
                             "labelm"};
```

```
//variable which designates control state
int labelID = 0;
```

- FSM M の変数の表現は、テストケースの変数の表現と同じである。つまり、FSM M の変数集合の順番は、テストケースの変数集合の順番と同じである。

FSM M のラベルと変数の表現に基づいて、FSM M の状態の表現を出力する。FSM M の状態の表現を出力するテストドライバは、以下のように符号化される。

```
//C Program
handleEventei();

//TEST DRIVER: to print state of implementation
//(d_i is type of variable v_i)
printf(“ FOR TEST... STATE<%s, %d_1,%d_2,..,%d_n>\n ”,
        StateLabel+labelID,v_1,v_2,...,v_n);
```

FSM M の入力記号の表現

FSM M の入力記号の表現は、テストケースの入力記号の表現と同じである。FSM M の入力記号は、実装が受けるイベントに対応するので、FSM M の入力記号の表現は、イベントの内容を表現する文字列である。実装がイベントを受けると、FSM M の入力記号を出力する。そのため、FSM M の入力記号の表現を出力するテストドライバは、イベントを受けるCコード直後に埋め込まれる。FSM M の入力記号の表現を出力するテストドライバは、以下のように符号化される。

```
//C Program
case ei: //catch event ei

   //TEST DRIVER: to print input symbol of implementation
   printf("FOR TEST... Input: content of ei\n");

   handleEventei(); //handle event ei
   break;
```

また、FSM M の入力記号の内容とテストケースの入力記号の内容に一貫性があるため、実装のイベントの内容とPromelaのメッセージの内容に一貫性が必要である。

FSM M の出力記号の表現

FSM M の出力記号の表現は、テストケースの出力記号の表現と同じである。FSM M の出力記号は、イベントを処理するアクションに対応するので、FSM M の出力

記号の表現は、アクションの内容を表現する文字列である。イベントに対応するアクションを実行すると、FSM M の出力記号を出力する。そのため、FSM M の出力記号の表現を出力するテストドライバは、イベントに対応するアクションを実行するCコード直後に埋め込まれる。FSM M の出力記号の表現を出力するテストドライバは、以下のように符号化される。

```
//C Program
handleEventei() //handle event ei
{
   action a1;

   //TEST DRIVER: to print output symbol of implementation
   printf("FOR TEST... Output: content of a1\n");

   action a2;

   //TEST DRIVER: to print output symbol of implementation
   printf("FOR TEST... Output: content of a2\n");
   ...
}
```

また、FSM M の出力記号の内容とテストケースの出力記号の内容に一貫性があるため、実装のアクションの内容とPromelaのアクションの内容に一貫性が必要である。

4.5 2つのFSMの整合テスト

Promela記述と実装をFSMに変換してから、Promela記述と実装の整合性をテストすることは、FSM D と M の整合性をテストすることになる。本節は、2つのFSMの整合テストについて説明する。

4.5.1 テストケース生成

W-方法のように、本研究は2つの集合 P と W を用いてテストケースを生成する。しかし、W-方法と違い、本研究は、集合 W として入力記号の集合 I を使用する。また、集合 P を生成するアルゴリズムも変更される。

P の作成. W-方法のように、P はFSM D の全ての遷移を網羅するシーケンスの集合である。P を作るには、FSM D のテスト木を生成すれば良い。このテスト木の全ての部

分的なパスが P を構成する。一つの部分的パスは、連続的枝のシーケンスである。これは、テスト木のルートから終端節点または非終端節点までのパスである。しかし、W-方法と違い、P のシーケンスは、（入力記号、出力記号、次の状態）のシーケンスである。テスト木の各枝には（入力記号、出力記号、次の状態）の組で表示される。それで、P は、（入力記号、出力記号、次の状態）のシーケンスの集合である。空シーケンスも P の要素である。また、Spin の探索機能を利用するため、FSM D からテスト木 T を生成する手続きを変更する。Spin の標準探索アルゴリズムは深さ優先探索である。そのため、本研究は、W-方法で使われる幅優先探索の代わりに深さ優先探索によってテスト木 T を生成する。幅優先探索による生成されるテスト木は、深さ優先探索による生成されるテスト木と違うが、どちらでも FSM D の全ての遷移を網羅する。そのため、P は本研究でのテスト木 T に基づいて作れば、FSM D の全ての遷移を網羅するシーケンスの集合である。

S_V を表示された状態の集合とする。深さ優先探索によるテスト木を生成する手続きを解説する。

1) T のルートは、FSM D の初期状態である。これは、T の深さ 0 の節点である。この節点を S_V に入れる。
2) T の深さ k の節点 s_i を生成したと仮定する。２つのケースがある。

   ケース１．未表示の遷移 $(s_i, x/y, s_j)$ が FSM D に存在すれば、節点 s_i に一つの枝と子ノードを付ける。付けた枝のラベルは (x, y, s_j)、子ノードのラベルは、s_j である。この子ノードは T の深さ $k+1$ の節点である。また、遷移 $(s_i, x/y, s_j)$ に「表示された」マークを付ける。
   そのあと、深さ $k+1$ の節点 s_j が非終端節点であるかどうかを検討する。
   - 深さ $k+1$ の節点 s_j のラベルが S_V に含まれる節点のラベルと同じならば、終端節点（バックトラック）である。深さ $k+1$ の節点 s_j のラベルを S_V に入れる。また、深さ k の節点 s_i を引き返す。
   - そうでなければ、深さ $k+1$ の節点 s_j は非終端節点である。深さ $k+1$ の節点 s_j のラベルを S_V に入れる。2) に戻る。

   ケース２．s_i 状態からの未表示の遷移が FSM D に存在しないとき、
   - $k = 0$ の場合は、この手続きが終わる。
   - $k > 0$ の場合（バックトラック）は、深さ $k-1$ の節点 s_t を引き返す。ただし、節点 s_t からの枝 (m, n, s_j) がテスト木 T に存在する。2) に戻る。

FSM D では、有限数の状態が存在するので、上記の手続きは常に終わる。この手続きの結果は、$P =< s_0, (x_1, y_1, s_1), (x_2, y_2, s_2), ..., (x_n, y_n, s_n) >$ である。ただし、s_0 は D の初期状態、$x_i \in I_D$、$y_i \in O_D$、$s_i \in S(D)$ $(i = \overline{0, n})$ である。

集合 W の代わりの入力記号の集合 I．実装から変換される FSM M は、内部状態が見えるので、FSM M の任意の２つの状態を識別する必要がない。つまり、集合 W が不

必要である。そこで、本研究は、入力記号の集合を用いて、２つのFSMの同値な状態からの遷移集合は同じであるかどうか確認する。すなわち、4.5.2節で説明する。

Pを生成し、W-方法同様にPとIの連続でテストケースの集合を生成する。しかし、本研究のテストケースは、（入力記号、出力記号、次の状態）のシーケンスである。そのため、PとIの連続は、より複雑である。テストケース集合をTSとする。TSは、以下のように生成される。

```
TS = {};

generateTestcase() {
   TS = TS ∪ {s_0};
   for each p in P {
      //get state of last composition in p
      if(p is empty) {
         s = s_0;
      } else {
         s = last state in p;
      }
      //concatenate p • (x,ouput,state)
      for each x in I_D {
         if((s,x/y,s') exists in FSM D) {
            testcase = p • (x,y,s');
         }else {
            testcase = p • (x,null,null);
         }
         TS = TS ∪ {testcase};
      }
   }
   return TS;
}
```

4.5.2 テストカバレッジ

W-方法のように、本研究で生成したテストケースを用いて、操作エラーと遷移エラーを見つける。また、提案手法で生成したテストケースの集合のテストカバレッジを評価するため、本研究は、余分・欠遷移エラーを定義する。

余分・欠遷移エラーの定義. *FSM* AとA'を与える。Aは、*Promela*記述から変換される*FSM*であり、「基準」版である。A'は、実装から変換される*FSM*である。テストケース

$p \cdot (x, output, state)$ を与える。$FSM\ A$ と A' の初期状態にテストケースの部分 p を実行した後、A の状態は $s < l, a_1, a_2, ..., a_n >$、$A'$ の状態は $s' < l', a'_1, a'_2, ..., a'_n >$ である。また、この実行では、操作エラーと遷移エラーを見つけないので、s' は s と同値である。$\forall x \in I_A$ に対して、

1) 遷移 $(s, x/y, s_n)$ が $FSM\ A$ に存在すれば、$output = y$、$state = s_n$ がある。そのとき、遷移 $(s', x/y, s_n)$ が $FSM\ A'$ に存在しなければ、$FSM\ A'$ に欠遷移エラーがある。

2) そうでなければ、$output = null$、$state = null$ がある。そのとき、$(s', x/y', s'_n)$ 遷移が $FSM\ A'$ に存在すれば、$FSM\ A'$ に余分遷移エラーがある。

ただし、I_A は、$FSM\ A$ の入力記号の集合である。

本研究は、$I_M = I_D$ と $S(M) = S(D)$ の前提を用いて、$M = D$ であるかどうか確認する。任意の FSM は、(状態集合、遷移集合) で特徴付けられるので、$M = D$ は、$S(M) = S(D)$ と $Tr(M) = Tr(D)$ の意味である。ただし、Tr は、FSM の遷移集合である。本研究では、$I_M = I_D$ と $S(M) = S(D)$ の前提を用いて、$Tr(M) = Tr(D)$ があるかどうかテストする。

テストケースの集合 TS は P と I_D の連続に基づいて生成される。すなわち、$\forall p \in P$、$\forall x \in I_D$ に対して、テストケース $p \cdot (x, output, state)$ が常に存在する。ただし、$p \in P$、$x \in I_D$ である。テストケースの部分 p は、FSM D の全ての状態と遷移を網羅する。部分 p に基づいて、操作エラーと遷移エラーを見つける。FSM D と M の初期状態に部分 p を入力（実行）した後、D の状態は $s < l, a_1, a_2, ..., a_n >$、$M$ の状態は $s' < l', a'_1, a'_2, ..., a'_n >$ である。この実行で操作エラーと遷移エラーを見つけなければ、s' は s と同値である。つまり、$s = s'$ である。そのとき、テストケースの部分 $(x, output, state)$ を用いて、s' からの遷移集合は、s からの遷移集合と同じであるかどうかテストする。以下のように、2つのケースがある。

$\forall x \in I_D$ に対して、

1) 遷移 $(s, x/y, s_n)$ が FSM D に存在すれば、$output = y$、$state = s_n$ である。そのとき、

   1.1) 遷移 $(s', x/y, s_n)$ が FSM M に存在しなければ、FSM M に欠遷移エラーがある。

   1.2) 遷移 $(s', x/y', s'_n)$ が FSM M に存在するが、$y' \neq y$ または $s'_n \neq s_n$ であれば、FSM M に操作エラーまたは遷移エラーがある。

2) そうでなければ、$output = null$、$state = null$ がある。そのとき、$(s', x/y', s'_n)$ 遷移が FSM M に存在すれば、FSM M に余分遷移エラーがある。

また、$I_M = I_D$ と $S(M) = S(D)$ であるので、2つのケースにエラーを見つけなければ、s' からの遷移集合は、s からの遷移集合と同じである。

このように、テストケースの部分 p は、全ての状態と遷移を到達する。テストケースの部分 $(x, output, state)$ は、それぞれの状態に、この状態からの遷移集合の対応関係をテス

トする。そのため、$I_M = I_D$ と $S(M) = S(D)$ を前提として、本研究は $M = D$ が成立するかどうか確認することができる。つまり、M と D の整合性をテストすることができる。

本研究は、FSM が完全に規定され（completely specified）、ミニマル（minimal）であることを前提とせず、集合 W も不必要である。そのため、W-方法と比べると、扱い易い。また、W-方法と違い、本研究は、パステスト（Passing Test）と失敗テスト（Failing Test）の両方のテストケースを生成する。パステストのテストケースは、$(x, null, null)$ の組成を含まない。つまり、パステストのテストケースを実行するとき、正しい実装はテストケースが示した全ての組成を実行するべきである。失敗テストのテストケースは、$(x, null, null)$ の組成を含む。つまり、失敗テストのテストケースを実行するとき、正しい実装は、テストケースが表示した全ての組成を実行するべきではない。これらのテストケースを用いて、M と D の整合性をテストすることができる。

テストケースの部分 p と部分 $(x, output, state)$ の連続は、テスト木に基づく。テストケースの部分 p を実行した後、FSM D の状態は s である。$I_D \ni x$ に対して、遷移 $(s, x/y, s_n)$ が FSM D に存在すれば、この遷移を含む $p \in P$ が存在する。これは、集合 P が FSM D の全ての状態と遷移を網羅するからである。そのため、テストケースの部分 p と部分 $(x, output, state)$ の連続は不必要である。よって、テストケースの部分 p と部分 $(x, output, state)$ の連続は、$I_D \ni x$ に対して遷移 $(s, x/y, s_n)$ が FSM D に存在しないときにだけに行われる。すなわち、テストケースの部分 p と部分 $(x, null, null)$ の連続だけを行う。言い換えれば、テストケースの部分 p と部分 $(x, output, state)$ の連続は、失敗テストしか生成しない。

完全に規定される（completely specified）システムに対するテストケースの集合は、パステストのみを含む。そのとき、P と I_D の連続を行うことは不必要である。つまり、完全に規定されるシステムのテストケースの集合は、集合 P である。

4.6 Promela 記述からのテストケース自動生成

本章ではここまで、提案手法の基礎理論を説明してきた。本節は、Spin で表現される探索機能を用いて、Promela 記述からテストケースを自動生成する手法を説明する。

4.6.1 Spin の探索機能を分析

Spin で使われる基本的な探索アルゴリズムは、深さ優先探索である。Promela 記述を表現する有限オートマトン $A = (S, Tr, s_0)$ を与える。ただし、S は Promela 記述の状態集合、Tr は Promela 記述の遷移集合、s_0 は Promela 記述の初期状態である。Spin で使われる探索アルゴリズムは、以下のとおりである。

```
Stack D = {};
Statespace V = {};
```

```
Start()
{
   Add_Statespace(V,s_0);
   Push_Stack(D,s_0);
   Search();
}

Search()
{
   s = Top_Stack(D);
   for each (s,s') ∈ Tr {
      if(In_Statespace(V,s') == false) {
         Add_Statespace(V,s');
         Push_Stack(D,s');
         Search();
      }
   }
   Pop_Stack(D);
}
```

このアルゴリズムでは、スタック D **と状態空間** V **の** 2 **つのデータ構造を使う。状態空間** V **は、**A **の訪れた状態集合を保存し、スタック** D **は、初期状態** s_0 **から状態** s **までのパスを保存する。スタック** D **は、状態の順序集合である。スタック** D **を** $< s_0, s_1, s_2, ..., s_n, s >$ **で表現し、順序を記号** $<$ **かつ** $>$ **で表現すると、**$(s_0 < s_1 < s_2 < ... < s_n < s)$ **である。この順序は、状態の深さの順序である。現在の状態が** s **であるとき、任意の** $(s, s') \in Tr$ **に対して、**

- **状態** s' **に訪れたことがなければ、スタック** D **に** s' **を入れる。そのとき、スタック** D **は** $< s_0, s_1, s_2, ..., s_n, s, s' >$ **で表現される。なお、**$(s < s')$ **である。**
- **既に、状態** s' **に訪れている（バックトラック）場合、スタック** D **に** s' **は入れない。**

全ての $(s, s') \in Tr$ **を調べてから、スタック** D **が変わらなければ（もう一度、バックトラック）、スタック** D **のトップの状態を引き出す。そのとき、スタック** D **は** $< s_0, s_1, s_2, ..., s_n >$ **で表現される。**

Spin **は、探索アルゴリズムを実行すると、**A **の全ての状態と遷移を到達し、深さ優先探索木を生成する。**A **は** FSM D**、探索木は本研究の提案手法で定義されるテスト木** T **に対応する。また、スタック** D **に保存される状態の順序集合は、集合** $P \ni p$ **に対応する。そこで、本研究は、**Spin **で定義されるスタック** D **と状態空間** V **を用いて、**Promela **記述からテストケースを自動生成する。**

4.6.2 Promela 記述からのテストケース自動生成のプロセス

Promela からのテストケース自動生成のプロセスは、３つのステップを含む。

1) スタック D を用いて集合 $P \ni p$ を生成するため、Spin での探索木の枝にラベル（入力記号、出力記号、次の状態）を付ける。そのため、Promela 記述に（入力記号、出力記号、次の状態）を出力するＣコードを埋め込む。Promela 記述にＣコードを埋め込むことはＣコードからモデルを自動抽出することをサポートすることを主目的とする。

2) Ｃコードを埋め込んだ Promela 記述をコンパイルし、Spin が Promela 記述を検証する各ステップを詳しく記述するデータを抽出する。このデータは、冗長な検証のデータである。冗長な検証のデータも Spin が探索木を探索する各ステップを詳しく記述する。すなわち、Spin を用いて、Ｃコードを埋め込んだ Promela 記述からソーステキスト (source text) を生成する。Ｃ言語コンパイラーとともにオプション -DCHECK を用いて、 このソーステキストからファイル .exe にコンパイルする。なお、このコンパイルの情報は、ファイル dcheck.out に保存する。

3) ファイル dcheck.out からテストケースを自動生成する。テストケースを自動生成するため、本研究はプログラム TestcaseGenerator を作る。このプログラムは、ファイル dcheck.out を処理し、テストケースの集合を生成する。なお、プログラム TestcaseGenerator は本研究のテストケース生成ツールである。

Spin での探索木の枝にラベルを付ける。

Spin での探索木の枝にラベルを付けるため、ラベルを表現するデータ構造が必要である。このデータ構造は、Promela 記述に埋め込まれ、以下のように定義される。

```
c_decl{
   typedef struct Node{
      char label[256];//label of next state
      state_vector;//set of variables in next state
      char input[256];//input symbol
      char output[256];//output symbol
   }Node;
}
```

ただし、label は次の状態のラベル、state_vector は次の状態の変数集合、input は遷移の入力記号、output は遷移の出力記号を表現する。label と state_vector の組合わせは、次の状態を表現する。state_vector は、Promela 記述の変数集合であり、Spin の state vector の変数集合に対応する。Promela 記述をコンパイルす

ると、Spin の state vector は、ファイル pan.h に出力される。そのため、Promela 記述の変数集合を確認するため、Spin の state vector を用いる。

テスト木 T の枝は、FSM D の遷移に対応する。すなわち、この枝は、Promela 記述で内部プロセスがもらったメッセージを処理することに対応する。そのため、Promela 記述で内部プロセスがもらった任意のメッセージを処理した直後に Spin での探索木の枝にラベルを付ける。つまり、Spin での探索木の枝にラベルを付ける埋込 C コードは、内部プロセスが任意のメッセージを処理する Promela コード直後に埋め込まれる。技術的にこのタクスは、ラベルを表現するデータに新しいラベルを代入し、printf() を用いてこのデータを抽出する。Promela 記述で内部プロセスがもらった任意のメッセージを処理した直後、もらったメッセージは入力記号、メッセージに対応するアクションは出力記号、Promela 記述での現在のラベルは次の状態のラベル、Promela 記述での現在の変数値は次の状態の変数値に対応する。つまり、(Promela 記述での現在のラベル、Promela 記述での現在の変数値、メッセージ、アクション) は新しいラベルである。この新しいラベルは、ラベルのデータに代入され、抽出される。Spin での探索木の枝にラベルを付ける埋込 C コードを以下のように書く。

```
c_code{
   printNode(Node n) {
      printf("Input: %s\n",n.input);
      printf("Output: %s\n",n.output);
      printf("STATE<%s,%state_vector>\n",n.label,n.state_vector);
   }
}

/*After catching and handling message from enviroment in Promela*/
c_code{
   Node tempNode;

   strcpy(tempNode.input,"content of message");
   strcpy(tempNode.output, "content of actions");
   strcpy(tempNode.label,"current label in Promela");
   tempNode.state_vector = current values of variables in Promela;

   printNode(tempNode);
}
```

関数 printNode() の戻り値は、後でテストケースの形式を合わせることに使われるので、基準化される。この基準は、実装の内部状態を抽出するテストドライバを書く基準に対応する。また、テストケースの形式に初期状態 s_0 を付けられるので、

Promela 記述のプロセスを実行する前に初期状態をプリントする。すなわち、以下のとおりである。

```
init{
   c_code{
      Node initialNode;

      initialNode.input = "";
      initialNode.output = "";
      initialNode.label = initial label in Promela;
      initialNode.state_vector = initial values of variables
                                 in Promela;

      printNode(initialNode);
   };

   run InternalProcess();
   run Environment();
}
```

Spin が探索木を探索する各ステップを詳しく記述するデータを抽出する。

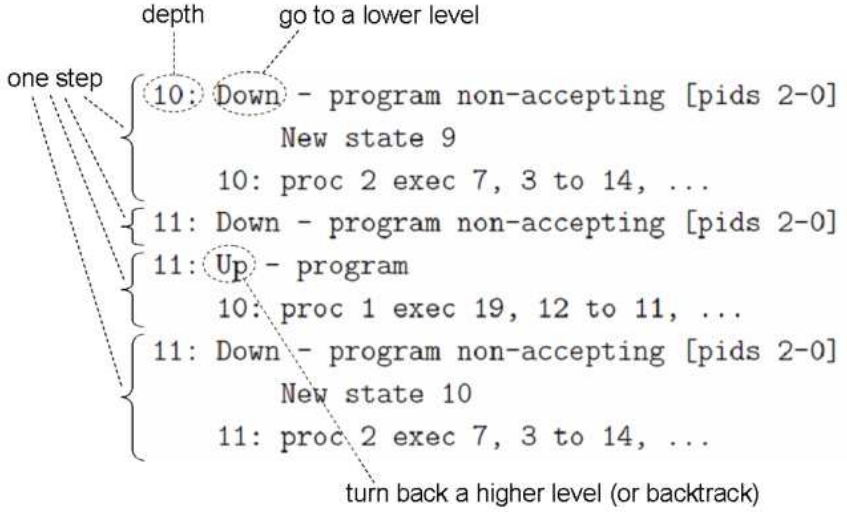

図 4.2: コンパイルの情報の標準的な出力

バージョン２．０以降の Spin は、Spin が Promela 記述を検証する各ステップで何をしているか調査するため、-DCHECK というコンパイル時のオプションをサポートし

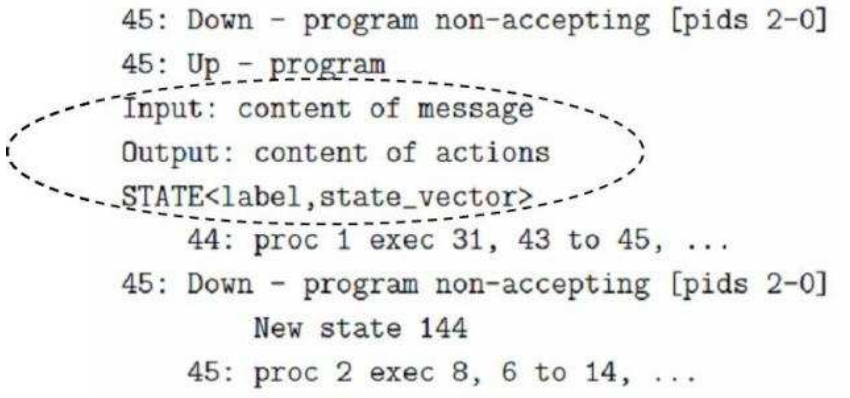

図 4.3: **ステップ１で付けたラベルのデータの標準的な出力**

ている。この機能を用いて、Ｃコードを埋め込んだ Promela **記述から冗長な検証のデータを抽出する。この手続きは、以下のとおりである。**

```
$ spin -a Promela.pml       #generate a verifier
$ cc -DCHECK -o pan pan.c   #compile the verifier with -DCHECK option
$ ./pan > dcheck.out        #execute the verifier
```

このコンパイルの情報は Spin **が** Promela **記述を検証する各ステップで標準的な形式で出力され（図** 4.2**）、ファイル** dcheck.out **に保存される。また、ステップ１で付けたラベルのデータも** Spin **が** Promela **記述を検証する各ステップで抽出される（図** 4.3**）。しかし、これは、** Spin **が** Promela **記述を検証する全てのステップではなく、内部プロセスがもらったメッセージを処理した各ステップだけで実行される。この標準的な出力の各ステップでは、深さ、探索の方向（**Up **または** Down**）、付けたデータの出力だけを使用する。なお、** Up **は、探索におけるバックトラックの意味である。**

テストケースを自動生成する。

ファイル dcheck.out **からテストケースを自動生成するため、プログラム** Testcase-Generator **を作る。このプログラムは、** dcheck.out **の標準に基づいて** dcheck.out **を処理し、テストケースを自動生成する。**TestcaseGenerator **は、** Spin **でのスタック** D **としてスタック** State_Stack **を使用する。**State_Stack **は枝のラベルの順序集合であり、集合** $P \ni p$ **を表現する。**State_Stack **は、以下のように定義される。**

```
typedef struct Node{
   char state[256];
   char input[256];
   char output[256];
   int depth;
```

```
}Node;

#define MAXSIZE 1000000

Node State_Stack[MAXSIZE];
```

State_Stack **の要素を表現するデータ構造は、ステップ１で付けたラベルのデータの出力を含む。それに加えて、このデータ構造は** depth **を含む。**depth **は、ラベルを表現するノードの深さを保存する。テストケースを生成する際、**depth **を用いてバックトラックするかどうか確認する。**TestcaseGenerator **のアルゴリズムは以下の通りである。**

```
generateTestcase() {
   char tempStr[256];
   char traverseDirection[256];
   Node tempNode;
   int currentDepth;

   open file dcheck.out;

   while (!(end of file dcheck.out)) {
      tempStr = read a line from dcheck.out;

      //tempStr = (depth): (Up or Down) ...
      if(isdigit(tempStr[0])) {
         currentDepth = get depth from tempStr;
         tempNode.depth = currentDepth;
         //traverseDirection = Up or Down
         traverseDirection = get traverse direction from tempStr;

         //Check backtrack
         if(traverseDirection == "Up") {//backtrack in Spin
            Node topNode = Top_State_Stack();

            //backtrack in Testing Tree
            if(currentDepth <= topNode.depth) {
               //first backtrack
               if(firstBacktrack) {
                  printP();
```

```
                }

                Pop_State_Stack(); //Turn back a higher level
            }
        }
    }

    //tempStr = "Input: ..."
    else if(tempStr[0] == 'I') {
        strcpy(tempNode.input,tempStr);
    }

    //tempStr = "Output: ..."
    else if(tempStr[0] == 'O') {
        strcpy(tempNode.output,tempStr);
    }

    //tempStr = "STATE<...>"
    else if(tempStr[0] == 'S') {
        strcpy(tempNode.state,tempStr);
        //Attack a branch to p
        Push_State_Stack(tempNode);
        //Print p
        printP();
    }
}

//Print last p
printP();

close file dcheck.out;
}
```

dcheck.out を読むとき、number（深さ）、I(入力記号)、O（出力記号）、および、S（状態）から始める行だけを処理する。つまり、４つの場合がある。

1) 深さの行を読む場合、探索の方向を調べる。探索の方向が Down ならば、何もしない。探索の方向が Up ならば、バックトラックを見つける。これは、Spin で使用される探索木のバックトラックである。しかし、テスト木のラベルのデー

タは、Spin が Promela 記述を検証する全てのステップではなく、内部プロセスがもらったメッセージを処理した各ステップだけで抽出される。そのため、このバックトラックはテスト木のバックトラックではない場合がある。よって、このバックトラックは、テスト木のバックトラックであるかどうか調べる必要がある。

- (currentDepht > topNode.depth) の場合
  バックトラックはテスト木のバックトラックではないため、何もしない。
- (currentDepht <= topNode.depth) の場合
  バックトラックはテスト木のバックトラックである。このバックトラックが初めてのバックトラックならば、p として State_Stack をプリントする。そのあと、Pop_State_Stack() で、直前の要素を引き返す。

2) 入力記号を読む場合、tempNode.input に新しい枝の入力記号を代入する。
3) 出力記号を読む場合、tempNode.output に新しい枝の出力記号を代入する。
4) 状態から始める行を読む場合、tempNode.state に新しい枝の入力記号を保存する。そのあと、新しい枝を p に付け、p をプリントする。

TestcaseGenerator は、集合 P の全て要素をプリントし、ファイル P.out に保存する。つまり、TestcaseGenerator の戻り値は、集合 P である。完全に規定されているシステムに対するテストケースの集合は集合 P であるので、ファイル P.out はテストケースの集合である。完全に規定されないシステムに対して、集合 P と入力記号の集合 I_D の連続を行う必要がある。この段階は、dcheck.out と P.out に基づいて、自動化できると考えられる。この自動化については今後の課題である。

その後、生成したテストケースで実装をテストする。すなわち、テストケースのシナリオのように、実装を実行する。エラーが見つかれば、実装はまだ設計モデルと整合していない。このエラーに基づいて、実装を修正する。全てのテストケースが成功すれば、実装は設計モデルと整合していると言うことができる。

第5章　実験と評価

提案手法を評価するため、キッチンタイマ [8] を題材として小規模な実験を行った。このキッチンタイマには、Promela による設計モデルが既に存在する。また、この設計モデルの正しさは Spin によって既に検証されている。

5.1　キッチンタイマの概要

一般的に、キッチンタイマとは料理をするときに時間を計るものである。本研究の例として用いるキッチンタイマは、１６ビット・マイコン搭載ボード「NE-R8C/25」で動作し、入力スイッチが２個、出力用の LED が２個ある（図 5.1）。

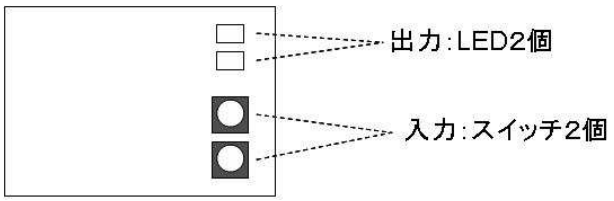

図 5.1: キッチンタイマのマイコンボードの入出力

5.1.1　キッチンタイマの仕様

キッチンタイマの仕様 は以下のとおりである。

1) キッチンタイマを起動する。
   プログラムを起動すると、LED は全消灯して操作待ちになっている。

2) 時間を設定する。
   スイッチ１で時間を設定する。設定時間は２進数で LED に表示される。そして、設定数は１分から３分までの１分刻みで設定する。

3) 計時を開始する。
   スイッチ２で計時を開始する。キッチンタイマがカウントダウンを開始すると LED が交互に点滅する。

4) **一時停止する。**
計時中にスイッチ１を押すと計時を終了する。スイッチ２を押すと計時を一時停止し、LED **が全消灯する。**
また、一時停止中にスイッチ２を押すと、計時を再開し、LED **が交互に点滅する。**

5) **時間が経過してアラーム表示になる。**
設定時間が経過するとユーザーに通知するようにアラームを開始し、LED **が３秒間同時全点滅する。**
アラーム中にスイッチ１またはスイッチ２を押すと、アラームを停止し、LED **が全消灯する。また、アラーム停止中にスイッチ１を押すと時間数を１に設定し、スイッチ２を押すと計時を開始する。**

このキッチンタイマでは、「キー入力監視」、「計時」、「表示」の３つの並行タスクと周期ハンドラがある（図 5.2**）。周期ハンドラは、タイミングを作るものである。計時またはアラーム中、周期ハンドラは１秒ごとに事象の通知を計時タスクに送る。計時中の表示は、周期ハンドラが** 500ms **おきに事象の通知を表示タスクに送る。アラーム中の表示は、周期ハンドラが** 100ms **おきに事象の通知を表示タスクに送る。キー入力監視タスクは、キードライバを経由してスイッチが押下されるのを監視する。計時タスクは、キー入力監視タスク、または、周期ハンドラからの事象通知を受けて時間設定の機能を動作させる。表示タスクは、計時タスク、または、周期ハンドラからの表示の通知を受けると、**LED **に表示する。**

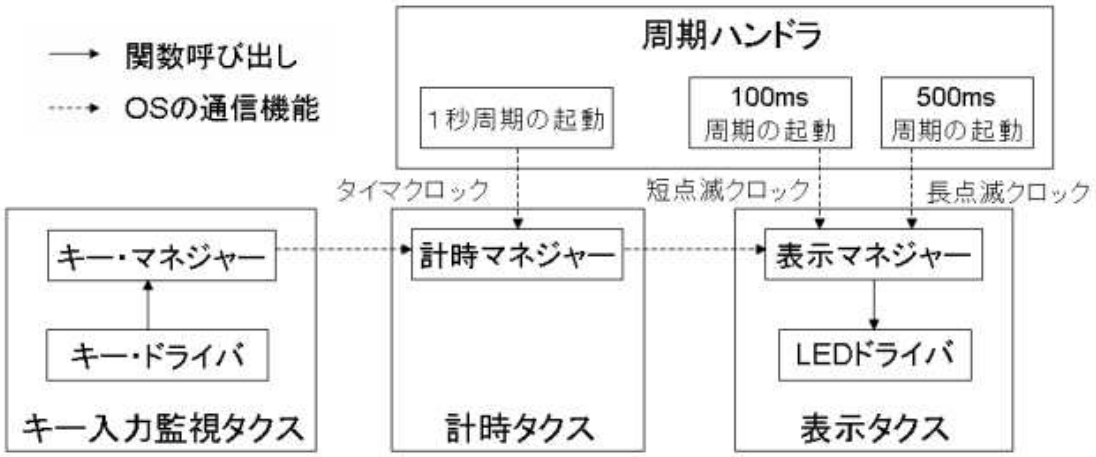

図 5.2: **キッチンタイマの機能ユニット関連図**

本研究は内部プロセスが一つしかない Promela **記述のみに着目している。そのため、このキッチンタイマの仕様を簡単なものに変更し、簡単なキッチンタイマを対象とした実験を行った。簡単なキッチンタイマでは、「表示」タスクがなくなり、「キー入力監視」、「計時」の** 2 **つの並行タスクと周期ハンドラがある（図** 5.3**）。計時またはアラームには、周**

期ハンドラが１秒ごとに事象の通知を計時タスクに送る。キー入力監視タスクは、キードライバを経由してスイッチが押下されるのを監視する。計時タスクは、キー入力監視タスク、または、周期ハンドラからの事象通知を受けて時間設定の機能を動作させる。さらに、計時タスクは内部システム、周期ハンドラとキー入力監視タスクはシステムの環境に対応する。

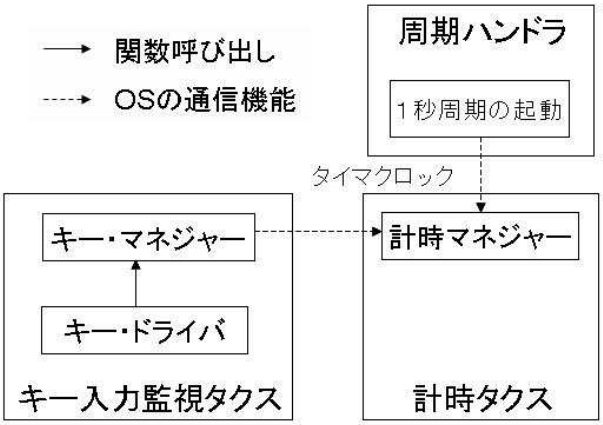

図 5.3: **簡単なキッチンタイマの機能ユニット関連図**

簡単なキッチンタイマの状態遷移図（図 5.4**）で簡単なキッチンタイマの仕様を記述する。また、図** 5.4 **の記号の説明を表** 5.1 **に示す。**

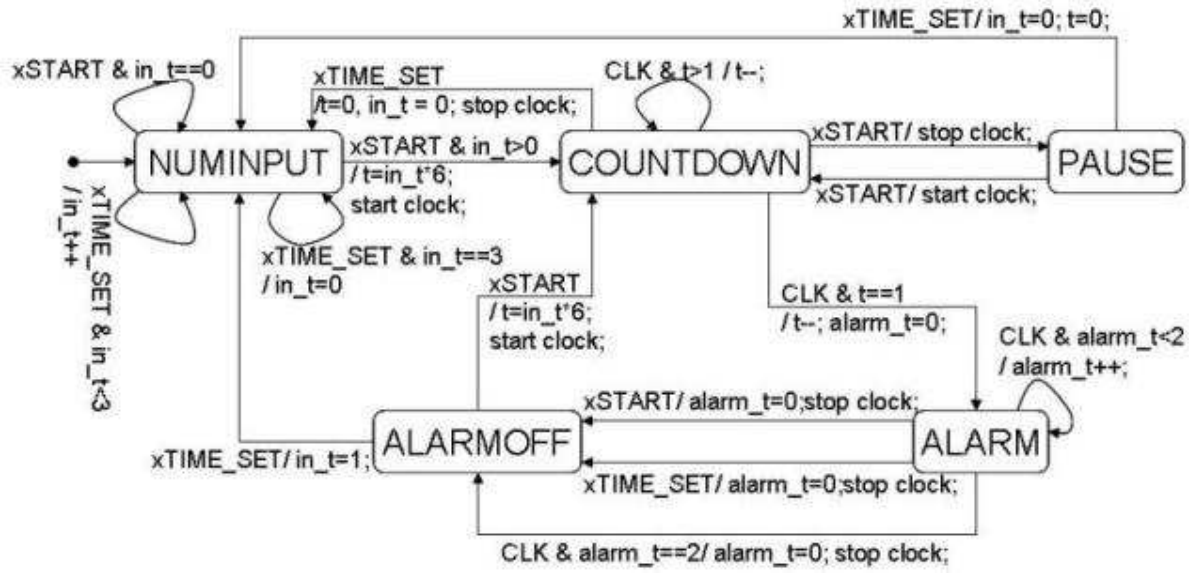

図 5.4: **簡単なキッチンタイマの状態遷移図**

表 5.1: 簡単なキッチンタイマの状態遷移図の記号表

記号	意味
NUMINPUT	時間設定の状態
COUNTDOWN	計時中の状態
PAUSE	一時停止中の状態
ALARM	アラーム中の状態
ALARMOFF	アラーム停止中の状態
xSTART	キー入力監視タスクからのスイッチ１を押した事象の通知
xTIME_SET	キー入力監視タスクからのスイッチ２を押した事象の通知
CLK	１秒ごとに周期ハンドラからの事象の通知
in_t	設定時間を表現する変数
t	カウントダウン時間を表現する変数
alarm_t	アラーム時間を表現する変数

以上の変更に合わせて、キッチンタイマの Promela 記述を簡単なキッチンタイマの Promela 記述に変更した。今後、「キッチンタイマ」は、簡単なキッチンタイマのことを指す。

5.1.2 キッチンタイマの Promela 記述

キッチンタイマの Promela 記述は、２つの並行プロセスを含む。Timer（計時）、および、Trigger（環境）である。プロセス Timer は、計時タスクを記述する。プロセス Trigger は、キー入力監視タスクと周期ハンドラを記述する。また、キッチンタイマの Promela 記述は、Spin により正しさが検証されている。

5.2 キッチンタイマのテストケースの自動生成

Promela 記述からのテストケース自動生成のプロセス（4.6.2 節）によって、キッチンタイマのテストケースを生成する。Spin での探索木にラベルを付ける C コードをキッチンタイマの Promela 記述に埋め込む。この Promela 記述をコンパイルし、コンパイルの情報を `dcheck.out` に保存する。`TestcaseGenerator` を用いて `dcheck.out` を処理し、キッチンタイマのテストケースを生成する。

5.2.1 Cコードをキッチンタイマの Promela 記述に埋め込み

Cコードをキッチンタイマの Promela 記述に埋め込む前に、キッチンタイマの Promela 記述のラベル集合、変数集合、メッセージ集合、アクション集合を確認する。キッチンタイマの Promela 記述をコンパイルし、キッチンタイマの Promela 記述の pan.h を抽出する。pan.h に記述される state vector に基づいて、キッチンタイマの Promela 記述の変数集合を確認する。キッチンタイマの Promela 記述の各集合を表 5.2 に示す。

表 5.2 の変数集合に基づき、キッチンタイマで Spin の枝のラベルを表現するデータ構造を以下のように定義する。

```
c_decl{
   typedef struct Node{
      char label[256];
      int in_t;
      int t;
      int alarm_t;
      unsigned bTimeClk;
      char input[256];
      char output[256];
   }Node;
   ...
}
```

その後、4.6.2 節で示された方法によって、入力記号、出力記号、状態を引き出し、出力するCコードをキッチンタイマの Promela 記述に埋め込む。

5.2.2 キッチンタイマの Promela 記述をコンパイル

オプション-DCHECK を用いて、キッチンタイマのCコードを埋め込んだ Promela 記述をコンパイルした。コンパイルの結果は、dcheck.out に保存される。キッチンタイマの dcheck.out の抽出物は、以下の通りである。

```
77: Down - program non-accepting [pids 2-0]
77: Up - program
Input: TIMER_CLK
Output: Time--
STATE<COUNTDOWN,3,17,0,1>depth=76
    76: proc 1 exec 31, 43 to 45, {c_code8}   non-accepting [tau=0]
77: Down - program non-accepting [pids 2-0]
        New state 219
```

表 5.2: キッチンタイマの Promela 記述の各集合

集合	要素	意味
ラベル集合	NUMINPUT	時間設定の状態
	COUNTDOWN	計時中の状態
	PAUSE	一時停止中の状態
	ALARM	アラーム中の状態
	ALARMOFF	アラーム停止中の状態
変数集合	in_t	設定時間を表現する変数
	t	カウントダウン時間を表現する変数
	$alarm\_t$	アラーム時間を表現する変数
	$bTimeClk$	周期ハンドラの状態を表現する変数
入力記号の集合	xSTART	キー入力監視タスクからのスイッチ１を押した事象の通知
	xTIME_SET	キー入力監視タスクからのスイッチ２を押した事象の通知
	TIMER_CLK	１秒ごとに周期ハンドラからの事象の通知
出力記号の集合	Increase InTime	in_t を１分から３分まで１分刻みで設定する。
	Clear InTime	$in\_t = 0$
	Clear Time	$t = 0$
	Clear AlarmTime	$alarm\_t = 0$
	Set Time = InTime * 6	$t = in\_t * 6$
	Time−−	$t - -$
	AlarmTime++	$alarm\_t + +$
	Stop clock	$bTimeClk = 0$
	Start clock	$bTimeClk = 1$
	Set Time = 1	$t = 1$

```
77: proc 2 exec 8, 6 to 14, chanTimer!xSTART  rendez-vous
    non-accepting [tau=0]
```

5.2.3 キッチンタイマのテストケースを生成

TestcaseGenerator を用いて、dcheck.out を処理し、キッチンタイマの集合 P を生成した。また、キッチンタイマの p の抽出物は、以下の通りである。

```
STATE<COUNTDOWN,3,9,0,1>
Input: TIMER_CLK
Output: Time--
STATE<COUNTDOWN,3,8,0,1>
Input: xSTART
Output: Stop clock
STATE<PAUSE,3,8,0,0>
Input: xSTART
Output: Start clock
STATE<COUNTDOWN,3,8,0,1>
Input: TIMER_CLK
Output: Time--
STATE<COUNTDOWN,3,7,0,1>
```

キッチンタイマは、完全に規定されるシステムなので、キッチンタイマの集合 P は、キッチンタイマのテストケースの集合である。また、この集合は、1155 個のテストケースを含んでいる。

5.3 キッチンタイマの実装

キッチンタイマの Promela 記述に基づいて、C コードでキッチンタイマを実装する。また、キッチンタイマを実装するとき、キッチンタイマの C プログラムにテストドライバを埋め込んでいる。

キッチンタイマを実装するとき、$I_M = I_D$ と $S(M) = S(D)$ の前提が成立する必要がある。$I_M = I_D$ の前提が成立するためにキッチンタイマの C プログラムは、Promela 記述のメッセージ集合に対応する 3 つのイベントを処理した。これは、xTIME_SET、xSTART、および、TIMER_CLK である。一方、$S(M) = S(D)$ の成立は、より複雑である。$S(M) = S(D)$ の前提は、実装のラベル集合が Promela 記述のラベル集合、実装の制御状態の遷移に影響する変数集合が Promela 記述の変数集合と同じであることを意味する。$S(M) = S(D)$ の前提を検討するため、キッチンタイマを 3 つの実装パターンで実装した。

5.3.1 パターン１: $L_{Imp} = L_{Promela}$、$V_{Imp} = V_{Promela}$

この実装パターンでは、Promela**記述と対応づけて実装する。そのため、実装のラベル集合は** Promela **記述のラベル集合、実装の制御状態の遷移に影響する変数集合は、**Promela**記述の変数集合と同じである。このとき、**$S(M) = S(D)$**が簡単に成立する。**

5.3.2 パターン２: $L_{Imp} > L_{Promela}$、$V_{Imp} = V_{Promela}$

この実装パターンでは、実装の制御状態の遷移に影響する変数集合は Promela **記述の変数集合と同じであるが、実装のラベル集合は、**Promela**記述のラベル集合と異なる。すなわち、実装のラベル集合の要素数は、**Promela**記述のラベル集合の要素数より多い（図** 5.5**）。この実装パターンのＣプログラムでは、**Promela**記述のラベル** `NUMINPUT` **を** `INITIAL` **と** `NUMINPUT` **の** 2 **つのラベルに実装した。**

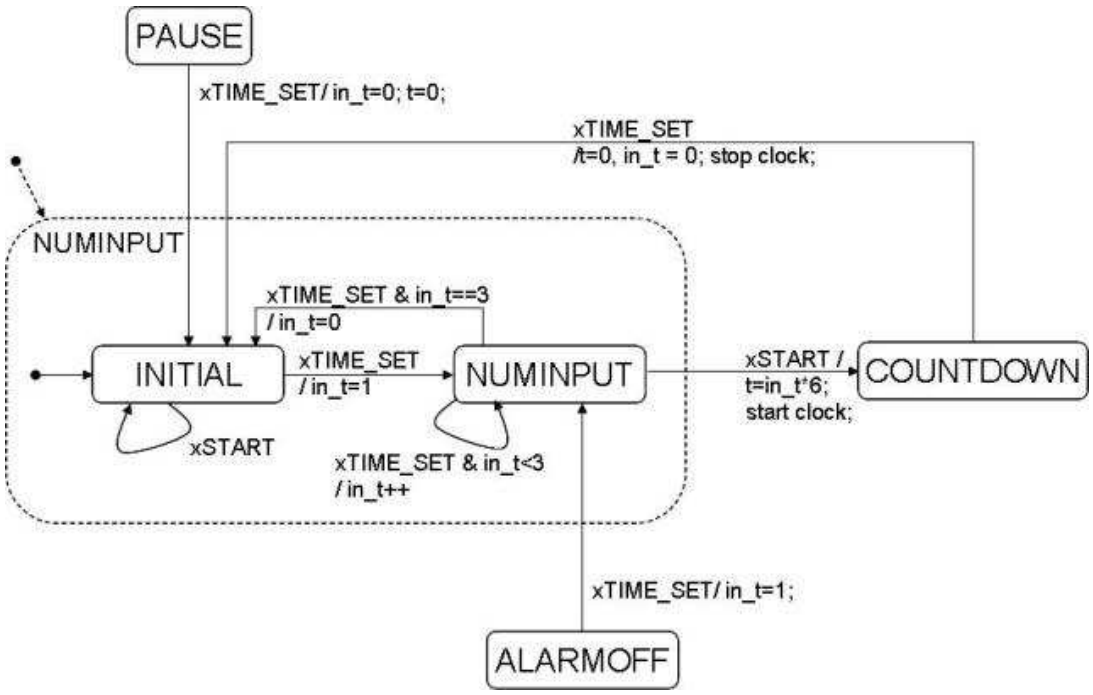

図 5.5: **キッチンタイマの実装拡張１の状態遷移図**

この場合、実装の `INITIAL` **を表現するテストドライバの文字列は、**`"NUMINPUT"`**に変換され、実装のラベル集合を表現するテストドライバは、以下のように符号化された。**

```
#define Timer_MOS0 0 //index of INITIAL state (in_t = 0)
#define Timer_MOS1 1 //index of NUMINPUT state (in_t = 1 or 2 or 3)
#define Timer_MOS2 2 //index of COUNTDOWN state
```

```
#define Timer_M0S3 3 //index of PAUSE state
#define Timer_M0S4 4 //index of ALARM state
#define Timer_M0S5 5 //index of ALARM_OFF state

/*Labels of states*/
/*==> Timer_m0StateLabel[0] = Timer_m0StateLabel[1] = NUMINPUT*/

char Timer_m0StateLabel[Timer_MOSTATEMAX][Timer_MOSTATELabelLengthMAX] =
{"NUMINPUT","NUMINPUT","COUNTDOWN","PAUSE","ALARM","ALARMOFF"};
```

また、制御状態 INITIAL **に関する遷移の入出力記号は、**Promela **記述の入出力記号に対応するように変換された。このとき、**$S(M) = S(D)$ **が成立する。**

5.3.3 パターン３: $L_{Imp} = L_{Promela}$、$V_{Imp} > V_{Promela}$

この実装パターンでは、実装のラベル集合は Promela **記述のラベル集合と同じであるが、実装の制御状態の遷移に影響する変数集合は** Promela **記述の変数集合と異なる。すなわち、実装の制御状態の遷移に影響する変数集合の要素数は、**Promela **記述の変数集合の要素数より多い（図** 5.6**）。**

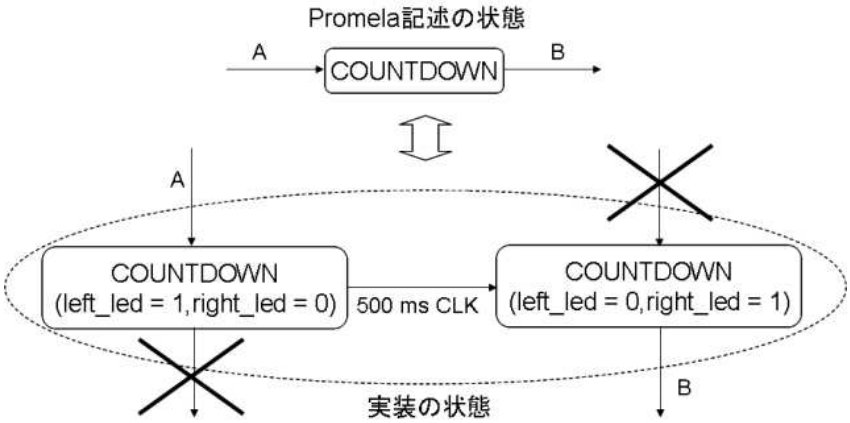

図 5.6: **キッチンタイマの実装拡張２の状態遷移図**

この実装パターンのＣプログラムでは、変数 left_led **と** right_led **を用いて、制御状態** COUNTDOWN **の表示を実装した。すなわち、ラベル** COUNTDOWN **の** Promela **記述の状態に対して、実装では** 2 **つの状態になる。これは、ラベル** COUNTDOWN **の左の** LED **が点滅する状態とラベル** COUNTDOWN **の右の** LED **が点滅する状態であり、これらの状態は隣接している。また、ラベル** COUNTDOWN **の左の** LED **が点滅する状態は、ラベル** COUNTDOWN **の** Promela **記述の状態までの遷移だけを行う。ラベル** COUNTDOWN **の右の** LED **が点滅する状態は、ラ**

ベル COUNTDOWN の Promela 記述の状態からの遷移だけを行う。ラベル COUNTDOWN の左の LED が点滅する状態に入ると、500ms 後にラベル COUNTDOWN の左の LED が点滅する状態へ自動的に遷移する。

この場合、ラベル COUNTDOWN の状態の表現を出力するテストドライバには、特別なマークがある。始点マークは、ラベル COUNTDOWN の各状態の初め、終点マークはラベル COUNTDOWN の各状態の終わりを示す。実装を実行するとき、これらのマークを見つければ、始点マークから終点マークまでのラベル COUNTDOWN の各状態は、テストケースのステップの一つのラベル COUNTDOWN の状態と見なす。また、実装だけにある部分を示す特別な表現を追加した。例えば、Output*、STATE*などである。実装を実行するとき、これらの表現を見つけても、無視する。ラベル COUNTDOWN の状態の表現を出力するテストドライバは、以下のようになる。

```
//Mark for starting of COUNTDOWN states
printf("FOR TEST... >>>ExtraState>>>\n");

left_led = LEFT_ON;
right_led = RIGHT_OFF;

//TEST DRIVER: to print extended parts in Implementation
printf("FOR TEST... Output*: LEFT_ON\n");
printf("FOR TEST... STATE*<COUNTDOWN, %d, %d, %d, LED(%d,%d)>\n",
        input_time,t,alarm_time,left_led, right_led);

sleep 500ms;
left_led = 0;
right_led = 1;

//TEST DRIVER: to print extended parts in Implementation
printf("FOR TEST... Output*: RIGHT_ON\n");
printf("FOR TEST... STATE*<COUNTDOWN, %d, %d, %d, LED(%d,%d)>.\n",
        input_time,t,alarm_time,left_led, right_led);
//Mark for ending of COUNTDOWN states
printf("FOR TEST... <<<ExtraState<<<\n");
```

あるテストケースを行うとき、このソースコードを実装すると、以下の出力が印刷される。この出力は、テストケースの出力 FOR TEST... STATE<COUNTDOWN,1,5,0>に対応する。

```
FOR TEST... >>>ExtraState>>>
FOR TEST... Output*: LEFT_ON
```

```
FOR TEST... STATE*<COUNTDOWN, 1, 5, 0, LED(1,0)>
#after 500ms
FOR TEST... Output*: RIGHT_ON
FOR TEST... STATE*<COUNTDOWN, 1, 5, 0, LED(0,1)>
FOR TEST... <<<ExtraState<<<
```

このとき、$S(M) = S(D)$ が成立する。

今回の実験では、以上のパターンを検討した。それぞれのパターンに対して、$S(M) = S(D)$ が成立するための解法を提案した。

5.4 実験結果の評価

提案手法で生成したキッチンタイマのテストケースの集合を評価するため、この集合のテストカバレッジを確認する。キッチンタイマのテストケースの集合を用いてキッチンタイマの実装でテストを行った。キッチンタイマの実装にバグを混入させ、テストケースの集合でバグを発見できるかどうかテストした。実装のバグは、３つの種類に分けられる。操作エラー、遷移エラー、余分・欠遷移エラーである。操作エラーと遷移エラーは、W-方法で定義されたものであり、余分・欠遷移エラーは、本研究で定義したものである。

5.4.1 キッチンタイマのテストケースのテストカバレッジ

キッチンタイマは、完全に規定されるシステムであるため、余分遷移エラーを含む実装がない。そこで、キッチンタイマで操作エラー、遷移エラー、欠遷移エラーの３つの実装のバグを作った。キッチンタイマのテストケースの集合でこれらのバグを含むキッチンタイマの実装をテストした。このテストの結果、キッチンタイマの操作エラー、遷移エラー、欠遷移エラーの実装のバグが見つかった。

さらに、$D \models p \leftrightarrow M \models p$ を確認するため、キッチンタイマの Promela 記述で検証された性質 p を満たさない実装のバグを作った。これは、キッチンタイマの Promela 記述で検証された性質 `in_t <= 3` を満たさない実装のバグを作った。キッチンタイマのテストケースの集合でこのバグを含むキッチンタイマの実装をテストしたこのテストの結果は、性質 `in_t <= 3` を満たさない実装のバグが見つかった。

5.4.2 キッチンタイマの実装パターンの考察

本研究は $S(M) = S(D)$ を前提とするので、システムが $I_M = I_D$、$S(M) = S(D)$ を満たすように実装する必要がある。キッチンタイマでは、３つの実装パターンが行われた。

1) パターン１（$L_{Imp} = L_{Promela}$、$V_{Imp} = V_{Promela}$）は、最も簡単な場合である。このパターンはかなり珍しい。このパターンでは、Promela 記述と同じように実装するため、

Promela 記述で表現されていること以外は実装されない。そのため、$S(M) = S(D)$ が簡単に成立する。

2) パターン２（$L_{Imp} > L_{Promela}$、$V_{Imp} = V_{Promela}$）は、実際のキッチンタイマを実装するパターンである。このパターンでは、実装の余分ラベルを表現する文字列を Promela 記述のラベルと同じように変換すれば、$S(M) = S(D)$ が成立する。また、余分ラベルに関する遷移の入出力記号も、Promela 記述の入出力記号と同じように変換される。

3) パターン３（$L_{Imp} = L_{Promela}$、$V_{Imp} > V_{Promela}$）は、最も複雑な場合である。このパターンでは、Promela の一つ状態は複数の状態で実装される。今回は、隣接した複数の状態について検討した。$S(M) = S(D)$ を成立させるため、特別なマーク、表現を定義した。テスターは、テストの際、これらのマークと表現を見つけると、それぞれに応じたテスト方法を行う。

今回、本研究では以上のパターンを検討した。それぞれのパターンに対して、$S(M) = S(D)$ が成立するための解法を提案した。

5.4.3 W-方法との比較

キッチンタイマの設計モデルは、ミニマルではないので、W-方法でテストケースを直接生成することができない。W-方法と比べると、本研究の提案手法は、とても扱い易いテストケース生成法である。すなわち、集合 W の生成が必要ではなく、完全に規定される条件を満たさない設計モデルにも適用することができる。

W-方法と違い、本研究で生成されるテストケースの集合は、パステストケースだけではなく、失敗テストケースを含む。また、本研究は、$I_M = I_D$、$S(M) = S(D)$ を前提とする。この２つの前提は、ソフトウェア開発が設計モデルに基づいて実装することに基づく。また、本研究も、実装の内部状態が見えることを前提とする。この前提は、ソフトウェアのテストでテストドライバを用いることに基づく。これらの前提とともに生成されるテストケースの集合を用いて、$M = D$ が成立するかどうか確認することができる。

また本研究は、Promela 記述からテストケースを自動生成する手法を提案した。そのため、本提案手法は、適用性が高い。

第6章　おわりに

6.1　まとめ

本研究では、W-方法を参考にして設計モデルからテストケースを自動生成する手法を提案し、このテストケースを用いて実装と設計モデルの整合性をテストする枠組みを提案した。また、この設計モデルは、Promela で記述され、正しさが Spin により検証されている。そのため、本研究で提案した手法は、Promela 記述からテストケースを自動生成する手法である。また、提案手法を評価するため、キッチンタイマを用いて小規模な実験を行った。

本研究は、ソフトウェアテストにテストケース生成法の一つを提供する。本テストケース生成法は、MBT 手法の問題を解消し、W-方法の問題を克服する。また、本提案手法は、Spin で表現される探索機能を用いて、Promela 記述からテストケースを自動的に生成する。このモデル検査とテストケース生成法の連結により、提案手法の実用性が高くなる。

6.2　今後の課題

- Promela 記述からテストケースを自動生成するプロセスでは、`TestcaseGenerator` を用いて集合 P を自動生成することが行えた。今後は、集合 P と入力記号の集合 I_D の連続を自動的に行う必要がある。この作業は、`dcheck.out` と `P.out` に基づいて自動化できると考えられる。

- 今回は、内部プロセスが一つしかない Promela 記述に着目したため、内部プロセスが２つ以上の Promela 記述を検討する必要がある。内部プロセスが２つ以上の Promela 記述に対して、環境と内部プロセスの間でメッセージを受け渡すこと以外にも、各内部プロセスの間でメッセージを受け渡すことがある。また、FSM の状態は、各内部プロセスの状態の組合わせである。そのため、FSM を再定義する必要がある。そして、Spin で表現される探索機能を用いて、Promela 記述からテストケースを自動生成することができると考えられる。

- $S(M) = S(D)$ の前提が成立することを検討するため、本研究は３つの実装パターンを実験し、それぞれのパターンに対して考察を行った。今後は、ほかの実装パターンについても実験する必要がある。これらの実験結果に基づいて、$S(M) = S(D)$ の前提が成立するための基本的な実装規則を提供するべきであると考えられる。

謝辞

北陸先端科学大学院大学　安心電子研究センター　青木利晃特任准教授には、本研究を進めるにあたり直接の御指導を頂き、ここに感謝の意を表明したいと思います。また、有益な助言をして頂いた、同学　片山卓也教授、島津明教授、小川瑞史教授、矢竹健朗助教に厚く御礼を申し上げます。

その他、貴重な御意見、討論を頂き、共に研究生活を過ごす事ができた青木研究室、片山研究室、岸研究室、Defago 研究室の皆様に深く感謝いたします。

また、研究生活を支援してくださった TIS 株式会社に感謝いたします。

最後に、研究生活を様々な面で支えてくださった家族に感謝します。

参考文献

[1] Ammann, P., Black, P.E., Majurski, W. *Using model checking to generate tests from specifications.* In: ICFEM 1998. 2nd IEEE International Conference on Formal Engineering Methods, Brisbane, Australia, December 1998, p. 46. IEEE Computer Society Press, Los Alamitos (1998)

[2] Beizer, B.(1990). *Software Testing Techniques*. 2nd edition. Boston, MA: International Thomson Computer Press.

[3] T. Chow, **“** Testing software design modeled by finite state machines **”**, *IEEE Trans. Software Eng.*, vol. SE-4, pp. 178-187, Mar, 1978.

[4] Deepinder P. Sidhu, Ting-Kau Leung: Formal Methods for Protocol Testing: A Detailed Study. IEEE Trans. Software Eng. 15(4): 413-426 (1989)

[5] Ibrahim K.El-Far and James A.Whittaker. Model-based Software Testing. *Encyclopedia on Software Engineering*, 2001.

[6] Holzmann, G. J.(2004). *The SPIN Model Checker: Primer and Reference Manual*. Boston: Addison-Wesley.

[7] Liu Wayne. *Selecting Test Cases for Object Oriented Programs*. A proposal presented to the University of Waterloo in partial fulfillment of the Comprehensive Examination Requirements *for the degree of PhD* in Electrical Engineering, Waterloo, Ontario, Canada, 1996.

[8] **穴田啓樹** (2007, December 3). **モデルベース開発とは**. **日経エレクトロニクス**: **組み込みアカデミー**, pp. 133-140.

[9] **山崎真吾** (2008). *OSEK/VDSX* **に基づいたリアルタイムオペレーティングシステムのモデル化と検証**. Master's thesis *for Master's degree*, Japan Advanced Institute Science and Technology, Ishikawa, 2008.

[10] **青木利晃**. *Model Checking Multi-task Software on Real-time Operating Systems*. Japan Advanced Institute Science and Technology, Ishikawa.