深さ優先ヒューリスティック探索による
ソフトウェアモデル検査効率化

前岡 淳

電気通信大学 大学院情報システム学研究科

博士（工学）の学位申請論文

2015年9月

深さ優先ヒューリスティック探索による
ソフトウェアモデル検査効率化

博士論文審査委員会

主査　大須賀 昭彦 教授
委員　田中 健次 教授
委員　多田 好克 教授
委員　田原 康之 准教授
委員　石川 冬樹 客員准教授

Optimization of Software Model Checking with Depth-First Heuristic Search

Jun Maeoka

Abstract

Software model checkers can be used to detect failures in software. However, the state space explosion is a serious problem because the size of the state space of complex software is very large. Various heuristic search algorithms, which explore the state space in the order of the given priority function, have been proposed to solve this problem. This research proposes a new heuristic search algorithm called depth-first heuristic search (DFHS), which performs depth-first search but backtracks at some states that unlikely lead to an error. This paper evaluates the performance of DFHS with enhanced search engine of Java PathFinder which implements DFHS. Experimental results show that DFHS performs better than current algorithms in many cases and DFHS has high possibility to detect failures in software quicker than current algorithms.

深さ優先ヒューリスティック探索による ソフトウェアモデル検査効率化

前岡 淳

概要

モデル検査技術は，対象とするモデルが取りうる全状態を網羅的に探索し，モデルが満たすべき性質への適合を調べることで，モデルの正しさを検証する技術である．近年では，実装コードそのものを扱うソフトウェアモデル検査技術が注目されている．本論文で扱う Java PathFinder (JPF) は，Java プログラムを対象としたモデル検査器であり，プログラム中の不具合の検出や，プログラムの正しさの検証に利用することができる．

ソフトウェアモデル検査技術は，テスト工程における不具合検出に有効である．通常のテストは，想定される様々な入力や環境パラメータに対してプログラムを動作させることで，不具合を洗い出す．しかしながらテストにおいては，非決定的にプログラムが実行される部分があり，必ずしも不具合が発生する状況を再現することができない．例えば，並行プログラムにおいて，複数のスレッドが特定の順序で実行された場合に発生する不具合は，その再現がスレッドの実行順序という非決定要因に依存するため，テストおいて再現できない場合が発生する．これに対して，ソフトウェアモデル検査では，そのような非決定的要因について，全ての選択肢を試行することで，取りうる全てのプログラム状態を再現することで，不具合の発見漏れを防ぐことが可能となる．しかし，モデル検査技術は，モデルの取りうる状態を全て探索するため，モデルの規模に応じて，探索すべき状態数が指数的に増加し，現実的な時間で検証が終了しないという状態爆発への対応が課題となる．とくにソフトウェアモデル検査では，プログラムという抽象度の低いモデルを対象とするため，この状態爆発の問題が顕著である．そこで，この課題を解決する手法として優先度に基づくヒューリスティック探索手法が提案されている．ヒューリスティック探索では，各探索状態について，その先に不具合の含まれる見込みが高いと思われる状態から優先的に探索を進めることによって，不具合の発見を早期化することを目的としている．本研究の対象とする JPF においても探索アルゴリズムとして選択可能である．しかし，従来のヒューリスティック探索には二つ課題があった．一つは，優先度を算出するためのヒューリスティックを適切に選択しないと，優先度が均一となり，横方向に広く探索を行うことになり，探索空間の深いところに到達しづらいという点である．もう一つは，活性等の検証に用いる Linear Temporal Logic (LTL) 検証の実装に適さないという点である．LTL 検証に広く活用されているアルゴリズムは，深さ優先探索をベースとしたものである．これに対して，ヒューリスティック探索は，探索空間中を任意の順序で探索するため，深さ優先ベースの LTL 検証と組み合わせることが困難である．

そこで本研究では，従来手法と異なる考え方に基づくヒューリスティック探索手法を提案する．従来手法は，各状態に対して，その先，不具合に至る可能性の高さを見積もり，探索順序を決定する「優先順序付け」探索である．これに対して提案手法は，優先度付けは行わず，探索順序は深さ優先のままとし，その代わりに，不具合に至る見込みが低いと判断した場合に，その先の枝の探索を打ち切る「枝刈り」探索であ

る．提案手法を Depth First Heuristic Search (DFHS) と呼ぶ．従来手法の優先度の算出および，DFHS の枝刈りの判定は，いずれも各状態におけるヒューリスティックによって決定する．例えば，実行パスにおける各スレッドのインターリーブの状態が代表的な指標である．さらに DFHS では，探索順序は深さ優先ではあるものの，各状態から遷移可能な枝が複数存在する場合に，その状態からどの枝に遷移するかについてのみ順序制御可能とする．このときの指標も，例えばできるだけインターリーブが発生するような枝を先に選ぶ，といった観点で指定する．枝刈りのヒューリスティックと組み合わせることで，複合的な効果が期待できる．従来手法と DFHS は，ある意味でコインの両面であり，どちらの方式が有利であるか (あるいはどちらも 有効でないか) は，探索空間における不具合状態の分布によって異なると考えられるため，従来手法が苦手としていた探索空間で，DFHS が有効にはたらく可能性がある．本研究では，JPF の機能拡張機構を活用して DFHS を実装し，手法の検証を行った．

DFHS は，深さ優先探索を基本としているため，LTL 検証アルゴリズムにも適用可能である．そこで本研究では，LTL 検証についても DFHS の適用を行った．JPF はアサーションの確認，未捕捉例外の発生，デッドロックなどの安全性の検証を主眼としており，標準では活性の検証をサポートしていない．そこで，活性等の検証を可能とするために，LTL 検証のための探索エンジンを実装した．LTL 検証は，対象とするプログラムが，LTL 式で表された活性等の性質を満たすかどうかを検証する．LTL 式を BÃijchi オートマトンに変換したものと，プログラムの状態空間を掛け合わせた同期積に対して探索を実施するため，安全性検証よりもさらに大きな状態空間の探索となる．LTL 検証についても，DFHS の効果が大きく働くと期待できる．さらに，LTL 検証で重要となる公平性の考慮について DFHS を拡張した．LTL 式としては公平性条件を記述せず，反例の公平性充足を確認する手法により，さらなる改善を図った．

拡張した JPF を用いて検証ツール評価用テストプログラムによる DFHS の評価実験を行った．実験の結果，安全性検証，LTL 検証ともに，既存手法よりも多くのケースで DFHS が早期に不具合を発見できることを示し，DFHS による効率化が実現できる可能性が十分に高いことを実証した．

本研究による貢献は，(1) 安全性検証について新しいヒューリスティック探索手法を確立し，ヒューリスティック探索適用の幅を広げたこと，(2) 従来不可能であった LTL 検証に対してヒューリスティック探索を実現したこと，(3) 提案手法を実用のツールとして実装したこと，の三点である．

目 次

図 目 次

表 目 次

Listings

第1章　はじめに

1.1　背景

ハードウェアやソフトウェアの品質向上を目的として，広く用いられる手法がテストである．テストでは，実際の利用環境で想定される様々な環境，入力を洗い出し，これらをパラメータとして，対象を動作させ，不具合を洗い出す手法である．これに対して，モデル検査技術 [6, 1] は，対象とするモデルが取りうる全状態を網羅的に探索し，モデルが満たすべき性質への適合を調べることで，モデルの正しさを検証する技術である．そのため，テストでは発見できないような不具合の発見に効果を発揮する．

モデル検査手法は，当初，ハードウェアの検証や通信プロトコルの検証に適用されて成功を収めてきた [26]．その後，ソフトウェア開発への適用が進展してきた．まず，Spin [20]，SMV [6]，UPPAAL [5, 4]，LTSA [24, 23]，FDR [14] といったツールに代表されるように，ソフトウェアの設計に対する検証への活用が進み，そして近年，実装コードを検証対象とするソフトウェアモデル検査技術 が注目されており，コード開発時，製品テスト時における不具合検出手法，品質保証手法としての活用が期待されている． 例えば，本論文で対象とする Java PathFinder (JPF) [37, 18] は，Java プログラムを対象としたモデル検査器であり，プログラムに含まれる不具合の検出や，プログラムの正しさの検証に利用することができる．従来のソフトウェア設計に対するモデル検査は，設計を表現するモデル記述を独自の言語やツールを用いて定義する必要があるため，技術の習得コストが高く，一般のソフトウェア開発者が導入するには敷居が高かった．これに対してソフトウェアモデル検査は，プログラムコードをそのままの形で扱えるため，広く開発者に利用可能な技術である．

ソフトウェアモデル検査技術は，プログラムの正しさを検証する目的に加えて，テスト工程における不具合検出に有効である．通常のテストは，想定される様々な入力やプログラムのパラメータセットに対してプログラムを実行することで，プログラムが動作する際に発生する様々な状況を再現することで，不具合を洗い出すことを目的とする．しかしながらプログラムの実行には非決定的な要素があり，必ずしも全ての状態を再現できるとは限らない．例えば，並行プログラムにおいて複数のスレッドが特定の順序で実行された場合に発生する不具合は，その再現がスレッドの実行順序という非決定要因に依存するため，テストで再現できない場合が発生する．これに対して，ソフトウェアモデル検査では，そのような非決定的要因について，全ての選択肢を試行し，取りうる全てのプログラム状態を再現することで，不具合の発見漏れを防ぐことが可能となる．

しかし，モデル検査技術は，モデルの取りうる状態を全て探索するため，対象モデルの規模に応じて，探索すべき状態数が指数的に増大し，現実的な時間で検証が終了しないという状態爆発への対応が課題となる．とくにソフトウェアモデル検査では，プログラムという抽象度の低いモデルを対象とするため，この状態爆発の課題が顕著である．

状態爆発の課題に対して，二つのアプローチによる研究が進められている．一つは，モデルの抽象化等により探索状態数を削減することで，モデル検査の持つ全探索網羅の特徴を保ちつつ検証を効率化する手法である [19, 2, 21]．こちらはプログラムの正当性検証に重点を置いたアプローチである．もう一つは，空間の探索方法を効率化することで，不具合発見を早期化する手法である．こちらはプログラムの不具合発見に主眼をおいたアプローチである．後者のアプローチとして，不具合の存在する状態に早く到達できるように探索を制御するヒューリスティック探索が知られている．ヒューリスティック探索は，探索状態の遷移先に不具合が含まれる見込みを様々な観点のヒューリスティック関数により評価し，探索順序を優先度付けすることで不具合発見の早期化を図る手法である（以下，本論文では従来手法におけるヒューリスティック関数を優先度算出関数と呼ぶ）[16, 31]．ソフトウェアモデル検査におけるヒューリスティック探索手法の多くは，膨大な時間のかかる全空間の網羅探索を行わず，限られた時間内での部分的な探索によって不具合を発見しようとする考え方が主流である．本研究の対象とする JPF においても探索エンジンとして選択可能である．

しかし，従来のヒューリスティック探索には二つ課題があった．一つは，優先度を算出するためのヒューリスティックを適切に選択できなかった場合に，優先度が均一となることで横方向に広く探索を行うことになり，探索空間の深いところに到達しづらいという点である．もう一つは，活性等の検証に用いる Linear Temporal Logic (LTL) [25] 検証の実装に適さないという点である．LTL 検証に広く活用されているアルゴリズムは，深さ優先探索をベースとしたものである．これに対して，ヒューリスティック探索は，探索空間中を任意の順序で探索するため，深さ優先ベースの LTL 検証と組み合わせることが困難である．

1.2　本研究の目的と貢献

従来手法における課題を解決するために，本研究では，従来手法と異なる考え方に基づくヒューリスティック探索手法を提案する．従来手法は，各状態に対して，その状態から将来不具合に至る可能性の高さを見積もり，探索順序を決定する「優先順序付け」探索である．これに対して提案手法は，探索順序は深さ優先のままとし，不具合に至る見込みが低いと判断した場合に，その先の枝の探索を打ち切る「枝刈り」探索である．提案手法を Depth-First Heuristic Search (DFHS) と呼ぶ．従来手法の優先度の算出および，DFHS の枝刈りの判定は，いずれも各状態におけるヒューリスティックによって決定する．例えば，実行パスにおける各スレッドのインターリーブの状態が代表的な指標である．さらに DFHS では，探索順序は深さ優先ではあるものの，各状態から遷移可能な枝が複数存在する場合に，その状態からどの枝に遷移するかについてのみ順序制御可能とする．このときの指標も，例えばできるだけインターリーブが発生するような枝を先に選ぶ，といった観点で指定する．枝刈りのヒューリスティックと組み合わせることで，複合的な効果が期待できる．現実的な時間で探索が不可能であるほど状態空間が広大である状況においては，空間の全探索を保証することは無意味であり，不具合をいかに早く発見するか，という観点がより重要である．DFHS は，不具合が未発見であるという出力の場合でも，不具合が存在しないことは保証しない．しかしながら，現実的な時間で既存のヒューリスティック探索よりも効率的に発見できるケースがあることが後述する実験結果でも示された．

従来のヒューリスティック探索は，探索空間全体を見渡して，もっともらしいところを優先的に探索するため，探索フロントを広くとるアプローチをとる．人工知能の分野などでは成功している手法であるが，ソ

フトウェアモデル検査に適用しようとした場合は大きな課題がある．それは，ソフトウェアモデル検査では一つ一つの状態を記憶するコストが高いため，探索フロントを広くとるための探索候補状態を無制限に保持しておくことはできないという問題である．これに対して DFHS は，探索フロントは極小とした上で枝刈りによる局所的な最適化を施すことで，従来のヒューリスティック探索のパラダイムを変える手法である．従来手法と DFHS は，ある意味でコインの両面であり，どちらの方式が有利であるか (あるいはどちらも 有効でないか) は，探索空間における不具合状態の分布によって異なると考えられるため，従来手法が苦手としていた探索空間で DFHS が有効にはたらく可能性がある．

従来手法，提案手法を含むヒューリスティック探索においては，プログラムの不具合に合わせて適切なヒューリスティックを選択することが不具合の早期発見のために重要である．しかしながら，これを検証の実行前に把握することは困難である．したがって，様々な観点のヒューリスティックを試行錯誤しながら適用することになる．本研究では，様々な探索手法，ヒューリスティックによる不具合検出を同時並行で実施することで，全体として不具合発見を早期化することを主眼とする．

DFHS は，深さ優先探索を基本としているため，LTL 検証アルゴリズムにも適用可能である．そこで本研究では，LTL 検証についても DFHS の適用を行った．JPF はアサーションの確認，未捕捉例外の発生，デッドロックなどの安全性の検証を主眼としており，標準では活性の検証をサポートしていない．そこで，活性検証を可能とするために，LTL 検証アルゴリズムをサポートする探索エンジンを実装した．LTL 検証は，対象とするプログラムが LTL 式で表された活性等の性質を満たすかどうかを検証する．LTL 式を Büchi オートマトンに変換したものとプログラムの状態遷移空間を合成した状態遷移系（同期積と呼ぶ）に対して探索を実施するため，安全性検証よりもさらに大きな状態空間の探索となる．そのため，LTL 検証についても枝刈りの効果が大きく働くと期待できる．さらに，LTL 検証で重要となる公平性の考慮について，LTL 式としては公平性条件を記述せず，反例の公平性充足を確認する手法を実装し，さらなる効率化を図った．

提案手法の有効性を検証するために，JPF に対して DFHS を実装し，検証ツール評価用ベンチマークセットを用いて，既存手法との比較実験を行った．安全性検証については，ベンチマークに含まれる全 25 プログラムを用いた従来のヒューリスティック探索との比較実験を行った．まず，従来手法の優先度算出関数に対応する DFHS の枝刈り関数を比較した場合は，3 件の比較中 2 件について，DFHS がより多くのプログラムで不具合を早期に発見できた（比較 1 件目は 16 プログラムで DFHS が優位，比較 2 件目は 14 プログラムで DFHS が優位，比較 3 件目は 13 プログラムで従来手法が優位であった．詳細については本文中で説明する）．次に実際の運用でのユースケースを想定し，複数の従来手法および提案手法を並行で動作させた場合の性能を評価するために，従来手法，提案手法それぞれベストな結果となったもの同士で比較した．その結果，25 プログラム中 17 プログラムについて，DFHS が不具合を早期に発見できた．また，LTL 検証については，独自のプログラムを追加した 28 プログラムを用いて評価実験を行った．既存のヒューリスティック探索は，LTL 検証に適用できないため，ヒューリスティック探索による効率化を行わない LTL 検証アルゴリズムと提案手法との比較実験を行った．まず，従来アルゴリズム（1 通り）と各枝刈りポリシー（8 通り）を比較した場合は，8 件中 6 件の枝刈りポリシーについて，DFHS がより多くのプログラムで不具合を早期に発見できた．実運用を想定した評価として，従来アルゴリズムと DFHS のベストケースとを比較した場合は，28 プログラム中 21 プログラムで不具合を早期に発見した．さらに，DFHS の公平性検証の最適化手法を適用することにより，公平性最適化を実施しない場合と比較して，12 プログラム中 11 プログラム

について，不具合を早期に発見できた．

以下に，本研究による貢献をまとめる．

安全性検証に対する新しいヒューリスティック探索手法の確立

本研究では不具合の早期発見を主眼としている．ヒューリスティック探索は，採用するヒューリスティック関数がその探索効率（不具合発見までの時間）に大きく影響する．しかしながら，問題の複雑さ故に，どのヒューリスティック関数がどの不具合に有効であるかを事前に見抜くことは現実的には困難である．このため実際には，利用可能な計算資源・時間等に応じ，単一または複数のヒューリスティック関数を選択して適用することになる．

計算機資源の制約が強く，従来手法もしくは DFHS のいずれかを選択して適用するような場合は，各ポリシーの比較結果が示すように 3 件中 2 件のポリシーにおいて DFHS が優位であるため，DFHS によって効率化が実現できる可能性が十分に高い．効率化の有効性に関する不確かさを踏まえると，提案手法を常に選択，適用することが実用上有効だと考える．

一方，計算機資源が十分な場合は，従来手法，DFHS 含めて複数のヒューリスティックを並列に検査を走らせることが可能である．近年のクラウド環境など (特に短時間の) 計算資源の入手の容易さを考えるとこのような利用法も十分現実的であるといえる．例えば，従来手法と DFHS の複数のヒューリスティックを並行して走らせた場合，実験結果に基づくベストケースの比較結果が示すように 25 のプログラムのうち，17 のプログラムで DFHS が優位な結果を示しており，この場合も従来技術のみによる並行検査に比べて，効率化が実現できる可能性が十分に高い．

以上の効果は，DFHS という新しいヒューリスティック探索手法が以下の特性を持つことで実現可能になった．

- 従来のヒューリスティック関数から機械的に枝刈り関数を作ることができる
- 実験結果より，DFHS が従来のヒューリスティック探索より平均的に早く不具合を見つける

LTL 検証へのヒューリスティック探索の導入

LTL 検証については，これまで困難であったヒューリスティック探索を LTL 検証に対して実現可能としたことが貢献である．LTL 検証は安全性検証と比べて状態数が多く，状態爆発の問題が顕著であるため，ヒューリスティック探索の効果が高いことが期待される．しかしながら，従来のヒューリスティック探索は，深さ優先探索をベースとする一般的な LTL 検証アルゴリズムと組み合わせること自体が不可能であった．これに対して，提案手法は LTL 探索アルゴリズムと組み合わせて利用することができる．さらに検証実験により，従来の LTL 検証アルゴリズムより早期に不具合を発見できることを示し，DFHS を組み合わせることで実際に効率化できる可能性が高いことを実証した．

実用ツールの提供

本研究では，提案手法について実際の Java プログラムに対して利用可能な実装を提供した．Java のモデル検査ツールとして広く認識されている JPF に対して，DFHS が実際に利用できる実装を提供したことによ

り，提案手法が単に理論として有効であることを示したのみならず，実用への道を開いたといえる．

1.3 本論文の構成

本論文は次章から以下のような構成を持つ．まず第 2 章で背景技術について説明したのち，第 3 章で，ソフトウェアモデル検査における既存アルゴリズムについて述べる．次に第 4 章で従来手法との比較を交えながら提案手法の詳細について説明し，第 5 章で提案手法の JPF による実装について述べる．第 6 章で従来手法と提案手法の比較検証実験に基づく提案手法の評価について説明したのち，最後に第 7 章でまとめと今後の課題について述べる．

第2章　背景技術

2.1　テスト

テストはプログラムの品質を確認するために，不具合の存在を確認するための手法である．開発したプログラムに対して，各種環境パラメータ，入力パラメータを切り替えながら，対象プログラムを実際に動作させることで，不具合を再現させる手法である (図 2.1).

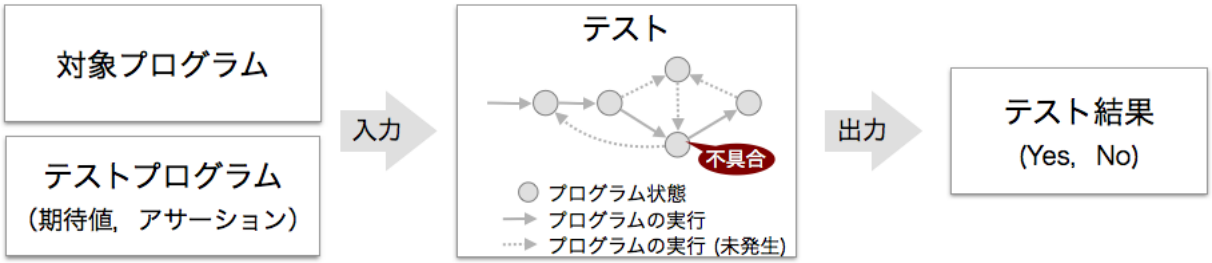

図 2.1: テスト概要

テストはソフトウェアの品質を確認するために広く使われているが，不具合を完全に発見することが困難な場合がある．それは，プログラムの実行状態が，入力や環境パラメータ以外の様々な非決定的要因に依存するため，必ずしも不具合が発生する状態を再現することができないことに起因する．この非決定的要因とは例えば以下のようなものがある．

スレッドスケジューリング 複数のプロセスやスレッドからなるプログラムの場合，OS のスケジューラが制御する各プロセス（またはスレッド）への CPU の割り当て順序や割り当て時間は実行の度に異なる．

ランダム要因 乱数や時刻など，データそのものが非決定要因を持っている場合，実行の度にプログラムの挙動は異なる．

外部環境との連携 通信プログラムのように外部環境の挙動・通信内容が実行の度に異なる場合，プログラムの挙動が確定しない．

リスト 2.1 に示すような Java プログラムを考える．これはデッドロック発生の事例としてよく取り上げられる哲学者の食事プログラムである．このプログラムでは，哲学者 A がフォーク取った後，次のナイフを取得する前に哲学者 B がナイフを取得した場合，両者がお互いにフォークとナイフを取得しようとすることでデッドロックが発生する．各スレッドがそれぞれナイフとフォークを確保するようなタイミングでスレッドが実行された場合のみ発生する不具合である．このような不具合をテストで発見しようとした場合，不具合が発生する順序でスレッドが選択されない限り，不具合を発見することができない．すなわち不具合の発生が偶然性に依存している．

リスト 2.1: デッドロックの発生する例

```
1  /*
2   * フォークを先に取得するスレッド
3   */
4  public class PhilosopherA extends Thread {
5
6    Fork fork;
7    Knife knife;
8
9    public PhilosopherA(Fork fork, Knife knife) {
10     this.fork = fork;
11     this.knife = knife;
12     start();
13   }
14
15   public void run() {
16        synchronized (fork) {
17           log("fork!!");
18           synchronized (knife) {
19              log("A is eating!!");
20                 // do something
21              return;
22           }
23        }
24   }
25 }
26
27 /*
28  * ナイフを先に取得するスレッド
29  */
30 public class PhilosopherB extends Thread {
31
32   Fork fork;
33   Knife knife;
34
35   public PhilosopherB(Fork fork, Knife knife) {
36     this.fork = fork;
37     this.knife = knife;
38     start();
39   }
40
41   public void run() {
42        synchronized (knife) {
43           log("knife!!");
44           synchronized (fork) {
45              log("B is eating!!");
46              // do something
47              return;
48           }
49        }
50   }
51 }
52
53 public class Fork {       // フォークを表すリソースクラス
54 }
55
56 public class Knife{       // ナイフを表すリソースクラス
57 }
```

このような課題を解決する手段として，モデル検査技術が知られている．次節でモデル検査について説明する．

2.2　モデル検査

2.2.1　モデル検査の概要

モデル検査技術とは，対象システムを状態遷移系としてモデル化し，状態空間を網羅的に探索することによって，対象システムが満たすべき性質を検証する技術である（図 2.2）．不具合の有無を検証したり，モデルの正当性を検証したりする目的で用いられる技術である．モデル検査では，非決定要因によって以後のモデルの状態が変わるようなポイントでは，その非決定要因の全ての選択肢を順々に再現させることでモデルが取りうる全状態を検査する．これによって，テストで発見できないような漏れを検証することが可能となる．

図 2.2: モデル検査概要

モデル検査は，当初ハードウェアの検証や通信プロトコルの検証に適用され，成功を収めてきた．近年，ソフトウェアの設計，実装コードそのものの検証に適用されている．ソフトウェア設計の検証は，設計モデル検証と呼ばれ，Spin[20] 等のモデル検査器がある．Spin はモデル記述用の専用言語 Promela (Process Meta Language) を用いて，モデルを定義することで検証する．また，UPPAAL [5] 等は，性能を考慮した検証が可能である．UPPAAL は，状態遷移モデル構築時の遷移条件として時間制約を付加することができる．さらに，検証したい対象をモデルとして定義することなく，実装コードそのものを検証する技術がソフトウェアモデル検査器である．本研究で扱う Java PathFinder (JPF) [37, 18] は，Java バイトコードを対象としたソフトウェアモデル検査器であり，プログラム中で発生するデッドロック，アサート違反，未捕捉例外発生などを検出することができる．JPF は専用のプログラム実行エンジンを用いて，2.1 節で述べたプログラム実行に伴う非決定要因を制御することで，プログラムの取りうる状態を網羅探索する機構を有する．

スレッドスケジューリング　スレッド間に影響がある命令の前で実行を中断，状態を記憶し，スレッド選択の候補を一つ一つ試すことで網羅探索する．

ランダム要因　複数の値の候補値を順次生成する関数を提供しており，これをプログラム内で使用することで，ランダム要因の非決定要因を記述できる仕組みを提供している．例えば，ある変数に想定される値が複数ある場合，この関数を用いて値の候補を発生させることで，それらを順に試すことで網羅探索する．

外部環境との連携　ネットワーク通信やファイル IO など，外部との連携を行う命令については，初回実行時の内容を記録し，再実行時には初回実行時の内容を再現することで，非決定要因の発生するポイントでの再実行を可能にする機構が備わっている．

モデル検査では，安全性，活性といった性質が検証される．モデル検査器は，これらの性質をなんらかの形で定義し，対象モデルがこれらの性質を満たすかを検証する．

安全性 発生してはならない状態に陥らないことを表す性質である．例えば,「例外が発生しない」,「変数が常に正である」などが安全性の性質として定義できる．安全性の検証は，モデルにおける各状態を評価することで行う．そのため，状態空間の各状態をそれぞれ一度訪問する単純な状態空間探索で検証することが可能である．

活性 ある状態が将来必ず発生することを表す性質である．例えば,「終了ボタンを押すと必ずいつか終了する」,「リソースが無限回確保できる」などが，活性として定義できる．活性の検証は，無限長の状態遷移パスを評価することで検証する．そのため，安全性の検証とは異なり，状態空間内で，性質を逸脱するような無限長の状態遷移系列を探すアルゴリズムで探索を行う．後述する Nested Depth-First Search (NDFS) といったアルゴリズムが用いられる．

2.2.2 検証性質の記述

性質を表現する方法として，線形時相論理 (Linear Temporal Logic, LTL) [25] が用いられる．LTL は時間的に変化する性質を表現することができる．Spin 等のモデル検査における性質記述にも広く用いられている．これは，AND, OR, NOT といった基本論理演算に加えて，以下のような LTL 特有の基本演算子を組み合わせて論理式として表現する．なお，p, q はある状態において真偽値が確定する原子命題であるとする（例えば,「変数 x が正である」「スレッド $th1$ が動作可能状態である」といった命題である）．

Gp p が常に成り立っている．[]p と表現することもある．

Fp p がいつか成り立つ．<> p と表現することもある．

Xp 次の時刻で p が成り立つ．

pUq q がいつか成り立ち，しかも，その直前まで p が成り立っている．

例えば,「*req* が発生した場合はいつか *arrive* が発生する，ということが常に正しい」という活性性質を表現する場合，以下のように記述する．

```
[] ( req -> <> arrive)
```

LTL 式は Büchi オートマトンで表現することが可能である．Büchi オートマトンとは，無限長の入力記号列に対して受理可能かどうかを表現すオートマトンである．たとえば上述の LTL 式を否定した論理式のオートマトンは，図 2.3 のように表される．原子命題の集合 $AP = \{req, arrive\}$ からなる無限長の記号列（例：$\sigma = \{\{req\}, \{req, arrive\}, \{req\}, \{req\}, , , , , \}$）が Büchi オートマトンの受理状態を無限回通る場合，その記号列は，Büchi オートマトンに受理されるという．ある入力列が，図 2.3 に示す Büchi オートマトンに受理される場合，その元となった否定の LTL 式を満たすことを意味するため，否定前の LTL 式すなわち，[](*req*− ><> *arrive*) を満たさないということを意味する．このようにして，LTL 式で表現された時相論理を Büchi オートマトンに変換することで，原子命題の入力列が論理を満たすかどうかをチェックすることが可能となる．

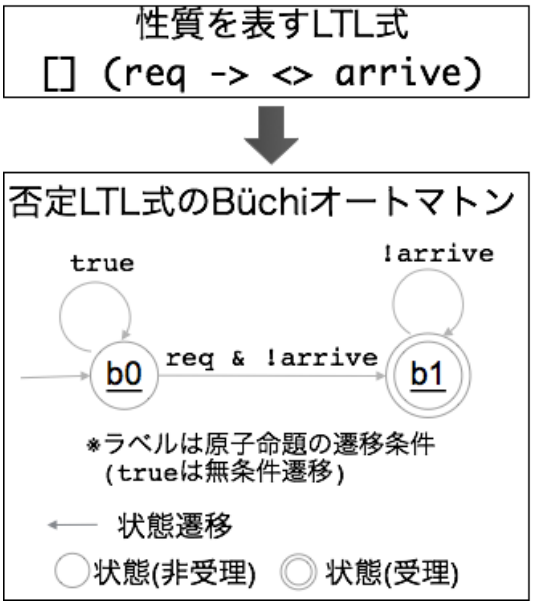

図 2.3: Büchi オートマトンによる LTL 式の表現

2.2.3　ソフトウェアモデル検査器

ソフトウェアモデル検査は，プログラム自体をモデルとして扱い，プログラムを実行しながら検証したい性質からの逸脱の有無をチェックすることで，ソフトウェアの正当性や不具合の有無を確認するモデル検査技術である．本項では，ソフトウェアモデル検査の手順について，ソフトウェアモデル検査器の一つである Java PathFinder (JPF) で実現されている方法をベースに説明する．

図 2.4 にソフトウェアモデル検査器による状態と状態遷移の流れを示す．ソフトウェアモデル検査では，プログラムコードをモデルとして扱うため，モデルの状態はプログラムの実行状態そのものであり，プログラムカウンター，変数値，メモリ内容，モニターのロック状態，などのセットとして定義される．図 2.4 に示すように，初期状態，すなわちプログラム実行開始時点からプログラムの実行を開始し，非決定的要因が発生するまで命令列を実行し，次状態に遷移する．命令列を実行すると，プログラムの内部状態が変化する．ソフトウェアモデル検査ではこれをモデルの状態遷移と捉える．各状態においては，複数の非決定要因を全て選択する必要があるため，後に状態を復元できるように，状態を記憶しながら探索を進めていく．すでに記憶済みのプログラム状態と全く同一のプログラム状態に到達した場合は，同一状態への遷移と捉える．この場合，状態遷移は訪問済み状態への遷移となる．したがって状態遷移空間は，任意の状態への遷移が許される状態遷移グラフで表現できる．この状態遷移グラフは実行に伴って作成されていくため，グラフの全体像は実行前にはわからない．

2.1 節で説明した哲学者の食事プログラムの例を用いて，ソフトウェアモデル検査実行の流れについて説明する．図 2.5 にプログラムの実行と状態遷移の実例を示す．初期状態 s0 からプログラムの実行を開始し，スレッド A の前半部分の命令列を実行することで，新しい状態 s1 に遷移する．次に，本例ではスレッド B の前半部分の命令列を実行し，状態 s2 に遷移する．命令列による状態遷移をトランジションと呼ぶ．このように，一度のトランジションでは，複数の命令がまとめて実行される．原理的にはプログラム状態は，一

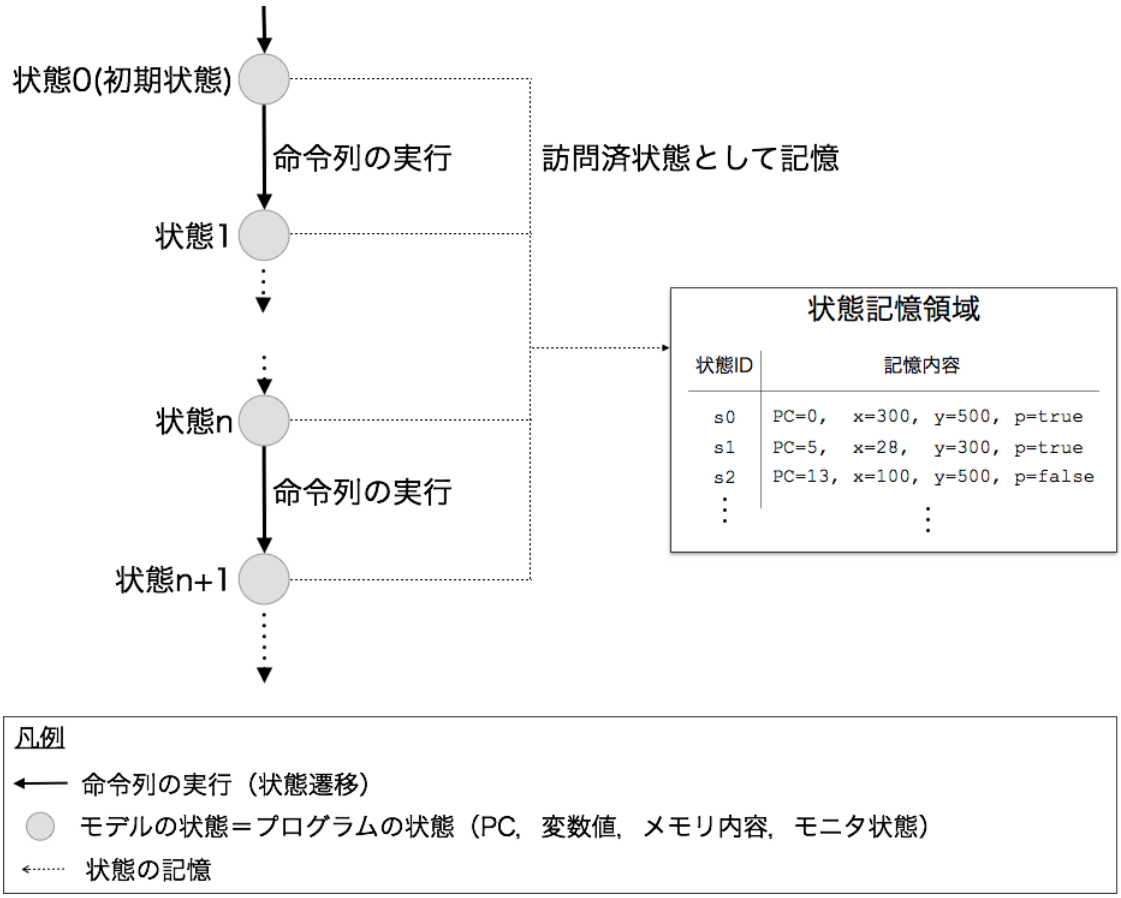

図 2.4: モデルの状態と状態遷移

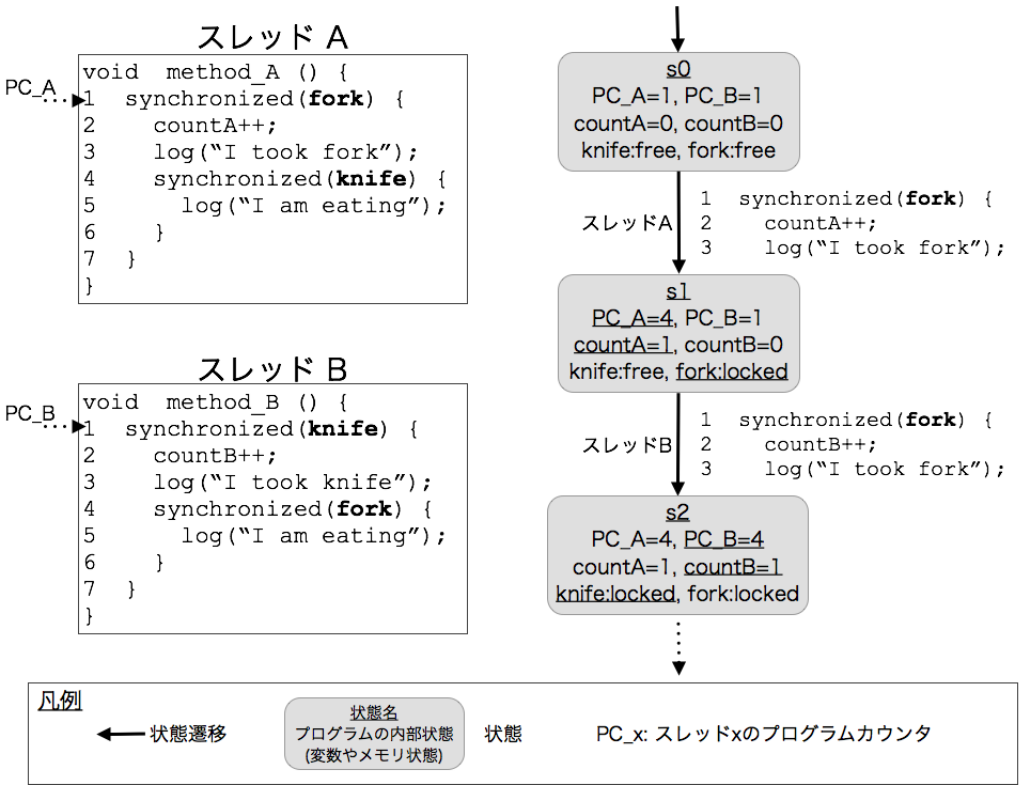

図 2.5: 哲学者の食事プログラムの状態と状態遷移の例

命令の実行ごとに変化する．しかし，一命令ごとに状態を生成，記憶していくと，モデルの状態数が膨大となってしまう．そのため，一度のトランジションでは，検証結果に影響のあるプログラム命令が出現する直前まで連続して実行し[1] [2]，そのときの状態をモデル検査の状態として記憶する．このような状態数の削減手法を半順序簡約と呼ぶ．以後，単に状態と述べた場合は，この記憶されたモデルの状態を示すこととする．トランジションを切る命令は，その命令実行にあたって考慮すべき非決定要因が存在し，その非決定要因の選択によって検証結果に違いが生じるような命令である．以下にこれらについて示す．

共有変数アクセス: スレッド間で共有している変数にアクセスする命令は，スレッドの実行順序が検証結果に影響を及ぼすため，トランジションを終了し，状態を生成する．Java では複数スレッドで共有されるフィールドへのアクセス命令が該当する．

モニターアクセス: モニターの取得，解放を行う命令についても同様にスレッドの実行順序によって検証結果が異なるため，状態を生成する．Java では，synchronized に関する命令が相当する．

乱数生成: モデル検査ツールによっては，変数に対して複数の候補値を生成する機能を有するものがある．この場合，開発者が候補値を設定する関数をプログラム内に記述する，この関数呼び出しに到達した場合は，新しい状態を生成し，値の候補を網羅検証する．以下に JPF による使用例を示す．

```
int multi_value = Verify.random(5);  // 0 から 4 までの値を順次生成する命令
```

この例では，バックトラックのたびに，変数 `multi_value` に対して 0 から 4 の値が順次代入されて実行

[1] 以後，検証に影響ある命令の直前で実行を中断することを，トランジションを切る，と表現する．
[2] 命令の種類とは無関係に，連続実行する命令数の上限を設けて強制的にトランジションを切ることも可能である．

される．

前述の例におけるトランジションの単位を見てみる．図 2.6 に示すように，モニター取得する命令をトランジションの区切りとして，A1, A2, B1, B2 という四つの命令列がトランジションの実行単位となる[3]．

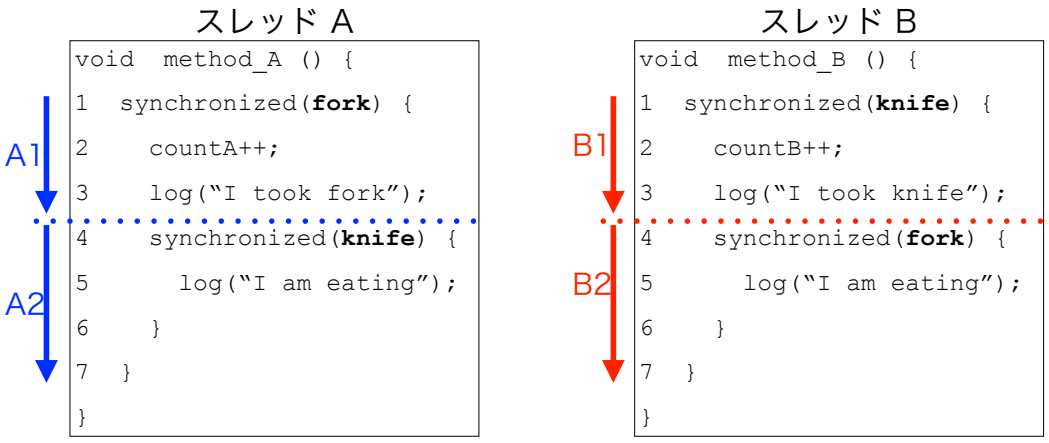

図 2.6: 哲学者の食事プログラムにおけるトランジションの例

生成された各状態では，非決定要因の複数の候補のうちの一つを選択し，さらに次の状態に遷移する．遷移先で既に訪問済の状態に到達した場合や，終了状態に到達した場合，状態遷移の一つ前の状態に戻り，別の候補を選ぶ[4]．この別の候補を選ぶために元に戻ることをバックトラックと呼ぶ．図 2.7 にバックトラックの概念図を示す．この例に示すように，ある状態 n でスレッド 1 とスレッド 2 が選択可能である場合，候

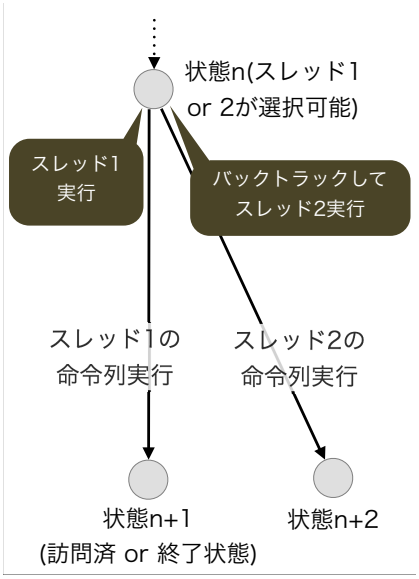

図 2.7: バックトラック

補を一つ選択・実行し（スレッド 1），その先の探索を終えたのち，別の候補を選択し（スレッド 2），実行

[3]JPF では，実際の探索処理は Java のバイトコード単位で実行されるが，わかりやすさのためにソースコードで説明している．
[4]ここでは，後述する深さ優先探索による手順に基づき説明している．

する．このようにして，各状態における非決定要因の全組合わせに対して探索を行う．

以上を踏まえて，哲学者の食事プログラムの探索の流れを図 2.8 に示す．状態 s1 では，スレッド 1，スレッド 2 の二つの候補があり，本例では，まずスレッド 1 を選択する．その先，探索が終了状態に到達すると，状態 s1 にバックトラックし，スレッド 2 を実行することで，状態 s5 に遷移する．s5 ではデッドロックが発生するため，検証結果として，その状態に至るまでの状態遷移パスと実行命令列を出力する．この不具合に至るまでの状態遷移パス出力を反例とよぶ．通常は，最初の反例を発見した時点でツールが終了す

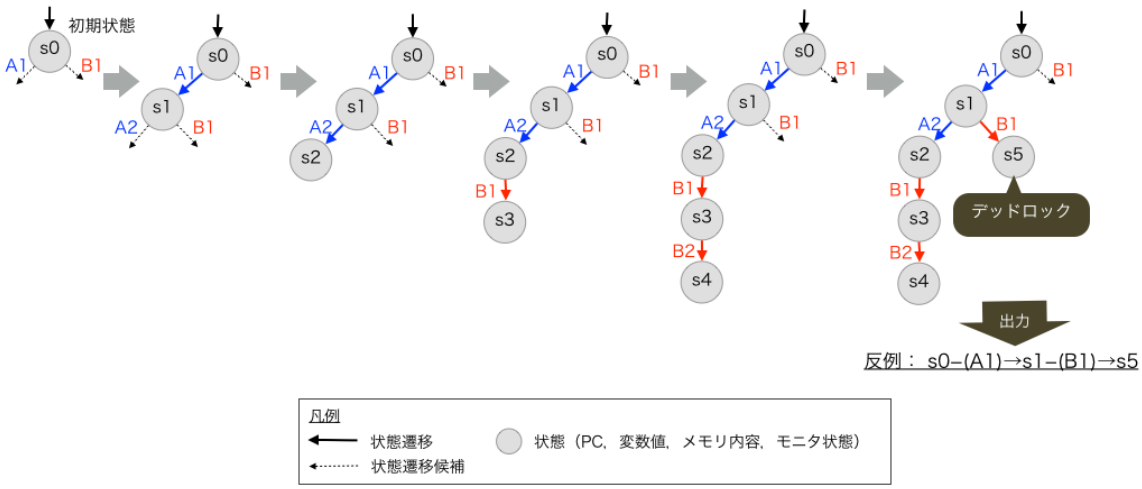

図 2.8: 探索の流れ

るが，探索空間の全体像を理解するために，仮に探索を継続した場合の全体像を図 2.9 に示す．[5]

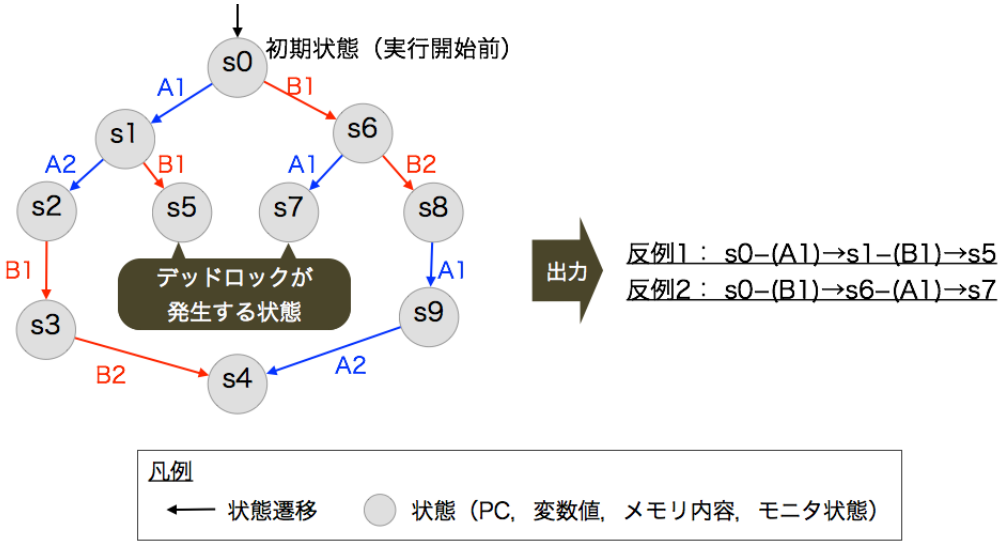

図 2.9: 探索空間の全体像

以上説明したソフトウェアモデル検査技術は，ソフトウェアの正当性検証と，内在する不具合の発見とい

[5]実際，JPF には，反例を発見しても探索を継続するモードがある．ただし，本研究では，最初の反例を発見することを目的としているため，以後の議論では一つ目の反例を発見するまでの性能について議論する．

う二つの目的で用いることが可能である．前者は，ソフトウェアの取りうる状態を全て検査し，仕様逸脱がないことを証明する必要があるため全状態網羅探索が前提となる．しかし，抽象度の低いソフトウェア実装に対してモデル検査を適用するソフトウェアモデル検査では，状態爆発の問題が顕著であり，現実的な時間で全探索空間を検査することは困難である場合が多い．本研究では，後者の不具合発見を目的とする．

2.2.4 Java PathFinder (JPF) の概要

JPF は，Java バイトコードを対象としたソフトウェアモデル検査器である．独自の JavaVM によって Java バイトコードを実行しながら，前節で説明したモデル検査を実施する [37]．図 2.10 に，JPF の機能構成を示す．JPF は大きく，プログラム実行部と探索エンジンからなる．プログラム実行部は，Java バイトコードを実行する部分であり，モデル検査におけるバックトラックを実現するために，プログラムの実行状態を保持する機能を持つ．探索エンジンは，状態遷移を制御するアルゴリズムの実装部である．標準では深さ優先探索が有効となっており，加えて第 3 章で述べる最良優先探索が実装されている．探索エンジンがプログラム実行部に対して，次状態への遷移，あるいはバックトラックの指示を出すことで探索を制御する．なお，探索エンジンを独自実装する仕組みが提供されており，独自の探索アルゴリズムを作り込むことも可能である（後述する LTL 検証の実装では，この機能を利用して実装している）．

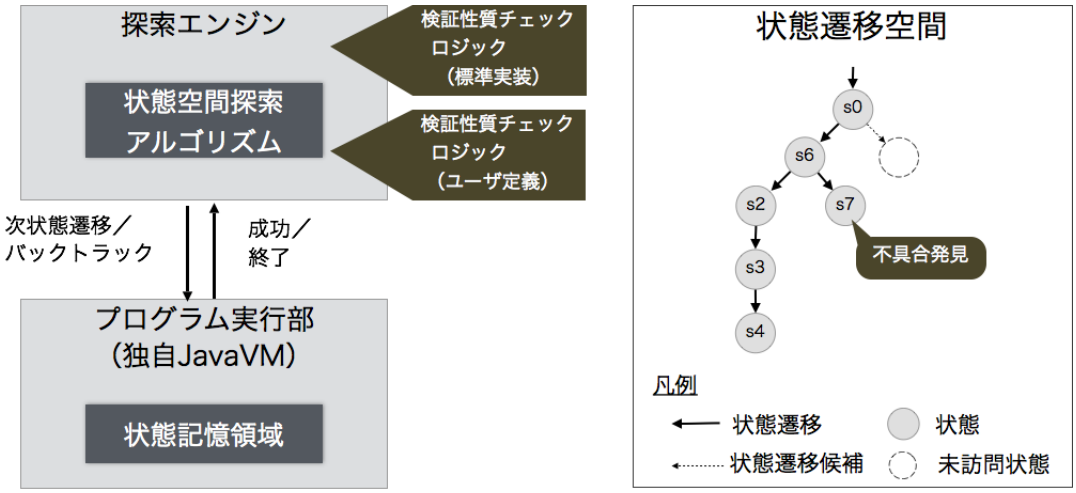

図 2.10: JPF の機能構成

JPF では安全性の検証が可能である一方，活性の検証はサポートされていない．安全性の検証は，全状態への単純な訪問で検証可能であるため，そのような探索エンジンが標準提供されている．検証したい性質を表現するロジックは，Java プログラムとして記述する．標準の検証項目として，デッドロック検出，未捕捉例外検出，アサート検出がサポートされており，対象とするプログラムの修正や，性質の記述なしに検証可能である．それ以外の安全性検証を行う場合は，ユーザ固有の検証ロジックを定義する．検証ロジックは実行エンジンのリスナーとして機能し，指定したタイミングで呼び出される．呼び出された検証ロジックは，プログラムの状態をチェックし，検証したい性質から逸脱していないかを判定する．性質逸脱があると判定した場合は，検証ロジック内で不具合発生のフラグを立てて実行エンジンに通知し，実行エンジンがその時点での状態遷移パスを反例として出力することで所望の性質逸脱を発見することができる．検証ロ

ジックの呼び出しタイミングは，状態遷移毎，バイトコード実行毎，などが指定可能であり，検証したい性質に合わせて選択する．

JPF は Java プログラムをそのままの形でモデルとして扱うことが可能なため，設計モデル検査ツールのような独自言語によるモデル定義を行う必要がなく，手軽に導入できる利点がある．その反面，プログラム実装という抽象度の低いモデルを扱うため，先に述べた半順序簡約などの状態削減を行ってもなお，状態爆発の課題が顕著である． また，安全性検証をのみを対象としており，活性検証を行えない．

そこで本研究では，ソフトウェア検査技術に基づく実用的な不具合抽出ツールの提供を目的とし，従来手法とは異なるアプローチによる効率化手法を提案し，JPF 上に実装する．さらに，LTL 検証のための拡張も実施する．

2.3　関連研究

ノードと有向エッジからなる状態空間の中から，目的とするノードを探索する問題を探索問題と呼ぶ．深さ優先探索や幅優先探索といった原始的なアルゴリズムの他，人工知能の分野において，目的とする状態をできる限り早く発見する，あるいは最適解を求める手段として，ヒューリスティック探索が知られている [33]．深さ優先探索や幅優先探索といった基本的な手法では，探索空間の情報のみを用いて探索を進めていく．これに対して，Best-First[28], A*[17], Beam といったヒューリスティック探索では，各状態の「良さ」を評価するヒューリスティック関数を用いて探索順序を制御する．

Best-First Search: 最もヒューリスティックの良い状態を優先して探索する方法

A* Search: 状態の良さの評価に，探索状態の深さをパラメータとして加味したヒューリスティックによって探索する方法

Beam Search: 幅優先探索を基本として，各探索の深さでヒューリスティックの良い状態を k-best だけ探索候補として残す方法

モデル検査における状態空空間の探索に対してヒューリスティック探索を適用し，状態爆発に対処しようとする試みが知られている [13, 12, 16, 31]．Edelkamp, Lluch-Lafuente, Leue らによる HSF-SPIN[12] は Spin[20] にヒューリスティック探索を組み合わせることで，より短い反例の提示を可能とするツールである．Gorce, Visser らによる JPF でのヒューリスティック探索の適用 [16] では，有効なヒューリスティック関数としてブランチ選択最大化ヒューリスティック，インターリーブ最大化ヒューリスティックが例示されており，それぞれ，特定条件のみ不具合が発生するプログラム，スレッドデータ競合を含むプログラムに有効であることが示されている．Runga，Mercer らの手法 [31] は，静的解析ツールの出力した警告箇所に至るまでのパスに基づく優先度と，従来のヒューリスティック値による優先度を組み合わせる．これによってヒューリスティック値のみを用いた場合よりも短時間で不具合が発見できるケースが示されている．

また，伝統的なヒューリスティック探索とは異なるアプローチで，探索を効率化する手法も提案されている [29, 27]．Parízek, Lhoták らの手法 [29] は，深さ優先探索処理において，探索の深さによって可変の確率でランダムにバックトラックする手法である．JPF による実装において，既存のヒューリスティック手法との比較で優位な結果を示している．この手法は，(枝刈り関数にランダム関数を使用することにより) 我々の提案手法の特別な場合であると考えることができる．Musuvathi, Qadeer らによる Context-Bounded Search[27]

はコンテキストスイッチの回数に閾値を設けることで探索空間を削減する方法であり，コンテキストスイッチの回数を数回に制限しても多くの不具合が検出可能であることが示されている．この Context-Bounded Search は，既存のヒューリスティック探索や我々の提案手法とは独立の観点に基づいている．実際後にみるように，枝刈り関数として，コンテキストスイッチ回数を採用することで，提案手法と組み合わせることが可能である．

また，LTL 検証においては，オートマトンベースのアプローチが用いられる [36]．探索空間は，ターゲットモデルの状態空間と LTL 式を表す Büchi オートマトンの同期積で表される．有力な探索アルゴリズムとして，Nested Depth-First Search[7]．および，強連結成分（Strongly Connected Components, SCC）による手法 [8] の二つが知られている．これらは，いずれも深さ優先探索に基づくアルゴリズムである [34] .

並列探索の分野では，幅優先探索の手法も示されている [3]． Sun らは，公平性を含んだ LTL 検証に対して，SCC ベースのアルゴリズムを提案している [35].．彼らの手法では，SCC を探索するのに，深さ優先探索により全空間の探索を行う．また，反例が見つかった後に公平性をチェックすることで，公平性検証を効率化するアルゴリズムが示されている．本論文で提案する枝刈り機能についても，今後の研究においてこの SCC ベースのアルゴリズムに適用可能である． 本研究ではまず，最もよく知られた LTL 検証アルゴリズムに対して手法を適用した．しかしながら，より効率的な LTL 検証も提案されており [15, 9]，将来的にはこれらに対する提案手法の適用も可能であると考えられる．

JPF に対する LTL 検証拡張が幾つか提案されている [22, 10]．Lomabard の実装 [22] では，LTL 式をプログラム中のアノテーションとして記述し，NDFS で検証する方法が提供されている．また，Coung らの手法 [10] は，軽量探索を主眼としてメソッドコールに着目したアプローチが取られている．これらの手法とは異なり，本研究における実装では，公平性を表現するための原子命題 (Atomic Proposition) を用意している．公平性を考慮したアルゴリズムも研究されている．特に強公平性に着目したオートマトンベースの効率的な探索手法が提案されている [11]．本研究では，検証よりもエラー発見に重点を置くことによる異なるアプローチを取る．

省メモリの観点においては，Bitstate Hashing や State Space Caching といった手法が用いられる [15, 30]. これらは，伝統的なヒューリスティック探索や，本論文で提案する手法に対しても独立に適用可能である．実際，本研究で用いているモデル検査ツール JPF においても，Bitstate Hashing が用いられている．

第 1 章で述べたようにヒューリスティックモデル検査は，全空間の探索を必ずしも実施しないことから不具合の非存在を示すことはできない．このことから，この技術はむしろパステスト生成に近いという印象を与えるかもしれない．しかし，パステスト生成がプログラムという静的な空間の網羅性向上を図るのに対して，(ヒューリスティックモデル検査を含む) モデル検査は，プログラム実行時の動的な状態からなる空間を探索する点において基本的に異なるアプローチである．

第3章　モデル検査における既存アルゴリズム

本章では，従来の探索手法である深さ優先探索，およびヒューリスティック探索の一つである最良優先探索について述べる．また，本研究における LTL 検証の基本アルゴリズムである Nested Depth-First Search について説明する．

3.1　深さ優先探索

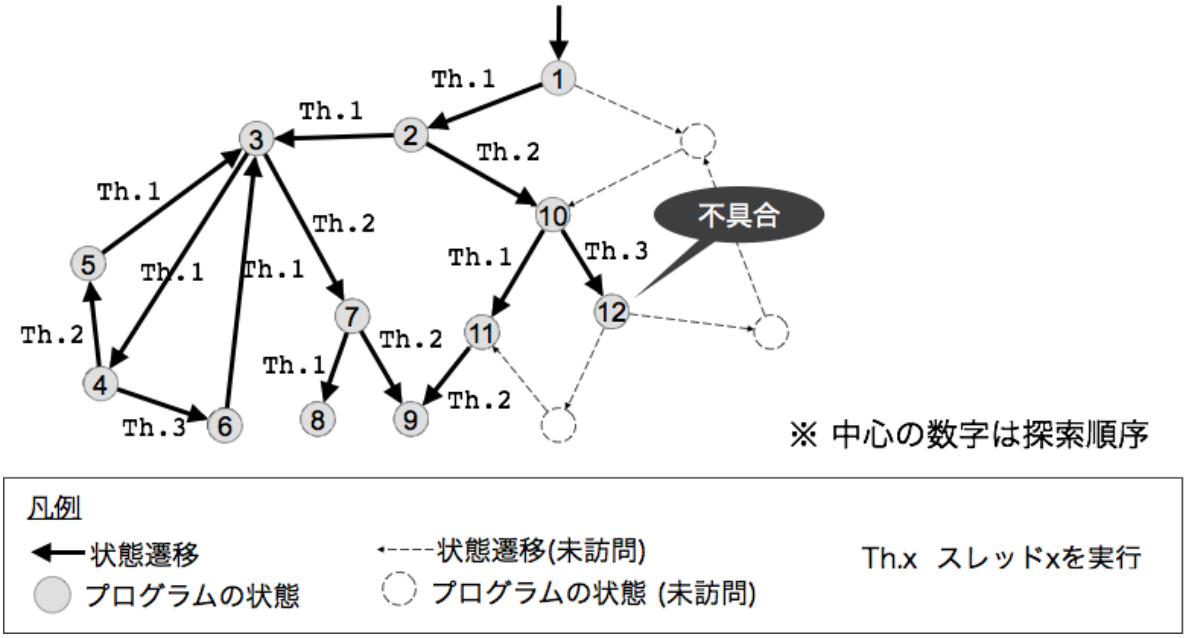

図 3.1: 深さ優先探索

本節では，伝統的な探索手法である深さ優先探索 (Depth-First Search，DFS) について説明する．後述の実装において用いている JPF では，深さ優先探索が標準の探索方法となっている．

深さ優先探索の流れを図 3.1 に示す[1]．この図では，ノードで表現された状態と，ノードからノードへの矢印として表された状態遷移からなる状態遷移モデルに対し，初期状態を起点に，探索空間に存在する不具合状態を，探索する様子を模式的に表している．図中の数字は，状態を探索する順番を表している．なお，図中では，探索空間全体を表示しているが，実際の探索では，不具合が発見された時点で，不具合に至るまでの探索パスを反例としてを表示して終了するのが一般的である．

深さ優先探索は，初期状態から探索を開始し，順次状態遷移を行いながら探索を継続する．状態の遷移

[1] ここでは，分かりやすさのために深さ優先探索をソフトウェアモデル検査におけるプログラムの状態遷移系に対して適用した例を示しているが，深さ優先探索は，状態遷移系の探索問題に一般に利用できる手法である．

は，遷移先が存在する限り深さ方向に探索を継続する．ある状態から遷移可能な枝が複数存在する場合は，なんらかの基準によって枝を一つ選択し，探索を継続する．探索済みの状態に再到達した場合，は，それ以上の探索は実施済みであるため，一つ前の遷移先状態における別の枝の探索を行う．このように，遷移先の状態から先の探索が全て終了してから，別の遷移先を探索することを特徴とする．図 3.1 では，12 番目に探索した状態に不具合が存在した例を示している．

リスト 3.1 に深さ優先探索のアルゴリズムの疑似コードを示す．`get_next_successor()` は，その状態から遷移できる状態候補のうちの一つを順番に返す処理を表す．`Stack` は，探索パス上の各状態を管理するスタックである．`push()` は，スタックに状態を保存する処理である．`Visited` は，探索済の状態を管理するための集合を表し，ここに登録されている状態に再度遷移した場合には，それ以降の探索は行わず，別の状態を探索する．

リスト 3.1: 深さ優先探索のアルゴリズム

```
1  depth_first_search:
2    stack Stack = [S0]
3    set Visited = {S0}
4    while (S = top(Stack)) != null
5       S' = get_next_successor(S)
6       if S' == null
7           pop(Stack)
8       else if is_error_state(S')
9           report_error(S')
10          return
11      else if not exists(Visited, S')
12          add(Visited, S')
13          push(Stack, S')
```

ソフトウェアモデル検査の探索における各探索状態とはプログラムの実行状態そのものである．新たな状態に遷移する度に，後のバックトラックに備えてプログラムの状態を保存しておく必要があるが，JPF に代表されるソフトウェアモデル検査器で良く用いられる手法では，プログラムが使用しているメモリを全て保持しておくため，この状態保持のコストが非常に高い．本節で説明した深さ優先探索では，スタック上に格納されているプログラム状態を保持しておく必要がある．このような制約から，スタックサイズに上限を設けることが一般的であり，この上限を超えた場合は，メモリエラーとして探索を終了するのが一般的である．

3.2 最良優先探索

探索順序を優先度付けするヒューリスティック探索には，空間の探索方法や使用するヒューリスティック関数によって様々なバリエーションがある．本節では Gorce,Visser らの検証実験 [16] で最も良い結果を得た方式であり，かつ，JPF の標準配布にも含まれている最良優先探索 (Best-First Search) について説明する．

最良優先探索の流れを図 3.2 に示す[2]．最良優先探索は，探索空間の情報 (探索空間の親子関係，深さ，幅など) だけでなく，現在検査対象としている状態に付随した情報 (例えば，スレッドの状態，変数の状態)，検査対象とする不具合に関する情報 (例えば，不具合発生箇所との距離) といった観点から定義されたヒューリスティック関数によって，各探索状態の遷移先に不具合が含まれるかどうかを数値化し，最も可能性が高

[2]深さ優先探索同様，最良優先探索の適用はソフトウェアモデル検査に限定されるものではない．

いと評価されたものから優先して探索を進めていくことを特徴とする．以後，最良優先探索で用いられるヒューリスティック関数を優先度算出関数と呼ぶことにする．図 3.2 の例では，初期状態から探索が開始され，状態遷移とともに新しい状態が生成されていく様子を示している．ある状態から状態遷移が発生し，新しい状態が生成されると，その状態に対する優先度スコアが算出される．そして既に生成済みの状態の優先度スコアを含めて，もっとも優先度が高い状態が次に展開される（図中では，各状態における優先度を $h =$ で表現している．また，JPF の実装にならって，値が小さいほど高優先度としている）．

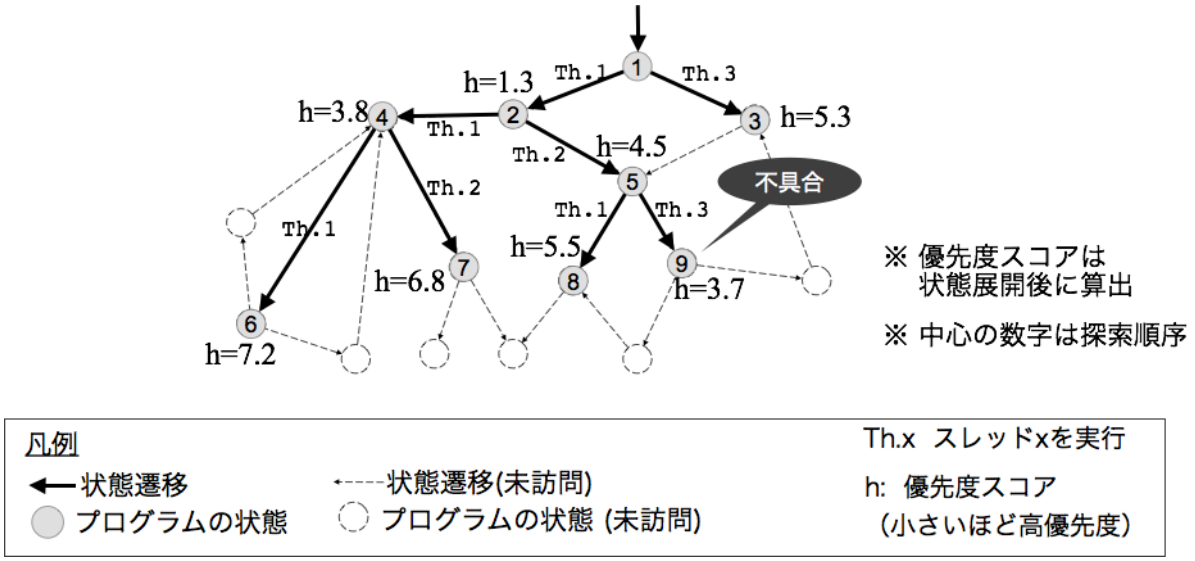

図 3.2: 最良優先探索

処理手順の概要を図 3.3 に示す．探索候補の状態が探索キューによって優先度順に管理され，最も優先度の高い状態から順次探索を進める．探索キューから最も優先度の高い状態を取出し，その状態から遷移可能な全ての状態に対して順番に遷移する．遷移先の各状態は，優先度算出関数によって優先度を算出し，探索キューに格納する．以上の処理を，初期状態を探索キューに格納してから，探索キューが空になるか，不具合を発見するまで繰り返し実行する．各状態から遷移先となる候補状態をすべて展開するため，幅優先探索のように，探索空間を横方向に広く探索する傾向を持つ．

リスト 3.2 に最良優先探索のアルゴリズムを表す疑似コードを示す．`h()` は，状態に対する優先度算出関数であり，順序付け可能な数値を返す．`Queue` は，遷移先状態を管理する優先度付きの探索キューを表す．`insert()` は，算出した優先度に合わせてソートした位置に状態を追加する処理である．

なお，使用できるメモリ量の制約などの理由により探索キューサイズに上限を設けることがある．この場合，`insert()` 処理で格納する際にキューサイズを超えた場合は，キューの中で最も優先度の低い状態が削除される (探索候補から外される)．したがって，不具合が存在しても不具合を報告せずに探索が終了することがある．

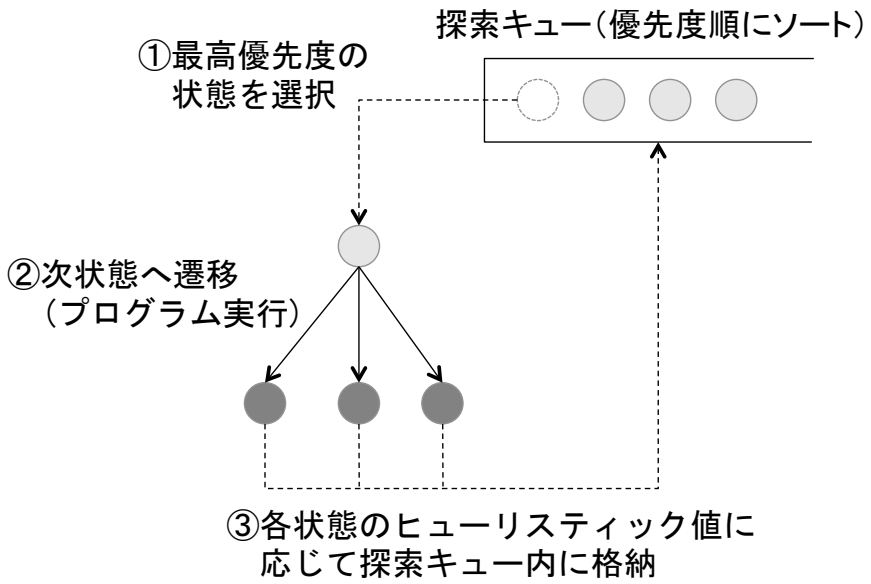

図 3.3: 最良優先探索の処理手順

リスト 3.2: 最良優先探索のアルゴリズム

```
1  best_first_search:
2    priority_queue Queue = [S0]
3    set Visited = {S0}
4    while (S = remove_best(Queue)) != null
5       while S' = get_next_successors(S) != null
6           if is_error_state(S')
7               report_error(S')
8               return
9           else if not exists(Visited, S')
10              Hval = h(S')
11              insert(Queue, S', Hval)
12              add(Visited, S')
```

以下，最良優先探索をソフトウェアモデル検査に適用した場合の探索の特徴について説明する．ソフトウェアモデル検査に適用した場合，優先度算出関数は，探索の深さといった探索空間の構造，スレッドの選択履歴，ブロックしているスレッド数といったスレッドの状態などによって算出される．スレッドの状態に着目した優先度算出関数 h() の例について，Gorce,Visser らの論文 [16] より引用する．

Interleaving(n): 最新の状態遷移で実行されたスレッドが，直近 n 回の状態遷移で実行された頻度が少ないほど，優先度の高いヒューリスティック値を算出する．

MostBlocked: ブロックしているスレッド数が多いほど，優先度の高いヒューリスティック値を算出する．

Random: 状態とは無関係にランダムなヒューリスティック値を算出する．

ここで例示した関数を含めて，従来のヒューリスティック探索に関する研究では，いかに「不具合のありそうな状態」を精度よく評価するかといった観点から様々な優先度算出関数が検討されている．しかし，適切な優先度算出関数をあらかじめ予想することは困難である．そのため，本来重要である探索パスが高い優先度を持たないことがあり得る．また，優先度算出関数が適切に設定されていたとしても，プログラムの初期段階など，優先度の優劣が少ない状態においてもやはり将来重要となる探索パスが正しく評価されな

いことが考えられる．

ソフトウェアモデル検査のヒューリスティック探索では，優先度管理する探索キューサイズを制限する必要がある．3.1 節でも述べたようにプログラムの状態の保存はコストが高い．また，最良優先探索は横方向に広く探索する傾向があるので，深さ優先探索と比較して多くの状態を保持しなければならない．また，多くの候補を探索キューに保持すると逆に探索効率が低下してしまう問題もある．このような理由から探索キューに制限を設けた場合，上述のような優先度が低く評価されてしまった重要な探索パスは，キューから溢れて探索されないことがある．

最良優先探索のもう一つの問題は，LTL 検証の一般的なアルゴリズムとの相性が悪いという点である．LTL 検証は深さ優先探索ベースのアルゴリズムを用いることが一般的であるが，最良優先探索は，探索順序が上記のとおり特定できないためである．

3.3　LTL 検証アルゴリズム

安全性の検証は，探索空間内の各状態への単純な訪問により実現可能である．一方，活性の検証は，無限長の状態遷移パスに対する検証であるため，単純な全状態への訪問では検証することができない．そのため，時相論理を用いた検証を行う必要がある．ここでは，時相論理の一つである線形時相論理 (Linear Temporal Logioc, LTL) を用いた活性の検証について説明する．

LTL 検証では，検証したい性質を表現する LTL 式を定義し，この論理式を満たさない状態遷移パスが対象システムの状態遷移系に存在するかを確認することで検証を実施する．状態遷移パスは，対象システムにおける状態遷移の列として定義されるため，対象システムが取りうる状態遷移列が，LTL 式を満たすかどうかをチェックすることで検証を実施する．モデル検査では，LTL 式の内部表現として Büchi オートマトンが用いられる．有限状態オートマトンが，有限の入力記号列に対する受理を判定するのに対して，Büchi オートマトンは，無限長の入力記号列に対する受理を判定できる．Büchi オートマトンでは，受理状態を無限回通る入力記号列を受理と判定する．検証したい性質を表現した LTL 式の否定論理式を表現する Büchi オートマトンを作成し（以後，LTL 式の否定論理式を否定 LTL 式，と呼ぶことにする），そのオートマトンと与えられたモデルの状態遷移空間を合成したオートマトンを探索空間として構築する．これが探索対象の空間となる．

`[] (req -> <> arrive)` という LTL 式を例に考える．まず，否定 LTL 式，`! [] (req -> <> arrive)` を表す Büchi オートマトンを構築する．図 3.4 に変換後のオートマトンを示す．

また，図 3.5 に，プログラム実行に伴う状態遷移空間の全体像の例を示す[3]．プログラム状態が変わることにより，原子命題の値が変化する可能性があるため，図中の状態の中にこれを記している．プログラムの状態を構成する要素として他の変数の値と同様にこの原子命題の値も含まれる．

プログラムの実行に伴って，原子命題の値が変化するため，この変化に伴って，否定 LTL 式の Büchi オートマトンの状態も遷移する．このようにプログラムの状態遷移と否定 LTL 式の Büchi オートマトンの状態遷移は同期的に発生するため，これを合成したオートマトン（以下，同期積と呼ぶ）を作成する．同期積の

[3]すでに述べたようにプログラムの状態空間は実行に伴って構築されるため事前にはわからないが，ここでは理解を助けるためにあえて全体像を表示して説明する．

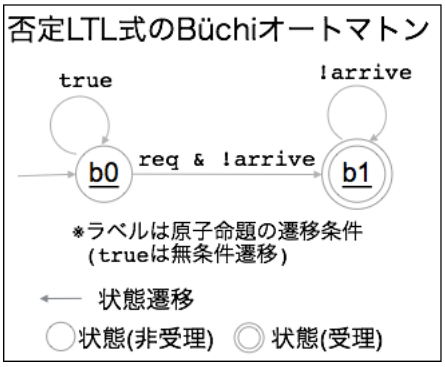

図 3.4: LTL 式の否定を表すオートマトンの例

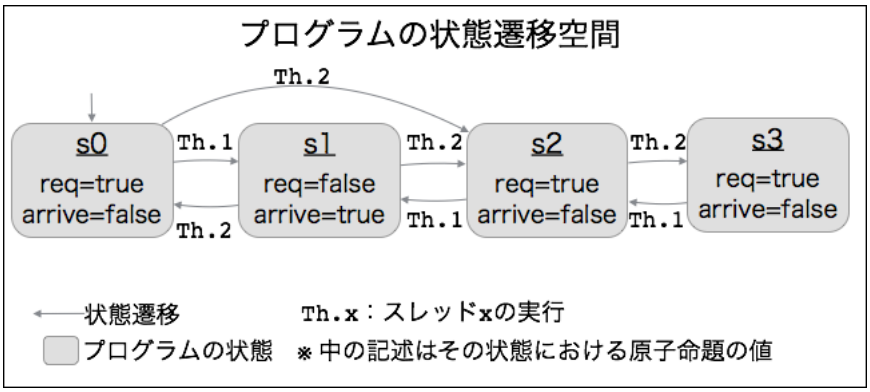

図 3.5: プログラムの状態空間の例

例を図 3.6 に示す．同期積では，否定 LTL のオートマトンの状態とプログラムの状態をセットとしたものを一つの合成状態とする．このとき，否定 LTL の状態が受理状態である場合，合成後の状態も受理状態とする[4]．このようにして合成された同期積に対して，2.2.2 項で説明した Büchi オートマトンの受理判定を行う．すなわち，受理状態を無限回通るような状態遷移ループが存在するかを探索する．

図 3.7 に反例パスの例を示す．本例では，点線で示した無限長の状態遷移パスが，受理状態 (b1, s2) または (b1, s3) を無限回通る（以後，状態遷移パス上のループ部分を状態遷移ループと呼ぶ）．受理状態を無限回通るようなパスが発見された場合，この同期積は，否定 LTL 式のオートマトンを満たす状態遷移パスが存在する．すなわち，元の LTL 式を満たさないケースがあるということを意味する．

以上説明した LTL 検証を行うためには，まず受理状態を探索し，次にその受理状態を通る状態遷移ループを探索する，という二つの探索を行う必要がある．Nested Depth-First Search (NDFS) は，LTL 検証の実装で広く用いられるアルゴリズムである．NDFS は，同期積に対して二種類の深さ優先探索を用いて反例を探索する．リスト 3.3 に NDFS の疑似コードを示す． まず受理状態を探すために，一つ目の深さ優先探索（Blue DFS と呼ぶ）を開始し（6 行目），再帰的に適用する (13 行目から 15 行目)．Blue DFS では，各状態から先の遷移を全て探索し終えたのちに，その状態からバックトラックする直前にその状態が受理状態であるかどうかの判断を行う．その状態が受理状態でなかった場合は通常のバックトラックを行うが，その状態が受理状態であった場合は，Blue DFS を一時中断し，ループ遷移を探索するためのもう一つの深さ優先探

[4]終了のあるプログラムの場合は，終了状態については，特別に自己ループを付与して検証を容易化する．

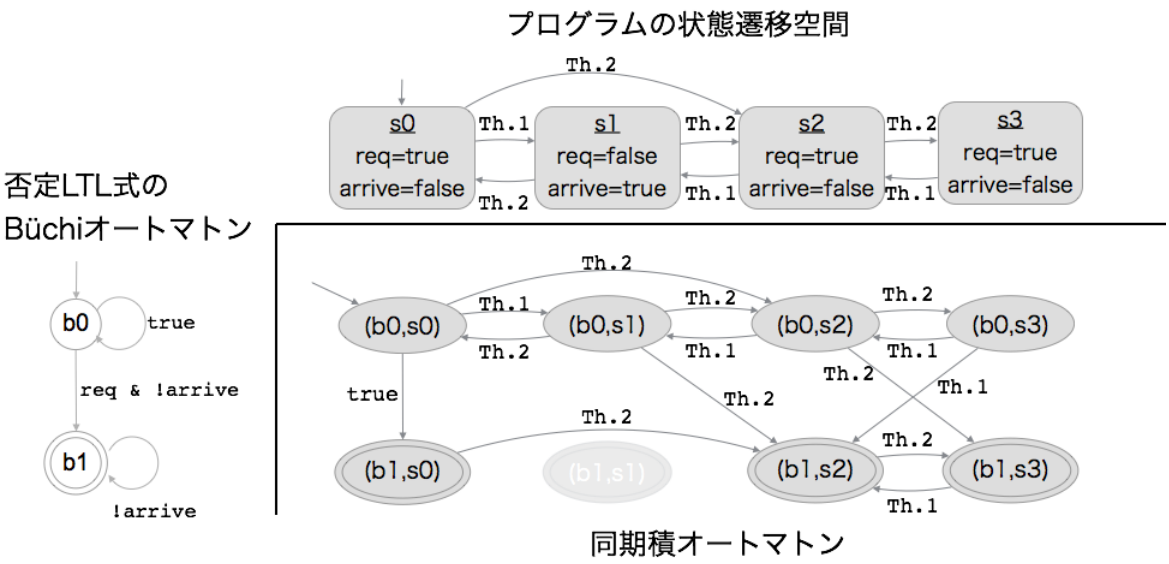

図 3.6: 同期積の例

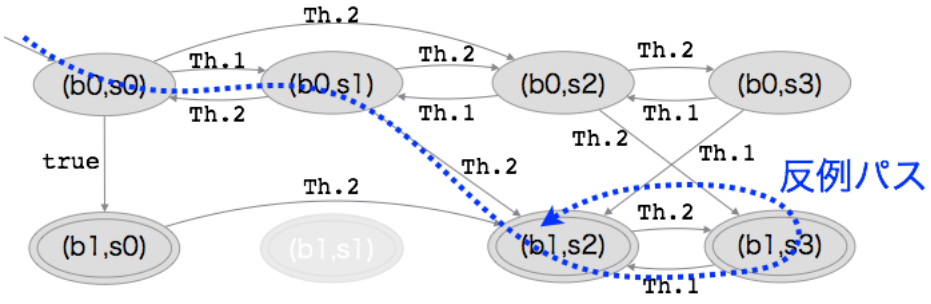

図 3.7: 反例パスの例

索（Red DFS と呼ぶ）を開始する (17 行目)．Red DFS はその受理状態から深さ優先探索を開始し，現在の探索パス上への遷移を探す (23 行目から 27 行目)．現在の探索パスとは，初期状態から Blue DFS の中断時点の状態への状態遷移パスである．Red DFS が現在の探索パスへの遷移を発見すると，これは，中断時の受理状態を含むループ遷移となるため，検証の反例として出力する (25 行目)．Red DFS が，バックループを発見することなく全ての探索を終えた場合は，中断時の受理状態を含むループは存在しないため，中断した状態からバックトラックし，Blue DFS を再開する (18 行目)．このようにして，反例を発見するか，Blue DFS が全ての状態を訪問し終えるまで探索を継続する．

上述のように Blue DFS では，各状態から先の探索を全て終えたのちに，受理状態からの Red DFS を開始する．この手順によって，Red DFS の探索済み状態を管理する VistedRed を，各受理状態から始まる Red DFS の間で共有することができ，二重の深さ優先探索であるにも関わらず，計算量を $O(n)$ (n は同期積オートマトンの状態数) に抑えることができることを特徴とする．

リスト 3.3: Nested Depth-First Search 疑似コード

```
1  nested_depth_first_search:
2    stack StackBlue = []
3    set VisitedBlue = {}
4    stack StackRed = []
5    set VisitedRed = {}
6    blue_dfs(S0)
7
8  sub blue_dfs(S):
9    if is_error_state(S)
10     report_error(S)
11   push(StackBlue, S)
12   add(VisitedBlue, S)
13   for each successor S' of S
14     if not exists(VisitedBlue, S')
15       blue_dfs(S')
16   if is_acceptant_state(S')
17       red_dfs(S)
18   pop(StackBlue)
19
20 sub red_dfs(S):
21   push(StackRed, S)
22   add(VisitedRed, S)
23   for each successor S' of S
24     if exists(StackBlue, S')
25       report_error_cycle(StackBlue+StackRed)
26     else if not exists(VisitedRed, S')
27       red_ndfs(S')
28   pop(StackRed)
```

第4章　深さ優先ヒューリスティック探索による効率化の提案

伝統的なヒューリスティック探索アルゴリズムは，3.2 節で述べたように，1) 状態間で優先度に差がつかない場合に，横に広く探索する傾向となり，空間の深いところに到達しづらい，2) 深さ優先探索ベースの LTL 検証アルゴリズムと相性が悪い，という課題がある．

そこで本研究では，これらの課題を解決するために，深さ優先ヒューリスティック探索 (Depth-First Heuristic Search, DFHS) を提案する．DFHS ではこれらを解決するために，深さ優先探索にヒューリスティック探索の概念を導入する．以下，本章では，DFHS を構成する三つの機能について説明する．

4.1　枝刈り機能

一つ目の機能は枝刈り機能である．DFHS は，状態の探索順序は，深さ優先探索を基本とする．これに対して，探索状態に関する情報から定義した「枝刈り関数」によって，不具合の含まれる見込みが少ないと判断した枝の探索を打ち切り，別の候補にバックトラックする．これによって，探索範囲を限定し，不具合発見の効率化を図る．

DFHS の探索の概念図を図 4.1 に示す．DFHS は深さ優先探索と同様の探索順序で探索を進める．その上で，各状態に遷移する度に，枝刈りポリシーに該当するかを判定するために枝刈り関数を呼び出す．条件に該当する場合，それ以降の探索を継続せず，バックトラックして別の探索パスに遷移する．図 4.1 では，3 番目の探索状態に遷移したときに，枝刈り関数によって，それ以降の探索を打ち切った例を示している．図 3.1 の深さ優先探索の例と比較すると，3 番目の状態で打ち切ることで，不具合検出が効率化されている．

リスト 4.1: DFHS のアルゴリズム (枝刈り機能)

```
1  depth_first_heuristic_search:
2    stack Stack = [S0]
3    set Visited = {S0}
4    while (S = top(Stack)) != null
5       S' = get_next_successor(S)
6       if S' == null
7           pop(Stack)
8       else if is_error_state(S')
9           report_error(S')
10          return
11      else if not exists(Visited, S')
12          add(Visited, S')
13          if size_of(Stack) > predefined_depth_limit
14             if not cutoff_function(S')
15                 push(Stack,S')
16          else
17             push(Stack,S')
```

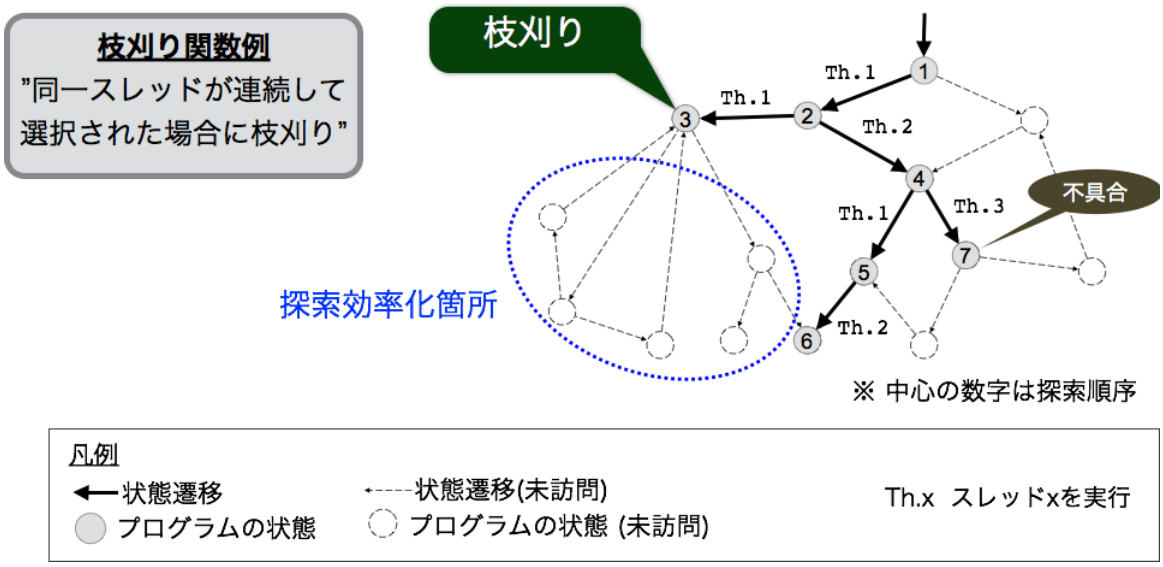

図 4.1: 深さ優先ヒューリスティック探索 (DFHS)

リスト 4.1 に DFHS のアルゴリズムの疑似コードを示す．14 行目で使用されている `cutoff_function()` は，状態遷移のたびに呼ばれる「枝刈り関数」を表す．true または false を返し，true は探索の打ち切りを，false は継続を意味する．これは，後述の通り，探索やモデルの状態に応じて任意のポリシーでに判定する．枝刈り判定は，状態遷移後，冒頭に判定するため，枝刈りと判定された場合は，その状態から遷移するすべての次状態遷移の探索を打ち切る．

なお，13 行目の判定文に示すように，探索打ち切りの判定は，一定以上の深さにおいてのみ適用する．`predifined_depth_limit` は，枝刈り判定を行う深さの下限を示す．探索初期の浅い段階で打ち切りを行うと，状態空間の大部分を削減することになり，不具合を発見することなく探索が終了する場合が多発するため，探索開始直後の過度な探索パスの削減を抑制するために実施する．

以下，DFHS をソフトウェアモデル検査に適用した場合の探索の特徴について説明する．ソフトウェアモデル検査に適用した場合は，状態はプログラムの実行スナップショットであり，探索の枝は，スレッド選択などのプログラムに起因する非決定要因の網羅探索に当たる．

もっとも重要な非決定要因であるスレッドの状態に着目して定義した枝刈り関数 `cutoff_function()` の例を以下に示す．これらは従来手法のスレッドに着目した優先度算出関数に対応する．従来の優先度算出関数が,「不具合の含まれそうな状態」を数値として推定するのに対して，枝刈り関数は,「不具合のなさそうな状態」を true/false として判定する点が異なる．なお，6.4 節で述べるように，これらは裏表の関係にあり，従来の優先度算出関数を用いて，枝刈り関数を機械的に定義するすることや，その逆も可能である．

Interleaving(n): 最新の状態遷移で実行されたスレッドが，実行可能スレッド数 – n 回の直近の状態遷移の中で 2 回選択されている場合にバックトラックする．特定のスレッドに実行が集中することを抑制することで，スレッド間で特定のインターリービングが発生した場合に起こるリソース競合やデッドロックの検出に有効であることが期待できる．図 4.2 に Interleaving ポリシーによる枝刈りの例を示す．

NonConsecutive(n): 直近 n 回の状態遷移において，全て同一スレッドが実行されている場合にバックトラックする．連続実行を防ぐことで，リソース競合の検出に有効と考える．図 4.3 に NonConsecutive ポリシー

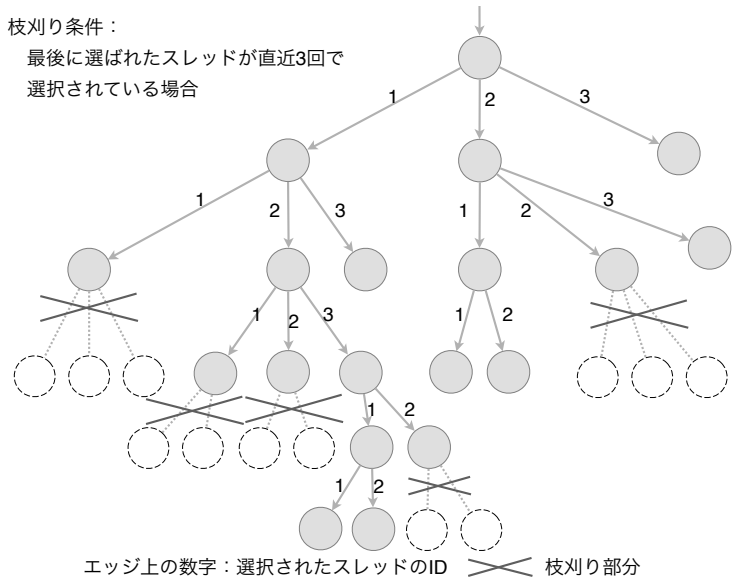

図 4.2: Interleaving ポリシーのイメージ

による枝刈りの例を示す．この例では，連続 2 回同一スレッドが選択された場合に探索を枝刈りする例を示す．

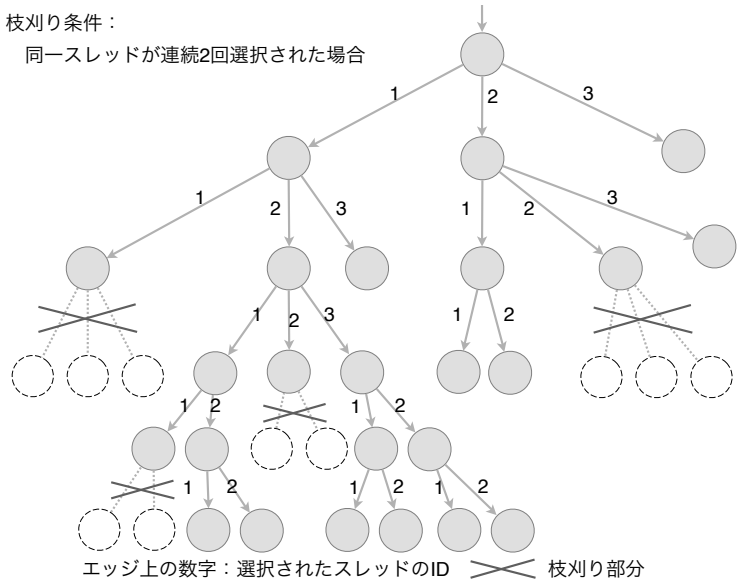

図 4.3: NonConsecutive ポリシーのイメージ

LessInterleaving(n, m): 直近 *m* 回の状態遷移の中で *n* 回以上選択されたスレッドが切り替わっている場合にバックトラックする．並行動作に起因する不具合が，たかだか数回のインターリーブで発生するような場合に，探索空間を大きく削減することが期待できる．図 4.4 に LessInterleaving ポリシーによる枝刈りの例を示す．

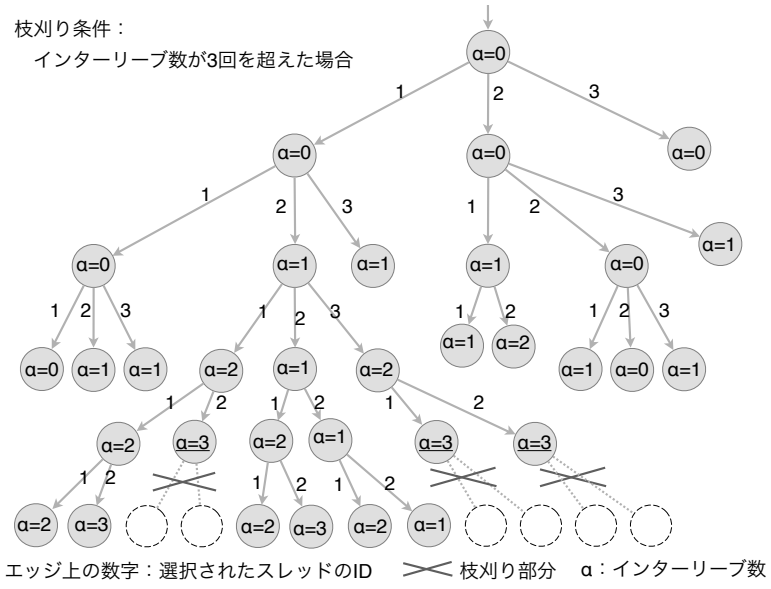

図 4.4: LessInterleaving ポリシーのイメージ

BlockedNum(n): 直近 n 回の状態遷移において，ブロックしているスレッド数が増加していない場合にバックトラックする．デッドロックの検出に有効と考える．図 4.5 に BlockedNum ポリシーによる枝刈りの例を示す．

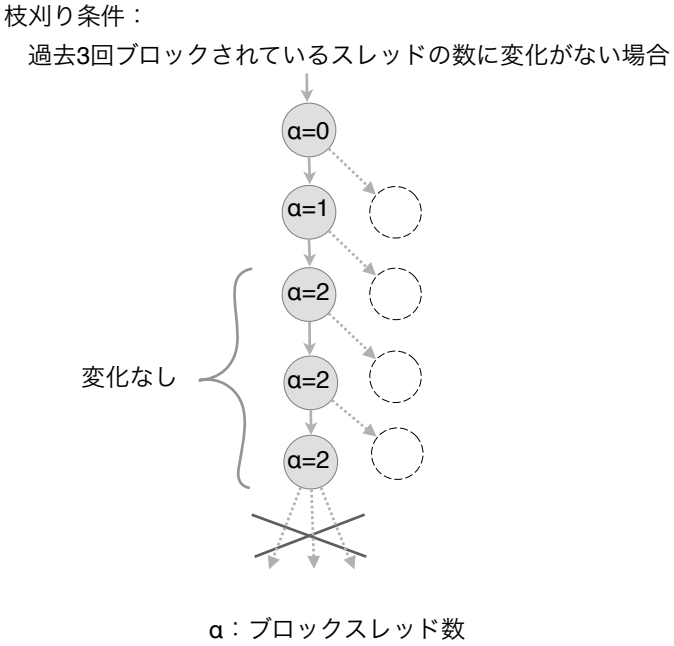

図 4.5: BlockedNum ポリシーのイメージ

Random(α): ランダムな確率 α でバックトラックする．なお，Parízek, Lhoták らの手法 [29] は，DFHS でこの Random ポリシーを適用した場合に相当する．プログラムや探索空間に依存せず広く有効なポリシーであると期待できる．図 4.6 に Random ポリシーによる枝刈りの例を示す．各状態において，一定の確

率でバックトラックするかどうかを決定する．

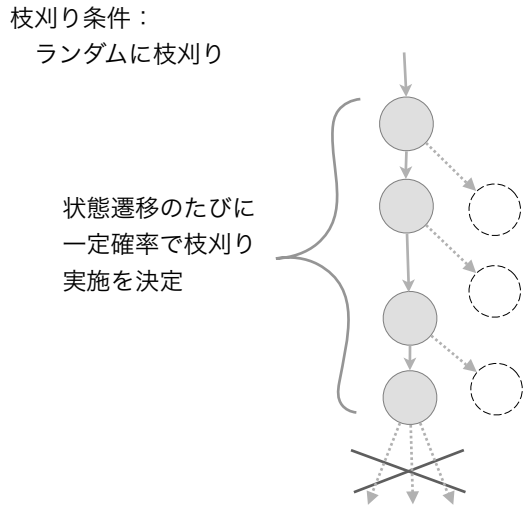

図 4.6: Radom ポリシーのイメージ

DFHS は，深さ優先を基本としているため，早期に探索空間の深いところに到達できる特徴を持つ．また，探索のバックトラックに備えてプログラムの状態を保持しておく必要があるが，この保存メモリを深さ方向に最大限利用できるため，メモリ効率よく広い空間を探索する上で有利である．すなわち，DFHS の特徴は，枝刈り関数によって推定した「不具合のなさそうな状態」以外の状態を，探索空間中の深い状態を含めて効率よく探索できる．

4.2　枝選択順序最適化機能

二つ目の機能は，枝選択順序の最適化である．DFHS は，深さ優先の順序で探索を実行する．これに加えて，ある状態から次状態に遷移する枝の選択順序を制御することで，最良優先探索の考え方を部分的に適用する．状態遷移時の子供の選択順序は，探索パラメータとして利用者が指定する．

リスト 4.2 に，枝選択順序最適化機能を含めた DFHS のアルゴリズムを疑似コードで示す．5 行目の `get_next_successor_in_order()` がリスト 4.1 と異なる．各状態から次状態に遷移する候補として，`get_next_successor_in_order()` 返される選択順序に従って探索を行う．

リスト 4.2: DFHS のアルゴリズム (枝刈り機能，枝選択順序最適化機能)

```
1  depth_first_heuristic_search:
2    stack Stack = [S0]
3    set Visited = {S0}
4    while (S = top(Stack)) != null
5       S' = get_next_successor_in_order(S)
6       if S' == null
7           pop(Stack)
8       else if is_error_state(S')
9           report_error(S')
10          return
11      else if not exists(Visited, S')
12          add(Visited, S')
```

```
13          if size_of(Stack) > predefined_depth_limit
14             if not cutoff_function(S')
15                 push(Stack,S')
16          else
17             push(Stack,S')
```

以下に，ソフトウェアモデル検査において枝選択順序最適化を適用する場合のポリシーの例を示す．図 4.7 に示すような，*n* 番目の状態で次に選択可能なスレッドが Thread1, Thread2, Thread3 の 3 つである場合における例を踏まえて説明する．

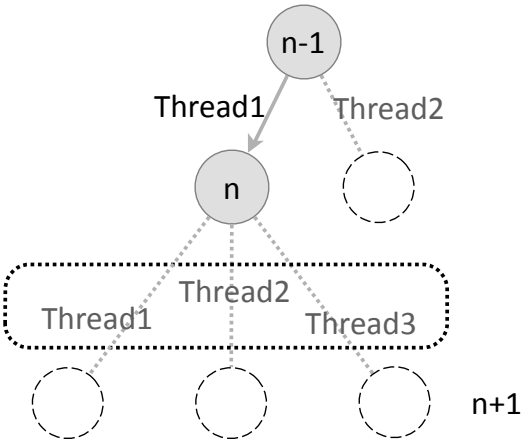

図 4.7: 枝選択順序

Interleaveing: 最後に実行したスレッドと異なるスレッドを先に選択する．図 4.7 の例では，Thread1 が選ばれた直後のため，Thread1 の選択を最後に探索する．

LessInterleaving: 最後に実行したスレッドを先に選択する．図 4.7 の例では，Thread1 を再び選択する．

Random: ランダムに選択する．図 4.7 の例では Thread1, Thread2, Thread3 のいずれかがランダムに選択される．

枝選択順序最適化と，第 4.1 章の枝刈り機能と合わせることで，ポリシーにあった状態を先に探索する部分的な優先探索を行いつつ，不具合を含む見込みの低い部分の枝刈りが実現でき，探索の効果を向上されることが期待できる．

4.3 公平性最適化機能

三つ目の機能は，LTL 検証における公平性検証を効率化するための機能である．本機能は，LTL 検証に対してのみ適用する．ソフトウェアモデル検査器は，あらゆるプログラム状態を再現し，検証しようとする．そのため，現実的なスレッドスケジューリングを想定した場合には起こり得ないような，無意味な反例を出力することが知られている．例えば，動作可能状態にあるスレッドが複数存在するにも関わらず，特定のスレッドのみが永遠に動作するようなといった反例が出力されることがある．しかし，実際のソフトウェア実行では，OS のスケジューリング等で，全てのスレッドに実行の機会が与えられることが一般的であり，本来探したい不具合ではない場合が多い．このようなスケジューリングをモデル検査の分野では，公

平性が満たされていないという．公平性が満たされない解を排除したい場合は，公平性条件の元で検証を実行する．公平性検証は，原理的には，LTL 式に公平性条件を表す論理式を追加することで実現できる．しかし，この方法では，LTL 式が複雑となることで検証そのものに時間がかかってしまうことになる．3.3 節で述べたように，LTL 検証の計算量は，同期積の状態数に依存するため，LTL 式が複雑になると同期積の状態数が増大してしまうためである．例えば，公平性条件のない下記の LTL 式の否定を Büchi オートマトンに変換すると，状態数は 2 である．

```
[](req -> <>arrive)
```

これに対して以下のような公平性条件をつけた LTL 式を変換した場合は，状態数は 9 となる．

```
([]<>th_enb->[]<>th_run)
        -> [](req -> <>arrive)
```

同期積の状態数は，オーダーとして LTL 式から変換した Büchi オートマトンの状態数とプログラムの状態遷移空間の状態数をかけたものとなるため，Büchi オートマトン側の状態数の増加は，大きな計算量増大につながる．

そこで DFHS では，LTL 式を変更することなく公平性条件のもとでの反例を出力する方法を実現する．リスト 4.3 に公平性最適化機能のアリゴリズムを示す．以下の手順を説明する．

- 反例パスを見つけた場合，反例に含まれるループ部分における各スレッドの状態を確認することで，公平性を満たすかどうかをチェックする．もし公平性を満たさない場合は，その反例を破棄し，他の反例の探索を継続する．公平性の充足確認は，ループ部分の各状態について，で存在する全てのスレッドについて，ループ部分の各状態で一度も実行可能状態になっていないか，あるいは一つ以上の状態で実行状態になっているか，をチェックすることで実施する (26 行目の `isFairLoop()` 関数).
- 公平性を満たす反例が見つかるか，全ての状態を探索し終えるまで探索を継続する．

以上の手順を表した概念図を図 4.8 に示す,

リスト 4.3: 公平性最適化機能のアルゴリズム

```
1  depth_first_heuristic_search_for_LTL:
2    stack StackBlue = []
3    set VisitedBlue = {}
4    stack StackRed = []
5    set VisitedRed = {}
6    blue_dfs(S0)
7
8  sub blue_dfs(S):
9    if is_error_state(S)
10     report_error(S)
11   push(StackBlue, S)
12   add(VisitedBlue, S)
13   if size_of(Stack) > predefined_depth_limit
14     if not cutoff_function (S')
15       for each successor S' of S
16         if not exists(VisitedBlue, S')
17           blue_dfs(S')
18         if is_acceptant_state(S')
19           red_ndfs(S)
20   pop(StackBlue)
21
22 sub red_ndfs(S):
23   push(StackRed, S)
24   add(VisitedRed, S)
25   for each successor S' of S
```

```
26      if exists(StackBlue, S') and isFairLoop(StackBlue+StackRed)
27          report_error_cycle(StackBlue+StackRed)
28      else if not exists(VisitedRed, S')
29          red_ndfs(S')
30  pop(StackRed)
```

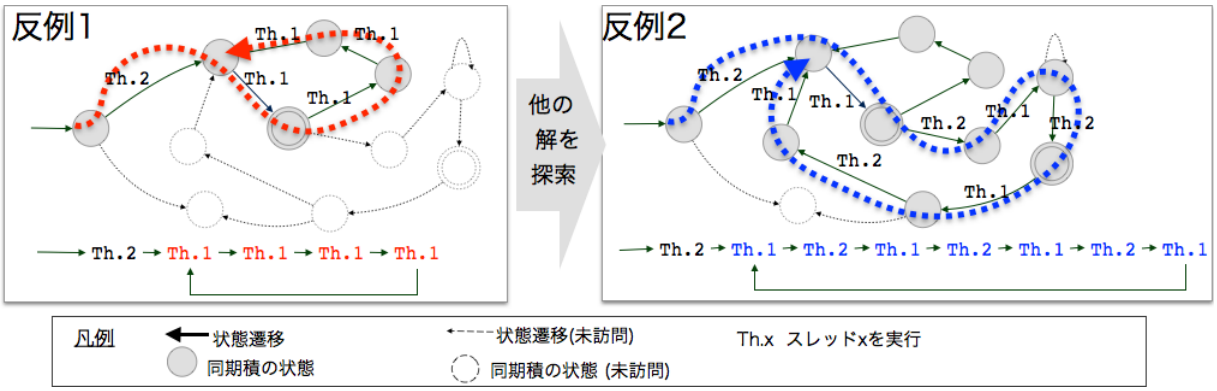

図 4.8: 公平性最適化機能に基づく反例の探索

第5章　提案手法の実装

本研究では，第 4 章で提案した DFHS をモデル検査器 JPF を用いて実装した．本章では，DFHS を構成する三つの機能の実装について説明する．

5.1　枝刈り機能

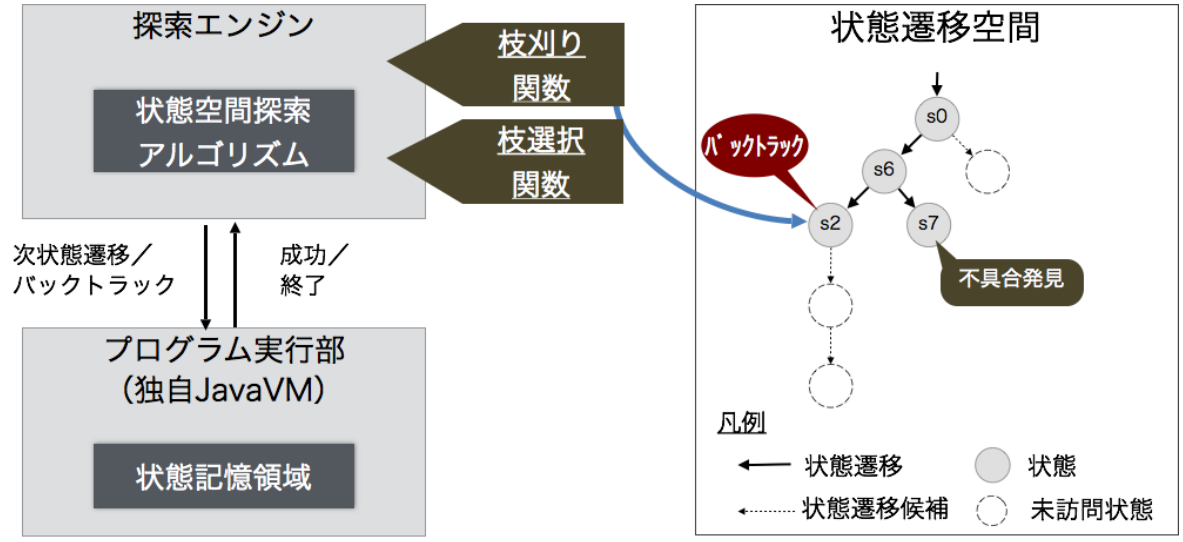

図 5.1: 探索エンジンの拡張

図 5.1 に提案手法を実現するための拡張を示す．JPF はプログラムを実行するプログラム実行部と，その実行を制御し，不具合の有無をチェックする探索エンジンからなる．既存の深さ優先探索エンジンに対して，新しい状態が作られたときに枝刈り関数を呼び出し，探索の継続・バックトラックを制御するように拡張を行った．

枝刈り関数は，リスト 5.1 で示した汎用リスナークラスを継承して作成する．各枝刈りポシリーの実装では，checkCutoff 関数 (45 行目) をオーバーライドし，この中で枝刈りするかどうかを判定し，枝刈りすると決定した場合は，true を返すようにする．checkCutoff は状態の遷移を完了する度に呼び出される．引数で渡される Search インスタンスは，プログラムの実行および探索に関する様々な情報を取得することができ，これらを枝刈りの判定に利用できる．

- 初期状態からの状態遷移，探索の深さ，探索パス上で選択されたスレッドなどの探索パスに関する情報
- スレッドの数，各スレッドの状態 (実行可能，ブロック等) といったスレッドに関する情報

28 行目に示すように，枝刈りの判定は，あらかじめ定数として設定した深さ以上の状態においてのみ適用

する．

リスト 5.1: 枝刈り関数実装用汎用クラス

```
/**
 *   Base class of cut-off functions
 */
public class DefaultCutOffPolicyListener extends ListenerAdapter {
   static private final JPFLogger log = JPF.getLogger(DefaultCutOffPolicyListener.class.getName());

   int cutoffCount = 0;
   final private int APPLY_CUTOFF_POLICY_DEEPER_THAN_THIS;

   protected Config config;

   public DefaultCutOffPolicyListener(Config config) {
      this.config = config;
      APPLY_CUTOFF_POLICY_DEEPER_THAN_THIS = config.getInt("search.cutoff.policy.depth", 5);
   }

   @Override
   public void stateAdvanced(Search search) {
      log.fine("stateAdvanced: ", search.getStateId());
      checkPolicy(search);
   }


   private void checkPolicy(Search search) {
      if (! search.supportsBacktrack()) {
         log.info("This search engine does not suppport requestBacktrack().");
         throw new RuntimeException("This search engine does not suppport requestBacktrack().");
      }

      if (search.getDepth() > APPLY_CUTOFF_POLICY_DEEPER_THAN_THIS)
      {
         if (checkCutoff(search)) {
            log.info("backtrack requested");
            search.requestBacktrack();
            cutoffCount++;
         }
      }
   }

   /**
    * a function to check a cut off policy
    */
   protected boolean checkCutoff(Search search) {
      return false;
   }

}
```

以下，4.1 節で説明した各枝刈り関数の記述例を示す．

Interleaving(n): リスト 5.2 に示す．最新の状態遷移で実行されたスレッドが，実行可能スレッド数 – n 回の直近の状態遷移の中で 2 回選択されている場合にバックトラックする．n は外部設定ファイルで設定する．

リスト 5.2: 枝刈り関数記述例（Interleaving）

```
/**
 *   Interleaving Policy
 */
   @Override
   public boolean checkCutoff(BASearch search) {

```

```
7         ThreadInfo[] threads = search.getVM().getLiveThreads();
8         int runnableCount = 0;
9         for (ThreadInfo th : threads) {
10           if (th.isRunnable()) {
11              runnableCount++;
12           }
13        }
14
15        int historyLimit = runnableCount - n;
16
17        Path path = search.getVM().getPath();
18        int pathSize = path.size();
19
20        if (pathSize > 5 && historyLimit > 0) {
21           if ((pathSize - historyLimit - 1) > 0) {
22              for (int i = pathSize - historyLimit - 1; i < pathSize - 1; i++) {
23                 if (path.get(i).getThreadIndex() == path.get(pathSize - 1).getThreadIndex()) {
24                    return true;
25                 }
26              }
27           }
28        }
29        return false;
30
31     }
```

NonConsecutive(n): リスト 5.3 に示す．同一のスレッドが n で指定された回数，連続で選択された場合にバックトラックを行う．n は外部設定ファイルで設定する．

リスト 5.3: 枝刈り関数記述例（NonConsecutive）

```
1  /**
2   * NonConsecutive Policy
3   */
4     @Override
5     public boolean checkCutoff(BASearch search) {
6        ThreadChoiceFromSet[] cgs = search.getVM().getChoiceGeneratorsOfType(ThreadChoiceFromSet.class);
7        ChoiceGenerator<?> cur = search.getVM().getChoiceGenerator();
8        if (cgs.length > APPLY_DEPTH && cur instanceof ThreadChoiceFromSet) {
9           ThreadChoiceFromSet tc = (ThreadChoiceFromSet) cur;
10          int id = tc.getThreadInfo().getId();
11          for (int i = 0; i < n; i++) {
12             tc = tc.getPreviousChoiceGeneratorOfType(ThreadChoiceFromSet.class);
13             if (tc == null || tc.getThreadInfo().getId() != id) {
14                return false;
15             }
16          }
17          return true;
18       }
19       return false;
20    }
```

LessInterleaving(n, m): リスト 5.4 に示す．直近 m 回の状態遷移中で，インターフィーブの回数が上限値 n を超えた場合に，バックトラックを行う．n, m は外部設定ファイルで設定する．

リスト 5.4: 枝刈り関数記述例（LessInterleaving）

```
1  /**
2   *   LessInterleaving Policy
3   */
4     @Override
5     public boolean checkCutoff(BASearch search) {
6        ThreadChoiceFromSet[] cs = search.getVM().getChoiceGeneratorsOfType(ThreadChoiceFromSet.class);
7        if (cs.length > APPLY_DEPTH) {
8           int interleaveNum = 0;
```

```
9           int cur = cs[cs.length - 1].getThreadInfo().getId();
10          int limit = cs.length > m ? cs.length - m : 0;
11          for (int i = cs.length - 1; i > limit; i--) {
12             int prev = cs[i - 1].getThreadInfo().getId();
13             assert cur == cs[i].getThreadInfo().getId() : "thread interleave check error";
14             if (prev != cur) {
15                interleaveNum++;
16             }
17             cur = prev;
18          }
19          if (interleaveNum > n) {
20             log.fine("intnum: " + interleaveNum + " depth: " + cs.length);
21             return true;
22          }
23          log.fine("intnum: " + interleaveNum + " depth: " + cs.length);
24       }
25       return false;
26    }
```

BlockedNum(n): リスト 5.5 に示す．*n* 回の状態遷移において，ブロックされたスレッド数の増加がない場合にバックトラックを行う．*n* は外部設定ファイルで設定する．

リスト 5.5: 枝刈り関数記述例（BlockedNum）

```
1  /**
2   *   BlockedNum Policy
3   */
4     @Override
5     public boolean checkCutoff(Search search) {
6        int size = blockedNum.size();
7
8        blockedNum.get(size - 1);
9        int b = blockedNum.get(size - 1);
10       boolean cutoff = true;
11       for (int i = 2; i <= n; i++) {
12          if (size - i < 0) {
13             cutoff = false;
14             break;
15          }
16          int bprev = blockedNum.get(size - i);
17          if (bprev < b) {
18             cutoff = false;
19             break;
20          }
21       }
22
23       return cutoff;
24    }
```

Random(randomRatio): リスト 5.6 に示す．確率 *randomRatio* に従って，ランダムにバックトラックを行う．*randomRatio* は外部設定ファイルで設定する．

リスト 5.6: 枝刈り関数記述例（Random）

```
1  /*
2   *   Random Policy
3   */
4     @Override
5     public boolean checkCutoff(BASearch search) {
6        if (random.nextDouble() < randomRatio) {
7           return true;
8        }
9        return false;
10    }
```

5.2　枝選択順序最適化機能

JPF には，ある状態から遷移可能な枝の管理を行う仕組みがあり（Choise Generator と呼ばれる），非決定要素の候補管理を行う．スレッドに関する Choise Generator は，次に選択可能なスレッド候補を管理し，バックトラックのたびに次候補を選択・実行する．このとき，選択されるスレッド候補の順序は，標準ではスレッドの ID の順で出力されるようになっている．また，JPF の設定でランダムな順序とすることも可能である．これに加えて，任意の試行順序を定義できるように，順序をカスタマイズできる機能拡張を行った．選択可能なポリシーは以下のとおりである．

Interleaving 直近の状態遷移で選択されたスレッドがスレッドの選択候補の中に存在する場合は，そのスレッドを選択候補の最後に回す．それ以外のスレッドについてはスレッド ID の順とする．インターリーブが発生するものを優先するポリシーであり，枝刈り関数の Interleaving とのセットでの活用を想定する．

LessInterleaving 直近の状態遷移で選択されたスレッドを，選択候補の一番に選択する．それ以外のスレッドについてはスレッド ID の順とする．インターリーブが発生しないものを優先するポリシーであり，枝刈り関数の LessInterleaving とのセットでの活用を想定する．

なお，Choice Generator による選択順序は，スレッドを選択実行する前にリストとして準備する．BlockdNum の枝刈り関数については，ブロックされるスレッドが増えるような枝を先に選択する枝選択ポリシーが望ましいが，ブロックされるかどうかは，実際にスレッドを実行してみるまでわからないため，上記のような望ましいポリシーを実現することは困難であるため提供していない．

リスト 5.7 に，Interleaving ポリシーに従って，枝の選択順序をカスタマイズする処理の記述例を示す．filter 関数は，探索エンジンから新しい状態が作成されるたびに呼び出される．このとき，引数といて，次に選択可能なスレッド一覧が配列として渡される．filter 関数の戻り値で返すスレッド一覧が実際に探索エンジンでスレッドが選択される順序となるため，枝選択順序最適化機能の実装では，filter 関数に与えられたスレッド一覧の配列の中身の順序を入れ替えて戻り値とすることで，機能を実現した．探索エンジンから渡されるスレッド一覧は，スレッド ID の順で格納されている．本実装では，配列の中から最後に選ばれたスレッドを探し，27 行目に示すようにリストの一番最後に配置しなおすことで，Interleaving ポリシーを実現している．

リスト 5.7: 枝選択順序最適化の記述例 (Interleaving)

```
1  public class InterleaveBASchedulerFactory extends DefaultBASchedulerFactory {
2
3      public InterleaveBASchedulerFactory(Config config, VM vm, SystemState ss) {
4          super(config, vm, ss);
5      }
6
7      @Override
8      protected ThreadInfo[] filter(ThreadInfo[] list) {
9          assert list != null;
10
11         int len = list.length;
12
13         if (len < 2) {
14             return list;
15         }
16
17         ThreadInfo currentThread = ThreadInfo.getCurrentThread();
18
```

```
19          int i = 0;
20          for (ThreadInfo ti : list) {
21              if (ti.equals(currentThread)) {
22                  break;
23              }
24              i++;
25          }
26          if (i < len - 1) {
27              System.arraycopy(list, i + 1, list, i, len - (i + 1));
28              list[len - 1] = currentThread;
29          }
30          return list;
31      }
32  }
```

5.3 公平性最適化機能

本節では公平性最適化機能の実装について説明する．まず，ベースとなる NDFS の実装を説明したのち，公平性最適化機能の実装について述べる．

5.3.1 NDFS の実装

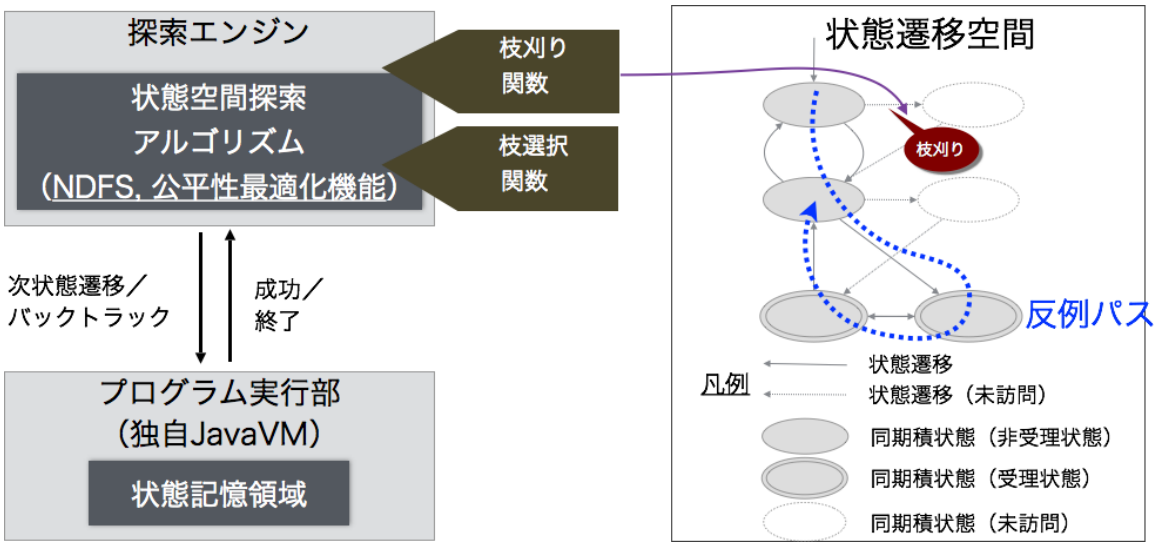

図 5.2: LTL 検証のための探索エンジンの拡張

JPF は，標準では LTL 検証をサポートしていない．LTL 検証のための取り組みがいくつか存在するが，いずれも研究的な位置づけのものである [22, 10]．そこで本研究では，LTL 検証を実現するために，3.3 節で説明した NDFS [7] を実装した探索エンジンを開発した (図 5.2)．

検証性質を表すために，原子命題，LTL 式を定義する方法および，プログラムのロジックに合わせて原子命題の真偽値を変更する方法を提供する．検証したい性質を表す LTL 式は JPF の設定ファイルで行う．リスト 5.8 に LTL 定義の記述例を示す．

リスト 5.8: LTL 式の設定例

```
ltl.formula =
    ([]<>th_enb1->[]<>th_run1) ->
    ([]<>th_enb2->[]<>th_run2) -> [](req -> <> arrive)
```

この中の th_enb*i*, th_run*i*, req, arrive は原子命題を表す．設定ファイル中でこのように定義することで，探索エンジンの内部で変数として定義される．原子命題は，ある状態において true または false が確定する論理値であり，プログラムの実行に伴って変化する．原子命題の値の変更は，プログラムのロジックに依存するため利用者自身が記述する．記述方法は，プログラム本体に追記する方法と，探索エンジンに設定するリスナー中で記述する方法があり，これを行うための *BAVerify* というクラスを提供している．原子命題は，通常の原子命題とスレッドの動作状況を表す原子命題の二種類存在する．まず，通常の原子命題の使用方法を以下に示す．設定ファイルで使用した原子命題は内部で変数化されるが，この値を変更するためには，本例で示すように，ソフトウエアのロジックに合わせて，*BAVerify.abValueVar* メソッドを用いる．第一引数の文字列で指定した原子命題の真偽値を第二引数で指定した真偽値に設定する．

```
public void elevatorRequested() {
    // 原子命題の値を変更
    BAVerify.apValueVar("req", true);
    ...
}
...
public void setFloorNum(int num) {
    ...
    // 原子命題の値を変更
    BAVerify.apValueVar("wrongFloor", num > MAX_FLOOR_NUM);
    ...
}
```

これに対して，スレッドの動作状態を表す特別な原子命題は，*BAVerify.declareThreadAP* メソッドを用いてスレッド実行の先頭で原子命題の名前を定義するだけでよい．以下に定義例を示す．

```
public void run() {
    // これを実行するスレッドに対して，動作可能状態，動作状態を表す二つの原子命題を定義する
    BAVerify.declareThreadAP("th_enb1", "th_run1");
    ...
}
```

本例で示すように，スレッド開始時に，原子命題の名前を指定するだけで．その変数は内部で管理され，探索エンジンが自動的に論理値を変更する．上記の例では，このスレッドが動作可能状態である場合に th_enb1 が真，動作不可能な状態である場合に th_enb1 が偽に変更される．また，th_run1 については，このスレッドが選択されて実行された状態遷移後の状態においてのみ真となる．このスレッドの実行状態を表す原子命題は，公平性検証を行いたい場合に LTL 式中で利用する．

探索エンジンの内部では，まず LTL 式の中で使われている原子命題を変数化するとともに，LTL 式の否定を表す Büchi オートマトンを構築する．そして，プログラム実行しながら，このオートマトンとプログラムの状態遷移空間との同期積を段階的に構築し，3.3 節で説明した NDFS アルゴリズムに基づき，反例ループを探索する．

原子命題の値はプログラムの実行状態の一部であると考えられる．値の変更は LTL 検証結果に影響があるため，2.2.3 節で説明したトランジションを切る命令に原子命題変更の命令も加える．

5.3.2 公平性最適化機能の実装

前項で実装した NDFS の実装をベースとして，公平性最適化機能を実装した．公平性最適化は，発見した反例パス上のループ部分の各状態について，スレッドの動作状況をチェックし，公平性を満たすかどうかをチェックする機能である．したがって判定のためには，反例ループ中の各状態のときのスレッドの動作可能状況，動作状況が必要となる．そこで反例パスを発見した場合は，ループ部分の各状態に保持されている原子命題の値のうち，前項で説明したスレッド動作状況を表すものを含む原子命題の値を取得し，これを確認することで公平性を満たすかどうかをチェックする．

以下のような．LTL 式を元に，公平性最適化機能の実装について説明する．この LTL 式は，公平性最適化機能を使わずに，LTL 式として公平性条件を付与した場合の LTL 式である．本例では，プログラム中の二つのスレッドの先頭部分で前項で述べた方法に従い，th_enb1, th_enb2, th_run1, th_run2 を定義していることを想定する．th_enb1, th_enb2 の原子命題で，それぞれのスレッドの動作可能状態を表す．また，th_run1, th_run2 で動作状態（実際に動作した直後に true）を表す．

リスト 5.9: 公平性条件付き LTL 式の例

```
ltl.formula =
    ([]<>th_enb1->[]<>th_run1) ->
    ([]<>th_enb2->[]<>th_run2) -> [](req -> <> arrive)
```

このようなケースで公平性最適化機能を有効にして実行する場合は，以下の LTL 式のように，公平性条件は付与しない状態で実行する．ただしプログラム中の各スレッドの先頭で，th_enb1, th_enb2, th_run1, th_run2 を定義する部分は残しておく．

リスト 5.10: 公平性条件なし LTL 式の例

```
ltl.formula =
    [](req -> <> arrive)
fairnesopt=true
```

探索エンジンは，反例パスを発見すると，その中のループ部分の各状態について，順番に th_enb1, th_enb2, th_run1, th_run2 の値をチェックする．th_enb*i* がすべて false であるか，あるいは，th_run*i* が少なくとも 1 つの状態で true であるかを満たす場合は，公平性を満たす反例であるとして反例パスを出力する．満たさない場合は，探索を継続し，別の反例を探す．

第6章　提案手法の評価

本章では，従来手法の深さ優先探索および最良優先探索との比較実験によって DFHS の有効性を評価する．評価は二つの観点で実施する．まず，提案手法のアルゴリズムとしての性能評価を行う．加えて，実運用ユースケースを踏まえた評価を行う．ヒューリスティック探索においては，アプリケーションの不具合に合わせた適切なヒューリスティックを選択することが重要であるが，これをツールを適用する前に特定することが困難である．したがって，様々な観点のヒューリスティックを試行錯誤しながら適用することになる．さらには複数のポリシーを並行して実行できるリソースがある場合は，どれかが早期に発見できればよい．このような並行実行での評価を実運用での評価として述べる．

6.1　ベンチマークプログラム

評価には，並列プログラムの検証ツール評価用に設計された Java 用のベンチマークプログラム [32] を用いた．本プログラムは，マルチスレッド処理に起因する典型的な不具合を含んでおり，スレッド数などのパラメータ設定によって，検証のスケーラビリティを評価できるようになっている．使用したプログラムの一覧を表 6.1 に示す．検出対象の不具合は，デッドロック，未捕捉例外，ASSERT 違反である[1]．

6.2　安全性検証実験

本節では，安全性の検証に関する評価実験について説明する．従来手法である深さ優先探索，最良優先探索と DFHS を比較する．評価は，探索効率とメモリ効率の観点から実施する．

6.2.1　実験環境

最良優先探索は優先度算出関数の選択によって不具合発見効率が異なる．いくつかの観点で実験を実施するため，3.2 節で取り上げた Interleaving($n = 1,024$), MostBlocked, Random の各優先度算出関数を用いた．探索候補の状態を保持する優先度付きキューのサイズ制限は JPF 実装の既定値である 1,024 状態とした[2]．

これらに対して，DFHS については，枝刈り機能および枝選択順序最適化機能を適用し，4.1 節で取り上げた Interleaving($n = 2$), NonConsecutive($n = 3$), LessInterleaving($n = 10$, $m = \infty$), BlockedNum($n = 3$), Random($\alpha = 0.8$) を用いた．枝刈り判定の対象は深さ 5 以上の状態とした[3]．また，タイムアウト 15 分，メ

[1]今回使用したプログラムでは，データ競合はプログラム中の ASSERT でチェックされている．
[2]最良優先探索の各優先度算出関数のパラメータやキューのサイズについては，幾つか実験を行った結果，比較的良い結果の値であるものを採用した
[3]提案手法のパラメータについては，幾つか実験した結果，あまりにも結果が悪いものを除いた上で，恣意的な実験となることを防ぐため，ランダムに決定した．

表 6.1: ベンチマークプログラム一覧

プログラム (パラメータ)	内容	不具合の種類	ソース行数
Account (n)	口座間で特定の入出金処理を実施．最終的な合計値が正しい値になっているかを assert でチェックするプログラム．	デッド ロック, データ競合	147
Account Subtype (n, m)	正常な口座処理と異常な口座処理が並行動作し，口座間の送金処理を行うプログラム．	データ競合	190
Airline (n, m)	予約席を複数の客が確保しようとするプログラム．	データ競合	97
AlarmClock	アラームを管理するスレッドとアラーム通知を受ける複数のクライアントが並行処理するプログラム．パラメータはなし．	未捕捉例外	339
AllocateVector (n, m)	各スレッドが Vector に対して同時に処理を実行するプログラム．	未捕捉例外	285
Apps (n)	計算を行うタスクが結果を受け渡しするプログラム．	デッドロック	183
BoundedBuffer (n, m)	Producer スレッドと Consumer スレッドが，サイズ固定のバッファに対して生成，取得を繰り返すプログラム．	デッドロック	140
Clean (n, m)	シグナル二つに対して，シグナル送信，シグナル受信を，指定回数繰り返し実行するプログラム．	デッドロック	112
Daisy	共有リソースクラスに対して，各スレッドが read,write を行なう．プログラム各所で ASSERT チェックを実施．	ASSERT 違反	242
Deadlock1	それぞれ二つのスレッドが，リソース取得処理を一回実施して終了するプログラム．リソース取得順序が異なるためのデッドロック．	デッドロック	41
Deadlock2	それぞれ二つのスレッドが，リソース取得処理を一回実施して終了．リソース取得順序が異なるためのデッドロック．	デッドロック	47
DEOS	OS (DEOS) のスケジューリングを模擬するプログラム．抽象化を実施しているので，空間サイズは小さい．	ASSERT 違反	2,297
DiningPhil Type1 (n)	円卓にならんだ複数の哲学者が左右のナイフとフォークを確保するプログラム．Type1 は．各哲学者がフォークの確保，食事，フォークの開放を一度だけ実行する．	デッドロック	69
DiningPhil Type2 (n)	上記と同内容で，Type2 は，各哲学者がフォークの確保，食事，フォークの開放を無限回繰り返し実行する．	デッドロック	64
Piper (n)	複数の顧客スレッドが有限の座席を予約するプログラム．	デッドロック	131
LinkedList (n)	複数のスレッドから，リンクリストの add, remove 処理を実施するプログラム．	ASSERT 違反	303
LoseNotify(n, m, l)	複数スレッド間で wait, notify を実行しあうプログラム．	デッドロック	108
NestedMonitor	Producer, Consumer 二つのスレッドがリソースアクセスを実施するプログラム．	デッドロック	125
ProCon (n, m)	バッファに対して，Producer, Consumer がアクセスするプログラム．Main スレッドの最後で競合発生有無をチェックする．	リソース競合	196
Raxextended (n)	複数スレッドでイベントを送受信するプログラム．	ASSERT 違反	228
RaxEnvFirst (n)	複数スレッドでイベントを送受信するプログラム．	ASSERT 違反	228
Reorder (n, m)	共有オブジェクトに対して更新処理と参照処理が並行処理するプログラム．	データ競合	101
ReadersWriters (n, m)	共有オブジェクトに対して，read, write を行うプログラム．	ASSERT 違反	195
SleepingBarber	Barber スレッドと Customer スレッド間の競合処理が発生するプログラム．	デッドロック	155
TwoStage (n, m)	Set スレッド，Check スレッドによるリソース処理を実施．	ASSERT 違反	137
Wronglock (n, m)	共有リソースに対して，同期処理を実施するプログラム．	ASSERT 違反	111

モリ制限 2,048MB とし，制限を超えた場合は不具合検出失敗とする．

探索効率の評価には，不具合発見までに探索した状態数を用いる．同一の状態を複数回探索した場合は，その回数だけ重複してカウントする（探索済みの状態であっても，その状態に遷移するための実行コストは変わらないため）．この値が小さいほど，効率的に不具合を発見できたことを示す．評価に実行時間ではなく状態数を用いるのは，プログラムの実行時間は状況（実行環境，CPU アイドル状況等）によって不確定であるのに対して，状態数は実行状態に依存せず確定的なためである．ただし，以下の実験結果では，実行時間についても参考情報として記している．

6.2.2　実験結果

表 6.2 に既存手法である深さ優先探索，最良優先探索の実験結果を，表 6.3 に DFHS による実験結果を示す[4]．不具合を発見できた場合には，上段に探索した状態数を記し，下段には，参考までに探索に要した時間を「(分:秒)」の形式で付記した．不具合を発見できずに探索が終了した場合は「–」，利用可能なメモリを使い切った場合には「OOM」，探索時間が 15 分を超えた場合には「TO」と記述している．なお，プログラム欄のプログラム名の後ろの数値は，プログラムに対するパラメータである．表中の太字表示のものは，それぞれの手法の中で最適であった結果を表し，さらに下線表示のものは，手法を問わずそのプログラムで最も優れた結果を表す．

アルゴリズム性能評価

ここでは，実験結果を用いて DFHS のアルゴリズムとしての性能を評価する．最良優先探索と DFHS について，対応する優先度算出関数，枝刈り関数同士で比較し，性能が良かったプログラム数を比較する（表 6.2 の最良優先探索のそれぞれの列について，同一観点の，表 6.3 の DFHS の列を比較し，全 25 プログラム中で優位となったプログラム数を整理）．最良優先探索の Interleaving ポリシーに対しては，同じくインターリーブさせることを意図した DFHS の Interleaving および NonConsecutive を比較した．また，MostBlocked に対して BlockedNum，Random に対して Random を比較した．DFHS の Interleaving, NonConsecutive, Random については，枝優先最適化のあり，なしの両ケースを比較した．表 6.4 に比較結果を示す．この結果が示すように，MostBlocked 対 BlockedNum 以外の比較においては，DFHS が優位となるプログラム数が上回った．

最良優先探索の優先度算出関数が Interleaving, MostBlocked, Random の 3 ポリシーの適用であるため，これらに対して同観点のポリシーである DFHS の Interleaving, BlockedNum, Random をそれぞれ比較した結果を，図 6.1，図 6.2，図 6.3 に示す．グラフの縦軸は，最良優先探索の状態数を DFHS による状態数で割ったものを対数表現したもので，1 が最良優先探索と同等，1 より大き場合は最良優先探索より性能が良い，1 より小さい場合は，性能が悪いことを意味する．また，グラフが上限まで到達しているものについては，最良優先探索では，タイムアウトまたはメモリオーバーフローで不具合を発見できなかったケース，逆にグラフが加減まで達している場合は，DFHS が発見できなかったケースである．

Interleaving については，16 プログラムについて DFHS が優位，Random については，14 プログラムにつ

[4]表サイズの都合により，Interleaving, NonConsecutive, LessInterleaving をそれぞれ，Int, NonCon, LessInt と表記している．

表 6.2: 安全性検証結果 (既存手法)

プログラム	深さ優先探索	最良優先探索		
		Interleaving	MostBlocked	Random
Account	**636** (00:01)	603,334 (01:14)	25,282 (00:05)	5,056 (00:02)
AccountSubtype	TO (15:00)	TO (15:01)	**966,371** (02:33)	5,250,146 (14:55)
Airline 10 3	TO (15:00)	**287,121** (00:29)	**287,121** (00:30)	– (01:14)
AlarmClock	**544** (00:01)	55,410 (00:10)	2,558 (00:02)	8,587 (00:03)
AllocateVector	TO (15:00)	194,800 (00:27)	1,024,422 (02:05)	**4,098** (00:03)
Apps	2,598,794 (06:24)	– (00:24)	**84,780** (00:19)	– (02:00)
BoundedBuffer	**2,658** (00:02)	430,838 (01:02)	176,495 (00:25)	TO (15:00)
Clean 3 3 3	**98** (00:01)	347,208 (00:48)	532,789 (01:47)	1,746,836 (04:42)
Daisy	99,915 (00:21)	6,747 (00:04)	**2,799** (00:03)	23,048 (00:10)
Deadlock1	78 (00:00)	198 (00:00)	206 (00:00)	**67** (00:00)
Deadlock2	**12** (00:00)	35 (00:00)	35 (00:00)	39 (00:00)
DEOS	87,213 (05:56)	34,775 (01:57)	34,775 (01:55)	**10,513** (01:51)
DiningPhil 10	282 (00:02)	195,361 (06:45)	**194** (00:00)	2,656 (00:02)
Piper	**25,464** (00:06)	TO (15:03)	TO (15:00)	2,804,790 (10:02)
LinkedList	**3,146** (00:03)	107,000 (00:22)	TO (15:03)	25,159 (00:08)
LoseNotify 3 3 3	**14** (00:00)	11,792 (00:05)	437 (00:01)	61 (00:01)
NestedMonitor	**5** (00:00)	19 (00:00)	11 (00:00)	20 (00:00)
ProCon	**13,530** (00:04)	763,092 (01:51)	3,177,820 (06:21)	1,624,666 (03:35)
Raxextended	TO (15:02)	78,577 (00:14)	94,399 (00:19)	**1,288** (00:01)
RaxEnvFirst	OOM (18:27)	TO (15:09)	TO (15:03)	**341,431** (01:08)
Reorder 5 2	297,683 (00:33)	21,487 (00:06)	12,434 (00:04)	**2,891** (00:02)
ReadersWriters	2,535,301 (04:38)	49,309 (00:12)	17,925 (00:08)	**3,341** (00:03)
SleepingBarber	**17** (00:00)	549 (00:01)	108 (00:00)	201 (00:00)
TwoStage 5 5	TO (15:00)	– (00:58)	– (03:17)	**90,321** (00:15)
Wronglock 10 1	17,480 (00:06)	202,439 (00:31)	**459** (00:01)	2,612 (00:02)

表 6.3: 安全性検証結果 (DFHS)

プログラム	表 6.2 のベスト	DFHS									
		(2)	(1)					(1)+(2)			
		DFS/ Int	Int	NonCon	LessInt	Blocked	Random	Int/ Int	NonCon/ Int	LessInt/ LessInt	Random/ Random
Account	636 (00:01)	609 (00:01)	343 (00:01)	386 (00:01)	636 (00:01)	20,652 (00:06)	561 (00:01)	312 (00:01)	**234** (00:01)	673 (00:01)	529 (00:01)
Account Subtype	966,371 (02:33)	TO (15:00)	**8,036** (00:03)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)
Airline 10 3	287,121 (00:29)	TO (15:00)	1,921,393 (02:42)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	**632** (00:01)	TO (15:00)	5,181 (00:03)	TO (15:00)
Alarm Clock	544 (00:01)	328 (00:01)	60 (00:00)	1,140 (00:01)	544 (00:01)	544 (00:01)	– (00:01)	208 (00:01)	**88** (00:01)	612 (00:01)	– (00:02)
Allocate Vector	4,098 (00:03)	**35** (00:00)	TO (15:00)	55 (00:01)	TO (15:00)	TO (15:00)	220 (00:01)	**35** (00:01)	**35** (00:01)	TO (15:00)	39,888 (00:09)
Apps	84,780 (00:19)	TO (15:00)	**499** (00:01)	3,512,404 (07:59)	– (02:44)	TO (15:00)	1,224,545 (02:55)	3,088,261 (07:35)	TO (15:00)	– (01:44)	125,107 (00:21)
Bounded Buffer	2,658 (00:02)	3,885 (00:02)	TO (15:08)	30,149 (00:17)	473,986 (01:02)	674,175 (01:26)	**182** (00:01)	TO (15:10)	3,885 (00:04)	22,454 (00:08)	501 (00:01)
Clean 3 3 3	**98** (00:01)	131 (00:00)	1,754 (00:02)	234 (00:01)	TO (15:00)	**98** (00:00)	270 (00:00)	390 (00:01)	131 (00:01)	TO (15:00)	26,084 (00:05)
Daisy	**2,799** (00:03)	98,125 (00:21)	99,915 (00:22)	– (00:02)	99,915 (00:25)	4,778 (00:04)	– (00:01)	98,125 (00:20)	– (00:02)	99,434 (00:24)	– (00:01)
Deadlock -1	67 (00:00)	54 (00:00)	78 (00:00)	75 (00:00)	78 (00:00)	78 (00:00)	– (00:00)	54 (00:00)	52 (00:00)	100 (00:00)	**24** (00:00)
Deadlock -2	12 (00:00)	11 (00:00)	12 (00:00)	12 (00:00)	12 (00:00)	12 (00:00)	12 (00:00)	11 (00:00)	11 (00:00)	23 (00:00)	**9** (00:00)
DEOS	**10,513** (01:51)	87,213 (05:55)	87,213 (06:24)	87,213 (06:24)	87,213 (06:31)	87,213 (06:16)	– (00:09)	87,213 (06:23)	87,213 (06:24)	87,213 (06:13)	– (00:10)
Dining Phil 10	194 (00:00)	348 (00:01)	294 (00:01)	332 (00:02)	TO (15:00)	262 (00:01)	268 (00:01)	483 (00:01)	348 (00:01)	TO (15:00)	**118** (00:00)
Piper	25,464 (00:06)	25,186 (00:06)	TO (15:00)	26,969 (00:07)	TO (15:00)	TO (15:00)	4,253 (00:02)	103,642 (00:16)	25,186 (00:07)	TO (15:00)	**1,446** (00:01)
Linked List	3,146 (00:03)	784 (00:01)	3,146 (00:02)	521 (00:01)	3,146 (00:03)	**404** (00:01)	– (00:01)	784 (00:01)	20,899 (00:05)	3,278 (00:02)	– (00:00)
LoseNotify 3 3 3	**14** (00:00)	19 (00:00)	2,471 (00:02)	26 (00:00)	**14** (00:00)	**14** (00:00)	30 (00:00)	27 (00:00)	19 (00:00)	**14** (00:00)	35 (00:00)
Nested Monitor	**5** (00:00)	13 (00:00)	**5** (00:00)	**5** (00:00)	**5** (00:00)	**5** (00:00)	**5** (00:00)	13 (00:00)	13 (00:00)	**5** (00:00)	11 (00:00)
ProCon	13,530 (00:04)	**184** (00:00)	9,699 (00:03)	15,736 (00:05)	TO (15:00)	TO (15:00)	312 (00:01)	554 (00:01)	**184** (00:01)	TO (15:00)	310 (00:01)
Rax Extended	**1,288** (00:01)	TO (15:02)	50,176 (00:10)	TO (15:01)	TO (15:01)	TO (15:01)	TO (15:01)	TO (18:35)	TO (15:01)	TO (15:01)	OOM (23:39)
Rax EnvFirst	341,431 (01:08)	OOM (17:29)	**55,444** (00:12)	TO (15:05)	TO (:26:5)	1,541,069 (08:50)	TO (15:09)	TO (18:30)	TO (15:06)	TO (15:01)	OOM (12:48)
Reorder 5 2	**2,891** (00:02)	388,934 (00:39)	– (00:06)	332,688 (00:34)	297,498 (00:30)	297,683 (00:30)	37,223 (00:07)	– (00:05)	366,267 (00:39)	297,484 (00:31)	30,725 (00:07)
Readers Writers	3,341 (00:03)	TO (15:00)	TO (15:00)	4,087 (00:02)	1,480,875 (03:25)	65,023 (00:09)	1,263 (00:02)	**615** (00:01)	881,410 (02:32)	TO (15:00)	1,761 (00:01)
Sleeping Barber	17 (00:00)	**16** (00:00)	19 (00:00)	18 (00:00)	17 (00:00)	17 (00:00)	88 (00:00)	22 (00:00)	**16** (00:00)	17 (00:00)	65 (00:00)
TwoStage 5 5	**90,321** (00:15)	TO (15:00)	– (00:49)	TO (15:00)	TO (16:00)	TO (15:00)	TO (15:00)	– (01:03)	TO (15:00)	TO (15:00)	4,981,908 (09:47)
Wrong Lock 10 1	459 (00:01)	**56** (00:00)	18,173 (00:05)	104 (00:01)	19,315 (00:06)	17,480 (00:06)	128 (00:00)	86 (00:00)	**56** (00:00)	11,793 (00:05)	120 (00:01)

いて DFHS 優位である．一方，MostBlocked 対 BlockedNum については，13 プログラムについて，最良優先探索が優位であった．

表 6.4: 最良優先探索と DFHS の同一観点同士の比較結果

比較			結果 (プログラム数)		
最良優先探索	DFHS		最良優先探索	DFHS	同等または
優先度算出関数	枝刈り関数	枝選択順序最適化	優位	優位	未発見
Interleaving	Interleaving	–	7	16	2
Interleaving	Interleaving	Interleaving	5	17	3
Interleaving	NonConsecutive	–	5	17	3
Interleaving	NonConsecutive	Interleaving	6	15	4
MostBlocked	BlockedNum	–	13	10	2
Random	Random	–	10	14	1
Random	Random	Random	10	14	1

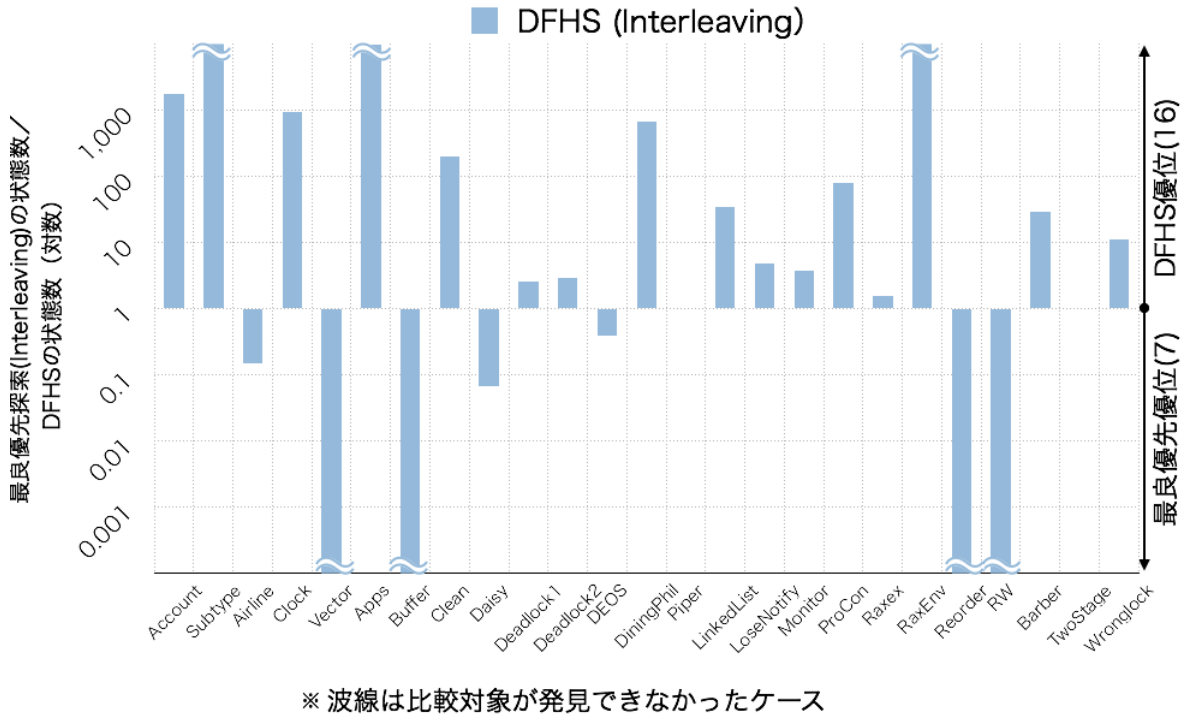

図 6.1: 最良優先探索（Interleaving）に対する DFHS (Interleaving) の性能比

実運用ユースケースを想定した評価

ここまで，従来手法と提案手法を同観点のヒューリスティック関数同士で比較することで提案手法がアルゴリズムの性能として，十分に有用であるかを評価した．次に，実際の運用ユースケースを想定した場合の評価を行う．

最良優先探索および DFHS を含むヒューリスティック探索では，プログラムごとに最適なヒューリスティック関数を選択することが，効率化を実現する上で重要であるが，実行に先立って最適なヒューリスティック

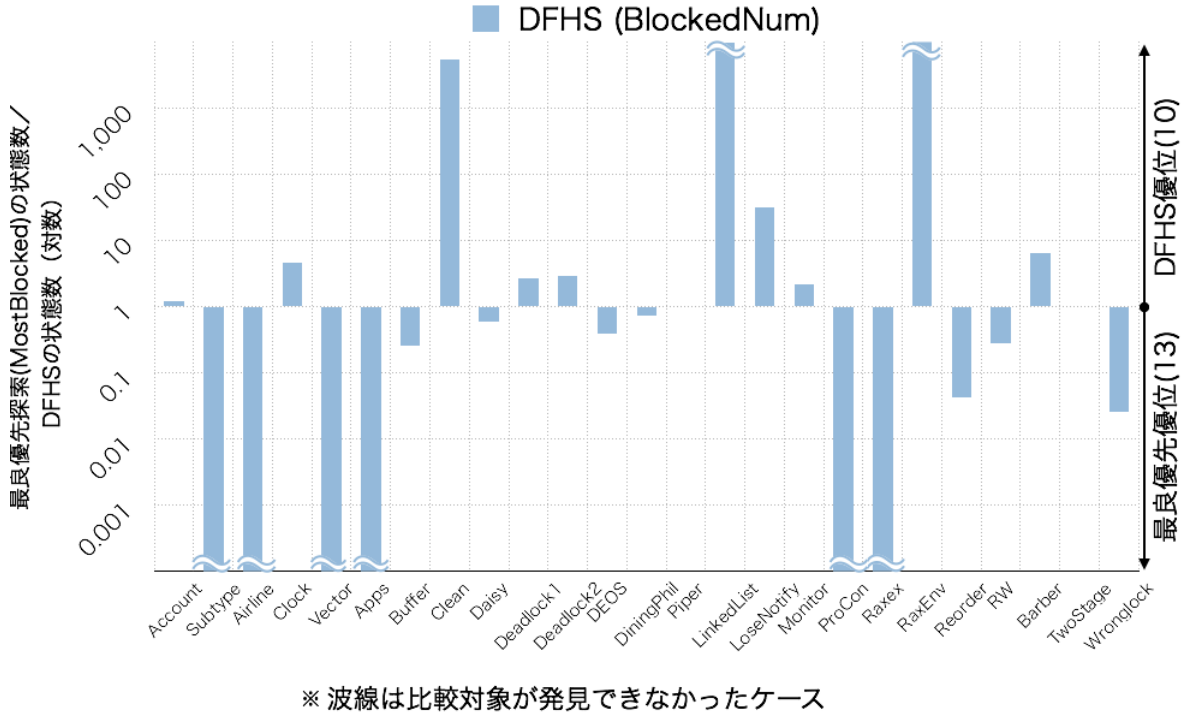

図 6.2: 最良優先探索（MostBlocked）に対する DFHS (BlockedNum) の性能比

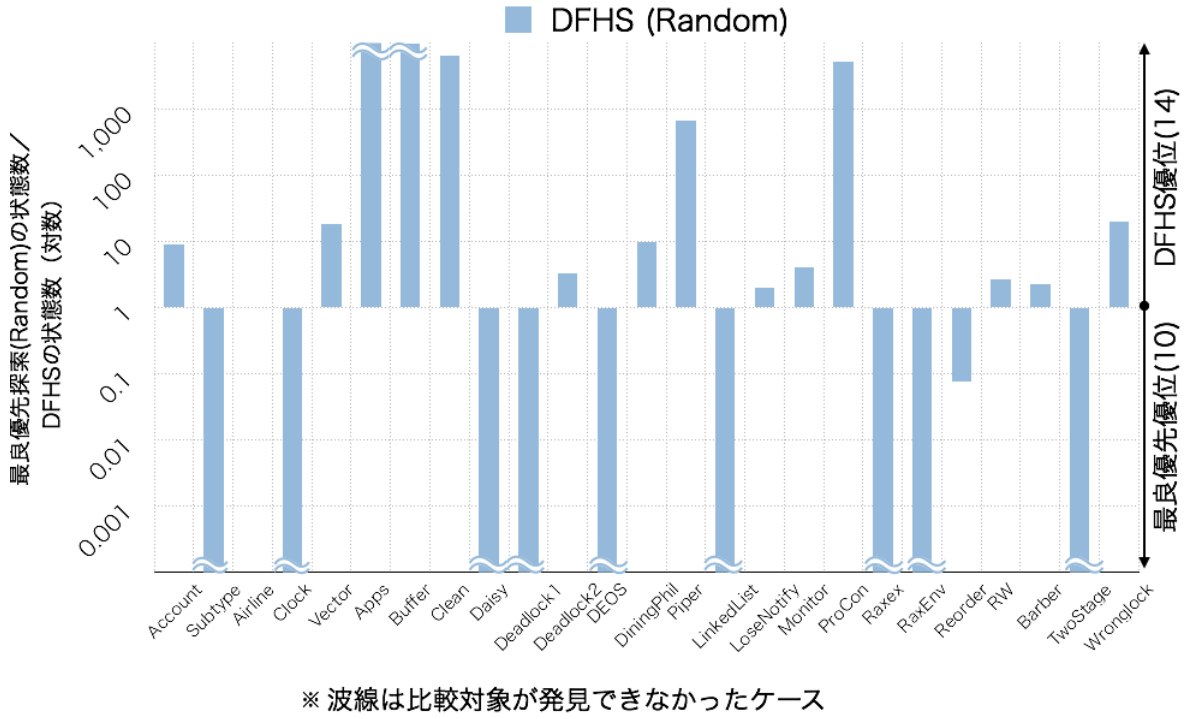

図 6.3: 最良優先探索（Random）に対する DFHS (Random) の性能比

を特定することは困難である．このような背景から，実際の運用では，複数のアルゴリズム，複数のヒューリスティックを同時並行に実行し，いずれかの手法によって効率化することで全体効率化を図るような運用が想定される．昨今のクラウド等，計算機資源を容易に確保できる状況を踏まえると，このような運用も十分現実的である．ここでは，最良優先探索，DFHS について各種ヒューリスティック関数を適用した結果の中から，各プログラムについて，それぞれのベストケースを抽出して比較することで，このような実運用での効率化性能について評価する．

以下，次に示す 4 つの観点で比較を行う．

- 深さ優先，最良優先探索（3 ポリシー），DFHS（枝選択最適化なし，3 ポリシー）の比較
- 最良優先探索（3 ポリシー） 対 DFHS（枝選択最適化なし，3 ポリシー）
- 従来手法（深さ優先＋最良優先 3 ポリシー） 対 DFHS（枝選択最適化なし，4 ポリシー）
- 従来手法（深さ優先＋最良優先 3 ポリシー） 対 DFHS（枝選択最適化あり，4 ポリシー）

(1) 深さ優先探索，最良優先探索，DFHS の比較

深さ優先探索に対する最良優先探索および DFHS の性能を評価する．図 6.4 に，深さ優先探索（表 6.2 の深さ優先探索の列）を基準としたときの，最良優先探索（表 6.2 の Interleaving, MostBlocked, Random の 3 ポリシーで最適であったもの.）および DFHS（表 6.2 の Interleaving, BlockedNum, Random の 3 ポリシーで最適であったもの）の効率比を，各プログラムについてグラフ化したものを示す．図 6.4 が示すように，最良優先探索

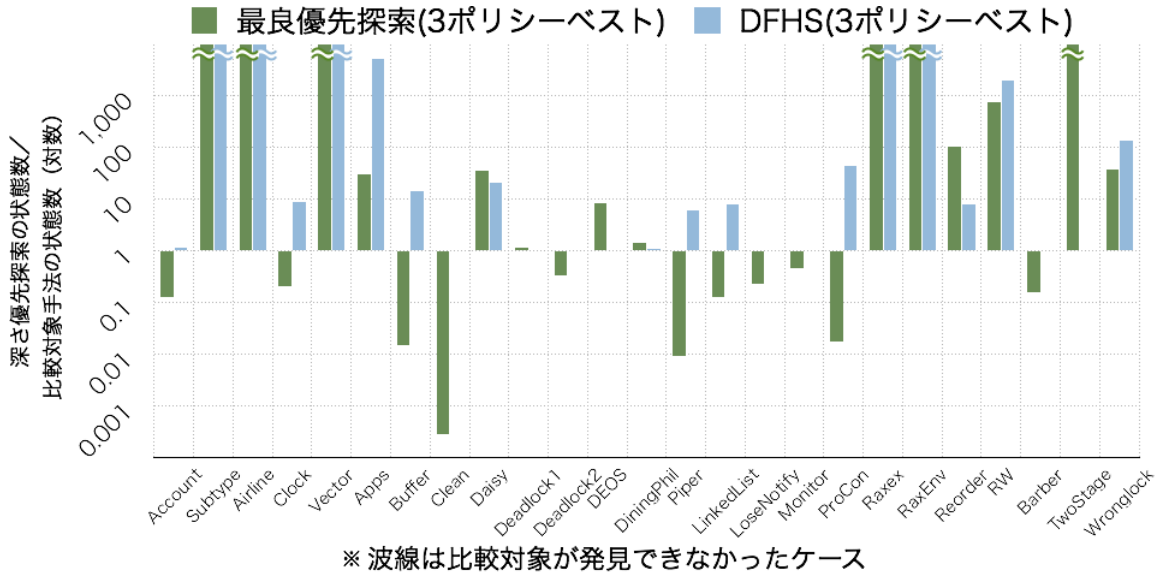

図 6.4: 深さ優先探索に対する最良優先探索および DFHS（枝選択最適化機能なし）の性能比

では，*Account*, *Clock*, *BoundedBuffer*, *Deadlock*1, *Piper*, *LinkedList*, *LoseNotify*, *NestedMonitor*, *ProCon*, *SleepingBarber* の 11 プログラムについて，深さ優先探索より悪い結果を示した．一方，DFHS は，これらを含めてた全てのプログラムについて，深さ優先探索と同等か優れた結果を示した． 特に，*AccountSubtype*, *AllocateVector*, *Airline*, *RaxEnvFirst* では，深さ優先ではタイムアウト/メモリエラーであったものが，1 秒〜12 秒で不具合を発見することができた．

(2) 最良優先探索と DFHS の比較

ヒューリスティック探索同士の比較として，最良優先探索と DFHS の比較を行う．図 6.5 に最良優先探索を基準としたときの DFHS の性能比をグラフ化したものを示す．グラフの見方については，最良優先探索を 1 とした点以外は，図 6.4 と同様である．この比較が示すように，25 プログラム中 17 プログラムについて，DFHS が優位な結果を示した．

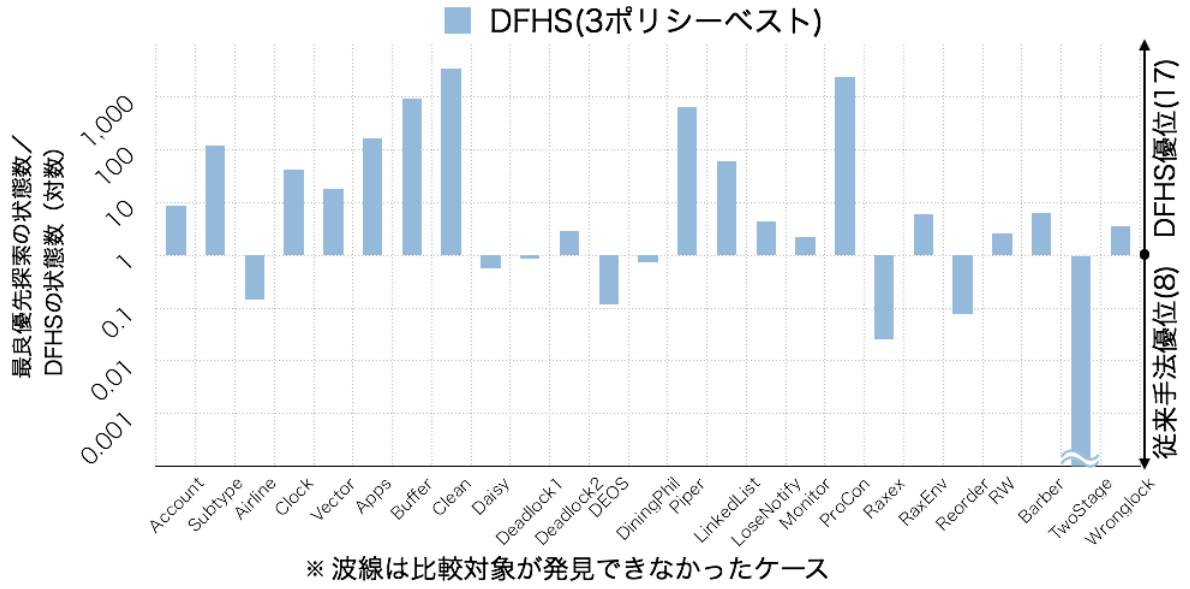

図 6.5: 最良優先探索に対する DFHS（枝選択最適化機能なし）の性能比

(3) 従来技術（深さ優先探索，最良優先探索）と DFHS との比較

最良優先探索を含めた従来技術に対する DFHS の効率化を評価する．次に，従来技術（深さ優先探索，最良優先探索）と DFHS との比較を行う．これによって，従来技術の深さ優先探索と最良優先探索を並行に実行させるのと比較して，DFHS を並行に実行させた場合の性能向上について評価する．図 6.6 に従来技術の深さ優先探索，最良優先探索（Interleaving，MostBlocked，Random）の 4 方式を適用した場合のベストケースに対して，提案手法（Interleaving，NonConsecutive，BlockedNum，Random）のベストケースを比較した結果のグラフ表示を示す．ここに示すように，12 プログラムで DFHS が優位，8 プログラムで従来手法が優位という結果となった．

この結果から，従来技術を 4 並列で実行した対して，DFHS を 4 並列で実行した場合は，4 プログラムで効率向上が可能となる．さらに，従来の 4 並列に加えて，DFHS を 4 並列で実行した場合は，12 プログラムで効率向上が可能となる．

(4) 従来技術（深さ優先探索，最良優先探索）と DFHS の比較 (枝選択最適化あり)

枝選択最適化の効果を検証する．図 6.6 と同様の比較パターンについて，枝選択最適化を適用した場合の結果を図 6.7 に示す．枝選択最適化適用後では，14 プログラムで DFHS が優位，9 プログラムで従来手法が優位という結果となった．この結果より，枝選択最適化による一定の効果が期待できる．

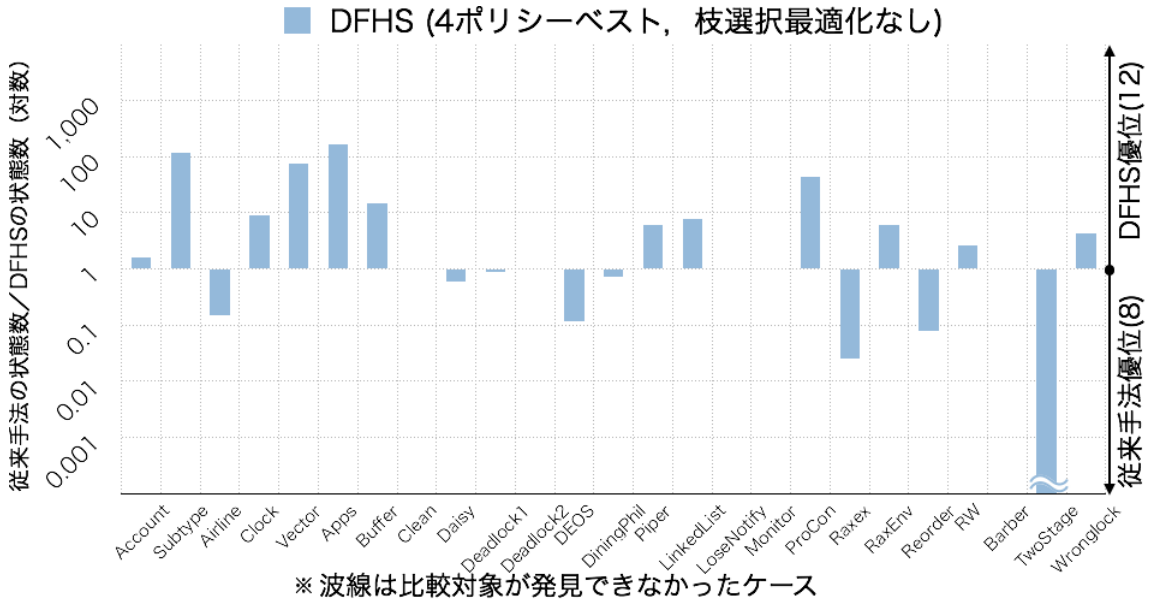

図 6.6: 従来技術に対する DFHS の性能比（枝選択最適化機能なし）

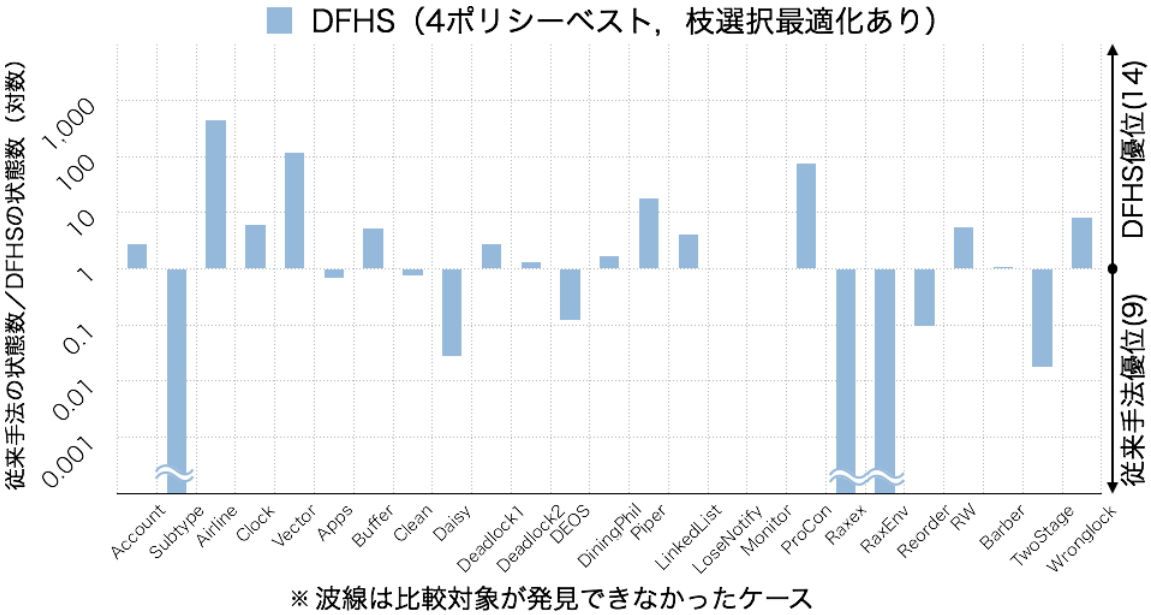

図 6.7: 従来技術に対する DFHS の性能比（枝選択最適化機能あり）

6.2.3　詳細実験

さらに詳細な性能評価を行うために，Account, DiningPhil Type1, DiningPhil Type2, AcountSubtype, Airline, AlarmClock, DESO, Reorder の各プログラムについて，各種パラメータ（スレッド数）を変化させて，追加実験を実施した．なお，本実験では，メモリ上限は 1024MB として実験している．また，DFHS は枝刈り機能のみの適用に絞っている．

探索効率

表 6.5: 安全性検証の追加実験結果 (枝刈り機能)

プログラム	深さ優先探索	最良優先探索			DFHS			
		Int	MostBlocked	Random	NonCon	BlockedNum	Int	Random
DiningPhil Type1 3	45 (00:00)	118 (00:01)	40 (00:00)	**21** (00:00)	27 (00:00)	15 (00:00)	45 (00:00)	**14** (00:00)
DiningPhil Type1 7	33,020 (00:07)	– (00:09)	**306** (00:01)	6,219 (00:03)	15,278 (00:05)	**53** (00:01)	– (00:04)	498 (00:01)
DiningPhil Type1 10	3,734,639 (06:59)	10,295 (00:03)	**768** (00:01)	161,700 (00:23)	1,819,380 (03:33)	**92** (00:01)	– (01:48)	80,034 (00:12)
DiningPhil Type1 15	TO (15:00)	3,193,117 (06:43)	**2,278** (00:03)	– (03:43)	TO (15:00)	**177** (00:01)	TO (15:00)	TO (15:00)
DiningPhil Type2 10	297 (00:01)	10,295 (00:03)	**759** (00:01)	1,614 (00:01)	171 (00:01)	91 (00:01)	**80** (00:02)	353 (00:01)
DiningPhil Type2 100	29,997 (00:10)	TO (15:00)	**122,499** (00:34)	TO (14:59)	15,156 (00:06)	5,446 (00:03)	TO (15:00)	**3,782** (00:03)
DiningPhil Type2 200	119,997 (01:13)	TO (15:00)	**284,999** (02:14)	TO (15:00)	60,306 (00:31)	20,896 (00:11)	TO (14:59)	**6,240** (00:05)
DiningPhil Type2 500	OOM (02:10)	TO (15:00)	TO (15:00)	TO (15:00)	OOM (02:09)	127,246 (02:01)	TO (15:00)	**14,709** (00:17)
DiningPhil Type2 700	OOM (03:18)	TO (15:00)	TO (15:00)	TO (15:00)	OOM (04:27)	248,146 (10:04)	247,757 (09:14)	**22,443** (00:45)
Account Subtype 1,2	499 (00:01)	**11,845** (00:03)	238,375 (00:44)	16,642 (00:06)	**425** (00:01)	– (00:01)	878 (00:01)	– (00:01)
Account Subtype 2,2	533 (00:02)	**3,052** (00:01)	418,543 (01:23)	37,546 (00:10)	**290** (00:01)	– (00:01)	1,099 (00:01)	– (00:01)
Account Subtype 3,1	346,796 (01:12)	62,977 (00:11)	414,093 (01:39)	**2,423** (00:02)	13,436 (00:06)	– (00:01)	**279** (00:01)	– (00:01)
Account Subtype 5,1	TO (14:59)	**110,812** (00:16)	547,157 (01:40)	1,657,656 (04:34)	TO (15:00)	– (00:00)	**535** (00:01)	– (00:00)
Account Subtype 5,2	**635** (00:02)	1,399,090 (03:06)	**673,308** (02:39)	2,510,001 (09:30)	948 (00:02)	– (00:01)	**666** (00:01)	– (00:01)
Account Subtype 5,5	772 (00:01)	4,692,439 (10:51)	**1,460,920** (09:11)	TO (14:59)	**540** (00:01)	– (00:01)	1,317 (00:01)	– (00:01)
Account Subtype 8,1	TO (15:00)	3,606,376 (08:23)	**582,812** (02:39)	TO (15:00)	TO (15:00)	– (00:01)	**218,244** (00:33)	– (00:01)
Account Subtype 8,2	**737** (00:02)	**4,543,414** (10:55)	– (04:49)	TO (14:59)	**1,022** (00:02)	– (00:01)	1,398 (00:01)	– (00:01)
Airline 3,1	1,906 (00:01)	**71** (00:01)	1,817 (00:02)	824 (00:02)	**569** (00:01)	– (00:01)	626 (00:01)	– (00:01)
Airline 4,1	39,611 (00:06)	**94** (00:01)	27,169 (00:05)	739 (00:01)	6,302 (00:02)	– (00:01)	**2,447** (00:01)	– (00:01)
Airline 5,1	946,331 (01:43)	**117** (00:01)	64,892 (00:16)	4,795 (00:02)	76,044 (00:11)	– (00:01)	**7,366** (00:03)	– (00:01)
Airline 5,2	1,385,307 (03:49)	**106** (00:01)	46,283 (00:11)	6,265 (00:03)	118,046 (00:23)	– (00:01)	**7,441** (00:03)	– (00:01)
Airline 6,1	TO (15:00)	**140** (00:01)	109,459 (00:20)	92,873 (00:18)	748,550 (02:02)	– (00:01)	**8,593** (00:03)	– (00:01)
AlarmClock	662 (00:02)	5,983 (00:03)	1,607 (00:02)	**723** (00:01)	173 (00:02)	– (00:01)	**146** (00:01)	– (00:01)
Reorder 2,2	2,134 (00:02)	2,241 (00:01)	**1,078** (00:02)	1,465 (00:02)	**1,905** (00:02)	– (00:01)	– (00:02)	– (00:01)
Reorder 5,2	457,159 (01:29)	63,766 (00:11)	**19,568** (00:04)	– (00:16)	**218,566** (00:47)	– (00:01)	– (00:02)	– (00:01)
Reorder 5,5	TO (15:00)	– (00:45)	**28,945** (00:09)	– (00:54)	TO (15:00)	– (00:01)	– (00:03)	– (00:01)

実験結果を表 6.5 に示す． また，DFHS が，深さ優先探索および最良優先探索と比較してどの程度有効であったかを状態数の比で比較したものを表 6.6 に示す．表には結果の優劣が顕著であったプログラムのみを記載している．表中の値は，比較対象の手法による状態数を DFHS の状態数で割った値であり，値が大きいほど DFHS が優位であることを意味する．また，∞ は DFHS のみが不具合発見に成功したことを示し，– は DFHS では不具合を発見できなかったことを示す．

表 6.6: DFHS の効率比

プログラム	DFHS の効率比	
	深さ優先探索比	最良優先探索比
DiningPhil Type1 10	40,593	8.35
DiningPhil Type1 15	∞	12.87
DiningPhil Type2 200	19.23	45.67
DiningPhil Type2 700	∞	∞
Account Subtype 5,1	∞	207.13
Account Subtype 8,1	∞	2.67
Airline 6,1	∞	0.02
AlarmClock	4.53	4.95
Reorder 5,5	–	–

実験の結果，DiningPhil Type1/Type2, AccountSubtype, AlermClock について，最良優先探索に対して DFHS が優位となった．この結果より，従来手法の効果が低い，あるいは苦手とするプログラムについて提案手法が有効となる場合があることを確認した．対象プログラムに対して，従来手法と提案手法を逐次あるいは同時に適用することで，互いの手法が苦手とするケースを補完しながら不具合発見の効率化を図ることが期待できる．以下，各プログラムの結果について説明する．

AccountSubtype: 最良優先探索が深さ優先探索より悪い結果を示しているのに対して，DFHS(Interleaving) は，深さ優先探索の効率が非常に悪いケース (3,1), (5,1), (8,1) において，優位 (∞) となっている．

DiningPhil Type1: 最良優先探索に対して DFHS(BlockedNum) が 8 倍〜12 倍優位な結果となった．また DFHS は，深さ優先探索に対して 40,000 倍〜∞ の優位となった．Type1 は，哲学者が食事を一度しか行わないため，不具合の発生条件がシビアである，すなわち，探索空間中の不具合状態が少ないため，DFHS による削減効果が非常に大きく働いたと考えられる．

DiningPhil Type2: 最良優先探索は深さ優先探索より悪い結果を示しているのに対して，DFHS(Random) は，深さ優先探索に対して，19 倍〜∞ の優位となっている．Random による打ち切り確率は 0.8 に設定している．大幅な打ち切りを行っているにもかかわらず良い結果が出たのは，探索空間内での不具合の分布が多かったためと考えられる．

AlarmClock: 最良優先探索は深さ優先探索より悪い結果を示している．これに対して DFHS(Interleaving) は深さ優先探索と比較して，4 倍程度，優位となった．

Airline, Reorder: 最良優先探索 (Interleaving) が最も優位な結果を示した．DFHS(NonConsecutive) は，深さ優先探索よりは優位であったが，最良優先探索の結果には至らなかった．

メモリ効率

表 6.7 に，最大メモリ使用量について特徴的な結果であったプログラムの結果を示す．また，図 6.8 にメモリ消費量の比較グラフを示す．各手法とも，最もメモリ使用量が少なかったポリシーの結果を採用した．

表 6.7: 最大メモリ使用量 (MB)

プログラム	深さ優先探索	最良優先探索	DFHS
DiningPhil Type2 500	OOM (1,015)	TO (194)	**114**
DiningPhil Type2 700	OOM (1,015)	TO (168)	**257**
Account Subtype 5,5	81	**1,015**	81
Account Subtype 8,2	81	**244**	81

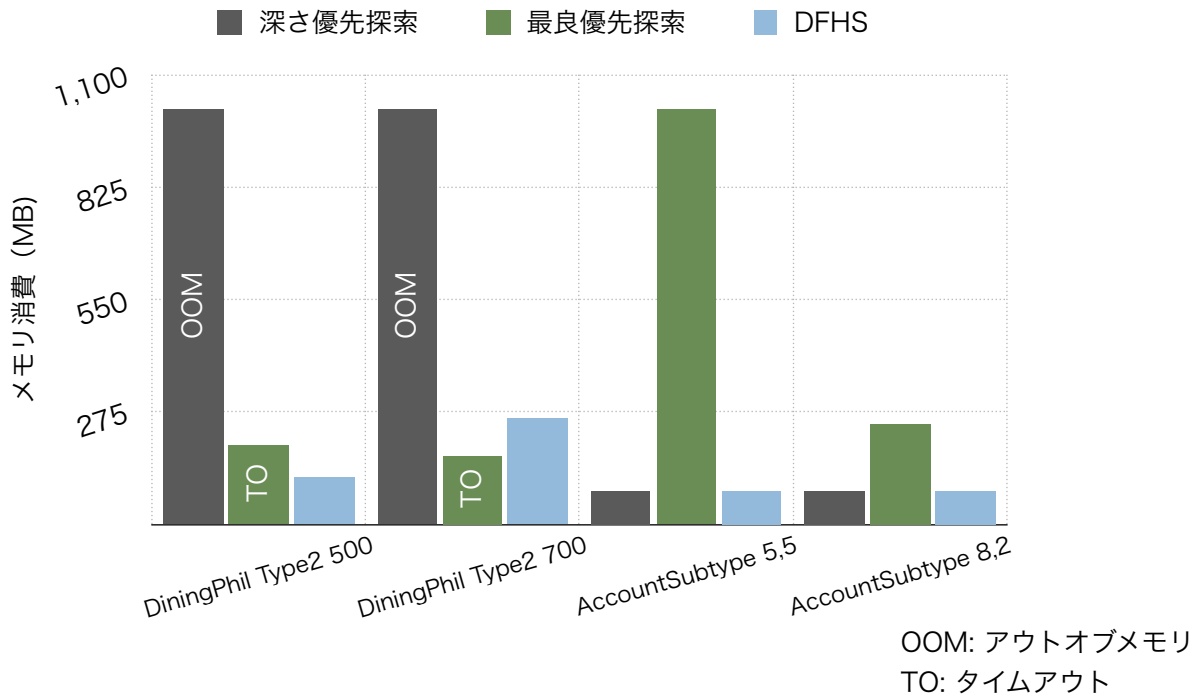

図 6.8: 最大メモリ使用量の比較

DFHS は，深さ優先探索，最良優先探索と比較してどのプログラムについても，同じかより少ないメモリ使用量であった．最良優先探索は，探索効率が悪かったケース (AcountSubtype(5,5), (8,2)) において最大メモリ使用量が増大していることが確認された．一方，DFHS は，DiningPhil Type2 (500), (700) で若干メモリの増大がみられたが，これらのケースでは，深さ優先探索，最良優先探索とも不具合の発見に失敗している．今回の実験からは，DFHS はメモリ消費面でも有利であることが示された．

3.2 節および 4.1 節で述べた通り，最良優先探索は，優先度付けした探索状態を探索キューで管理するのに対して，DFHS は，探索パス上の状態をスタックに保持する．状態を保存しておく意味では同じであるが，DFHS は，スタックを深さ方向に最大限利用できるため，仮に探索キューと同一の上限を設けたとして

も，より深い状態に到達することができるため探索効率が高い．メモリ消費の実験結果はこれを裏付けるものである．

探索空間の構造の影響

本節では探索空間の違いによる各手法の有効性について考える．まず，DiningPhil Type1 と DiningPhil Type2 に着目する．Type1/Type2 ともに DFHS が，最良優先探索より優位な結果を示した．さらに，Type2 の場合，従来手法の最良優先探索は，深さ優先探索よりも悪い結果となっている．Type1 は，各哲学者が一度しか食事を行わないのに対して Type2 は，哲学者が繰り返し食事を行う．そのため，Type2 は Type1 と比較して深さ方向にも長い探索空間となる[5]．そのため，幅方向に探索される傾向がある最良優先探索は，Type1 にはある程度有効に働いたが，Type2 に対しては効果がなかったと考えられる．これに対して，DFHS は，深さ方向の探索に有利で，幅方向の広がりの影響を受けないため，効果が大きかったと考えられる．Type2 のスレッド数が非常に多いケース (100〜700) のように，各探索状態からの遷移先候補が多いケースでは，幅方向の探索状態の候補が増大するため，この傾向がより顕著になる．

また，AccountSubtype についても，不具合のチェック ASSERT がプログラムの終盤に存在するため，深さ方向への探索が必要であるため，DFHS が最良優先探索より優位な結果を示したと考えられる．

一方，最良優先探索が DFHS より優れた結果を見せた Airline および Reorder については，不具合 (ASSERT 違反，例外発生) が，プログラムの各所で出現する．

表 6.8: 最良優先探索におけるキュー溢れ回数

プログラム	最良優先探索	
	状態数	キュー溢れ回数
DiningPhil Type1 15	2,278	0
DiningPhil Type2 200	284,999	**181,107**
AccountSubtype 5,2	673,308	235,717
AccountSubtype 5,5	1,460,920	**782,952**
AccountSubtype 8,1	582,812	204,365
AccountSubtype 8,2	4,543,414	**3,081,557**
Airline 6,1	140	0
AlarmClock	723	0
Reorder 5,5	28,945	4,179

以上の考察について，最良優先探索の探索キュー溢れ回数の観点からも検証する，最良優先探索において，キュー溢れ回数が特徴的なケースについて表 6.8 に示す．また，図 6.9 にグラフを示す．

前述の DiningPhil Type2 や AccountSubtype において結果が悪かったプログラムについて，探索キュー溢れの発生回数が多くなっている (表中の太字)．一方，他の良い結果を示したプログラムでは探索キュー溢れの発生回数が少ない (表中の太字以外)．このことから，探索キュー溢れにより重要パスの取りこぼしが発生し，状態数の増大につながっていると考えられる．

以上から，次のようなプログラムにおいて，DFHS が優位になる可能性があると考えられる．

深い探索空間に不具合が分布するケース: 最良優先探索では，探索の初期段階で優先度に優劣がつかず，探索の浅い段階を抜けられないことによって不具合の本質になかなか到達できない事がある．探索空間の

[5] この Type1 と Type2 の違いは，同種のプログラム，同種の不具合であっても，探索空間が全くが異なることを意味する．これは，最適な探索手法やヒューリスティック関数を事前に特定することが困難であることも示している．

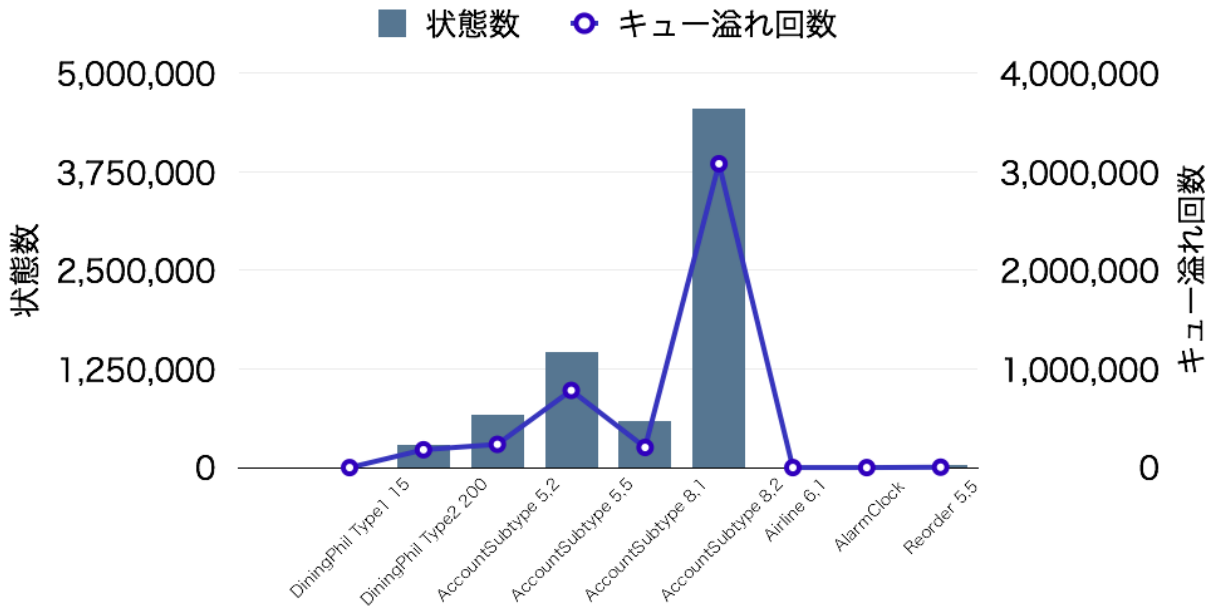

図 6.9: 最良優先探索におけるキュー溢れ回数

深いところに多数不具合が分布するような場合，DFHS は，深さ方向の探索を優先して実行することで優位になる可能性がある．

非決定要因が多いケース: スレッド数が多い場合や，共有リソースへのアクセスが複雑に影響し合うようなプログラムでは，各状態からの遷移候補が多いため，各状態からの遷移候補を全て展開する最良優先探索が非効率となる．また，探索キューサイズ制限の影響を受けやすい．このようなケースにおいても，DFHS は，幅方向の広がりに影響を受けずに探索可能である．

6.3　LTL 検証実験

次に本節では，LTL 検証に対する DFHS の有効性を評価する．前述の通り，従来手法である最良優先探索は，LTL 検証への適用が困難である．そのため，LTL 検証の一般的アルゴリズムである NDFS と提案手法の DFHS との間で比較実験を行う．

6.3.1　実験環境

検証には，安全性検証で用いたベンチマークプログラム (表 6.1)，に加えて，元のベンチマークセット [32] に含まれており，活性検証に適したアプリである Elevator および LTL 検証用に本研究で作成したプログラムを使用する．安全性検証で用いたベンチマークプログラムは，安全性検証用の不具合が作り込まれているため，それぞれのプログラムに含まれる安全性の不具合に対して，それを検証するための適切な LTL 式を設定して検証した．その際，LTL 検証の妨げとなる ASSERT 文を削除するなど，プログラムの一部修正を行った．また，ここで新たに追加したプログラムは，より活性の検証に即した不具合を含んでおり，実践的な LTL 検証における評価ができるような内容である．表 6.9 に追加プログラムの一覧を示す．

表 6.9: 追加プログラム一覧

プログラム (パラメータ)	内容	不具合の種類	ソース行数
Elevator (n, m, l)	人およびエレベータがスレッドとして表現されたシミュレーションプログラム.	リクエストしてもエレベーターがこないバグを含む	1,188
AppleOrange (n, m, l)	Producer スレッドは，apple または orange をキューに投入する．Consumer はキューから一つ取得する..	リソースを取れないバグを含む	491
LockThree	3 スレッドがリソースプールからリソースを取得するプログラム.	リソースを取れないバグを含む	135
LockUnlock (n, m, l, o)	複数スレッドがリソースプールからリソースを取得する．そのうちの一つのスレッドは，リソースが空いている場合，特定のリソースの取得を試行する.	所望のリソースを永久に取れないバグを含む	129

各プログラムに設定する LTL 式は，5.3.1 項で示した LTL 式と同様の形式の LTL 式を適用した．このとき，常に動作し続けるプログラム（終了しないプログラム）については，各スレッドが公平にスケジュールされている条件での検証を行うため，公平性付きの LTL 式を設定した．公平性を考慮した検証は，常に動作し続けるプログラムにおいてのみ考慮すべき性質であるためである．以下，各プログラムの LTL 検証内容について説明する.

Elevator: LTL 式は,「ある人がエレベータを要求するボタンを押すと必ずエレベータが到着する」，という活性を表現する．なお，これは公平性条件付の LTL 式である[6].

リスト 6.1: Elevator プログラムの LTL 式

```
([]<>person_enb1->[]<>person_run1)->
([]<>elevator_enb1->[]<>elevator_run1)->
        [](push_down1-><>elevator_arrive1)
```

なお，DFHS の公平性最適化機能を適用する場合は，以下のように公平性を表す部分を付加せずに実行する．以後のアプリも同様である.

リスト 6.2: Elevator プログラムの LTL 式 (公平性条件なし)

```
[](push_down1-><>elevator_arrive1)
```

AppleOrange: LTL 式は,「Consumer は無限回アップルを取得することができる」，という活性を表現する.

リスト 6.3: AppleOrange プログラムの LTL 式

```
([]<>Producer_enb0->[]<>Producer_run0)->
([]<>Producer_enb1->[]<>Producer_run1)->
([]<>Consumer_enb0->[]<>Consumer_run0)->
([]<>Consumer_enb1->[]<>Consumer_run1)->
      (<>Produced1->[]<>canEatApple1)
```

LockThree: LTL 式は,「リソースを無限回取得できること」を表現する.

リスト 6.4: LockThree プログラムの LTL 式

```
([]<>enb1->[]<>run1) ->
([]<>enb2->[]<>run2) ->
([]<>enb3->[]<>run3) ->
       []<>pos1
```

[6]このプログラムは，人，エレベーターのカゴが全てスレッドとして表現されたエミュレーションプログラムである．そのため，人およびエレベータそれぞれのスレッドについての公平性条件を付与している.

LockUnlock: LTL 式は,「特定リソースをいつかは取得できること」を表現する.

リスト 6.5: LockUnlock プログラムの LTL 式

```
([]<>enb1->[]<>run1)->
([]<>enb2->[]<>run2)->
([]<>enb3->[]<>run3)->
        [](want3_2-><>pos3_2)
```

28 のプログラムに対して LTL 検証を行い，最初の反例が見つかるまでの状態数で評価する．DFHS を構成する三つの機能（枝刈り機能，枝選択順序最適化機能，公平性最適化機能）について，枝刈り機能および枝選択順序最適化機能については，全ての実験対象プログラムに対して適用した．枝刈り関数および枝選択順序は，それぞれ，4.1 節，4.2 節で説明したものを適用した．また，公平性最適化機能については，公平性条件付き LTL を適用したプログラムに対してのみ実施し，適用あり，適用なし，の両ケースについて実験した．なお，メモリの最大サイズおよびタイムアウトは，6.2.2 項の安全性検証の実験と同じ設定とした．

6.3.2　実験結果

表 6.10 に公平性最適化なしの場合の実験結果を，表 6.11 に公平性最適化ありの場合の実験結果を示す．各セル中の数字は，探索した同期積（プログラムの状態遷移空間と，LTL 式から作成した Büchi オートマトンとの同期積）の状態数を表す．下線で示したセルは，それぞれのプログラムにおいて，最も性能が良かったものを示す．(1)(2)(3) はそれぞれ，枝刈り，枝選択順序最適化，公平性最適化を適用した場合を示す．プログラム *Account* から *Wronglock* は，終了するプログラム，*Daisy* から *SleepingBarber* は終了しない（無限に動作し続ける）プログラムである．*Elevator* から *LockUnlock* までは，本研究のために，無限ループおよび活性の不具合を含むように作成したオリジナルのプログラムである．

アルゴリズム性能評価

DFHS の性能を評価するために，NDFS と DFHS の各枝刈り関数について，性能が良かったプログラム数を比較する（表 6.10 の NDFS の列と DFHS のうちの一列を比較し，全 28 プログラム中で優位となったプログラム数を，DFHS のそれぞれの列について整理）．表 6.12 に，DFHS の各枝刈り関数の比較結果を示す．また，結果をグラフ化したものを図 6.10，図 6.11 に示す．本結果が示すように，Interleaving, NonConsecutive, LessInterleaving の枝刈り関数を採用した 6 件については，いずれも DFHS の方が性能が良いプログラム数が上回った．一方，Random の枝刈り関数では，2 件とも下回った．この結果より，既存の NDFS の代わりに DFHS を適用した場合，効率化できる可能性が十分に高いといえる．

実運用ユースケースを想定した評価

ここまで，従来手法と提案手法を同観点のヒューリスティック関数同士で比較することで提案手法がアルゴリズムの性能として，十分に有用であるかを評価した．次に，実運用におけるユースケースを想定した評価を行う．各プログラムについて，NDFS と，DFHS（(1) 枝刈り機能, (2) 枝選択順序最適化）のベストケースについて比較する．図 6.12 に NDFS を 1 としたときの DFHS の効率比を示す．この結果より，28 プログ

表 6.10: LTL 検証の結果

プログラム	NDFS	DFHS							
		(1)				(1) + (2)			
		Int	NonCon	LessInt	Random	Int/ Int	NonCon/ Int	LessInt/ LessInt	Random/ Random
Account 5	TO (15:00)	**346,389** (01:25)	4,510,368 (14:59)	TO (14:59)	TO (14:59)	536,502 (01:58)	TO (14:59)	TO (14:59)	TO (14:59)
AccountSubtype 2 1	TO (15:00)	**474,725** (01:33)	4,524,607 (13:56)	TO (15:00)	– (00:59)	557,839 (01:52)	TO (14:59)	TO (14:59)	– (02:54)
Airline 5 2	147 (00:01)	12,724 (00:15)	299 (00:03)	147 (00:01)	TO (15:00)	280 (00:03)	147 (00:03)	**85** (00:00)	TO (14:59)
AllocateVector	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (15:00)	TO (14:59)	TO (14:59)	TO (14:59)	TO (14:59)
Apps	74,931 (00:19)	20,468 (00:08)	39,881 (00:11)	**1,005** (00:02)	– (00:12)	400,615 (01:20)	32,664 (00:12)	58,823 (00:18)	– (00:15)
Deadlock1	268 (00:01)	283 (00:01)	174 (00:01)	268 (00:01)	– (00:01)	258 (00:00)	**126** (00:00)	188 (00:01)	– (00:01)
Deadlock2	74 (00:00)	74 (00:00)	72 (00:00)	74 (00:00)	– (00:01)	34 (00:01)	**32** (00:00)	118 (00:00)	– (00:01)
DEOS	**87,231** (09:58)	**87,231** (10:03)	**87,231** (10:05)	**87,231** (11:41)	– (00:12)	**87,231** (10:03)	**87,231** (11:08)	**87,231** (12:11)	– (00:10)
LinkedList 2 20	TO (14:59)	TO (14:59)	**2,640,362** (06:30)	TO (15:00)	– (00:01)	TO (15:00)	TO (15:00)	TO (14:59)	– (00:01)
LoseNotify 3 3 3	**20** (00:00)	55 (00:00)	51 (00:00)	**20** (00:00)	30 (00:00)	48 (00:00)	26 (00:00)	**20** (00:00)	30,120 (00:12)
piper 1 3 2	TO (14:59)	**546,163** (01:50)	TO (14:59)	TO (14:59)	TO (15:00)	637,716 (02:01)	TO (14:59)	TO (14:59)	TO (14:59)
ProCon 2 5 6	81,210 (00:21)	603 (00:01)	102,738 (00:25)	TO (14:59)	TO (15:00)	642 (00:01)	**196** (00:01)	81,206 (00:19)	TO (15:00)
Reorder	17,841 (00:07)	**2,276** (00:02)	13,053 (00:07)	17,841 (00:07)	– (00:35)	5,335 (00:03)	13,027 (00:06)	17,824 (00:07)	– (00:30)
ReadersWriters 2 2 100	2,561,430 (05:56)	985,973 (02:15)	**6,469** (00:04)	2,477,556 (06:38)	TO (15:00)	19,244 (00:05)	888,914 (02:42)	2,554,809 (07:38)	TO (15:00)
TwoStage 1 2	32,725 (00:10)	**7,019** (00:04)	24,854 (00:09)	32,725 (00:11)	– (00:05)	7,600 (00:05)	25,677 (00:10)	36,507 (00:12)	– (00:05)
Wronglock 3 2	TO (14:59)	**165,458** (00:39)	TO (14:59)	TO (14:59)	TO (14:59)	183,377 (00:39)	TO (14:59)	TO (14:59)	TO (14:59)
Daisy	21,862 (00:10)	21,862 (00:10)	2,604 (00:03)	TO (14:58)	– (00:15)	**1,522** (00:03)	**1,522** (00:03)	1,635 (00:02)	– (00:12)
BoundedBuffer 1 1 2 2	2,825,090 (11:52)	**35,845** (00:09)	1,194,051 (05:07)	4,769,245 (09:36)	50,211 (00:12)	40,785 (00:10)	1,649,872 (09:16)	TO (14:56)	42,366 (00:10)
Clean 1 2 3	OOM (03:31)	OOM (06:29)	TO (14:58)	TO (37:35)	TO (16:51)	OOM (06:46)	OOM ()	TO (14:54)	OOM (18:26)
DiningPhil 3	3,519,876 (06:52)	**211,912** (00:28)	2,272,538 (04:34)	3,116,457 (06:00)	361,666 (00:45)	250,254 (00:34)	2,360,218 (04:47)	3,377,763 (06:36)	358,669 (00:45)
NestedMonitor	**62** (00:00)	**62** (00:00)	211 (00:00)	**62** (00:00)	– (00:01)	146 (00:00)	146 (00:00)	**62** (00:00)	– (00:01)
RaxEnvfirst	TO (27:05)	TO (16:25)	TO (14:57)	TO (14:55)	TO (14:59)	TO (16:29)	TO (15:01)	TO (14:55)	OOM (32:14)
Raxextended	TO (14:55)	976,181 (01:53)	TO (14:54)	TO (14:55)	1,104,188 (01:56)	**721,387** (01:21)	TO (14:54)	TO (14:54)	1,432,702 (02:27)
SleepingBarber	TO (14:11)	**323,565** (00:44)	TO (14:11)	TO (14:10)	1,082,807 (02:26)	402,213 (00:54)	TO (14:10)	TO (14:09)	954,604 (02:11)
Elevator 4 3 4	OOM (18:09)	734,636 (01:50)	TO (15:01)	TO (14:59)	253,115 (01:05)	**93,903** (00:23)	TO (15:02)	TO (14:59)	OOM (17:33)
AppleOrange 2 2 2	OOM (08:10)	TO (14:47)	TO (17:06)	TO (14:47)	TO (14:47)	TO (14:48)	OOM (04:57)	TO (14:47)	OOM (12:01)
LockThree	1,814,669 (09:20)	**71,715** (01:03)	1,084,153 (07:51)	1,385,777 (07:42)	82,324 (00:52)	105,359 (01:22)	1,225,165 (09:04)	1,515,386 (07:34)	114,803 (01:00)
LockUnlock 3 3 2 2	1,028,749 (05:57)	– (00:50)	731,249 (04:22)	628,149 (01:58)	**41,791** (00:14)	– (00:35)	1,171,056 (09:25)	692,361 (02:17)	66,920 (00:21)

表 6.11: LTL 検証の結果 (公平性最適化適用)

プログラム	DFHS								
	(3)	(1) + (3)				(1) + (2) + (3)			
		Int	NonCon	LessInt	Random	Int/ Int	NonCon/ Int	LessInt/ LessInt	Random/ Random
Daisy	20,742 (00:11)	20,742 (00:12)	29,044 (00:12)	2,284,374 (12:20)	– (00:03)	418,910 (01:26)	29,453 (00:14)	**1,614** (00:03)	– (00:03)
BoundedBuffer 1 1 2 2	19,132 (00:13)	1,104 (00:01)	139,976 (00:32)	3,592 (00:03)	– (00:18)	2,153 (00:02)	**145** (00:01)	3,662 (00:04)	– (00:18)
Clean 1 2 3	OOM (02:37)	TO (15:00)	**655,922** (01:46)	4,009,441 (09:16)	– (00:24)	TO (14:59)	TO (15:52)	5,871,552 (14:35)	– (00:35)
DiningPhil 3	135 (00:00)	3,998 (00:04)	137 (00:00)	826 (00:01)	– (00:13)	82 (00:00)	69 (00:00)	**38** (00:00)	8,406 (00:05)
NestedMonitor	46 (00:00)	46 (00:00)	811 (00:01)	46 (00:00)	365 (00:01)	225 (00:00)	755 (00:01)	46 (00:00)	**28** (00:00)
RaxEnvfirst	OOM (18:09)	TO (14:59)	TO (15:03)	TO (14:59)	TO (15:00)	**8,700** (00:04)	TO (15:03)	OOM ()	TO (17:37)
Raxextended	4,214 (00:04)	1,582 (00:01)	4,038 (00:02)	307 (00:01)	5,092 (00:04)	1,765 (00:01)	**35** (00:00)	50,271 (00:14)	101,185 (00:22)
SleepingBarber	**28** (00:00)	35 (00:00)	61 (00:00)	**28** (00:00)	460,802 (01:15)	38 (00:00)	**28** (00:00)	**28** (00:00)	842,546 (02:11)
Elevator 4 3 4	TO (17:23)	203,846 (00:37)	TO (15:02)	TO (15:00)	TO (10:23)	**30,704** (00:11)	TO (15:04)	TO (15:00)	OOM (18:22)
AppleOrange 2 2 2	**408** (00:01)	37,594 (00:14)	1,046 (00:02)	381,397 (01:29)	1,159,729 (07:47)	73,538 (00:23)	2,097 (00:04)	441,587 (01:47)	TO (15:00)
LockThree	3,575 (00:06)	**651** (00:02)	1,773 (00:04)	2,487 (00:04)	– (00:43)	15,224 (00:36)	2,313 (00:04)	1,925 (00:04)	20,220 (00:19)
LockUnlock 3 3 2 2	515 (00:02)	– (00:08)	1,259 (00:03)	5,543 (00:06)	1,500 (00:03)	– (00:09)	410 (00:02)	**146** (00:01)	31,621 (00:15)

表 6.12: NDFS と DFHS の各ポリシー採用時の比較結果

DFHS		結果 (プログラム数)		
枝刈り関数	枝選択順序最適化	NDFS 優位	DFHS 優位	同等または未発見
Interleaving	–	4	15	9
Interleaving	Interleaving	6	16	6
NonConsecutive	–	4	14	9
NonConsecutive	Interleaving	3	11	14
LessInterleaving	–	3	5	20
LessInterleaving	LessInterleaving	3	10	15
Random	–	12	7	9
Random	Random	12	6	10

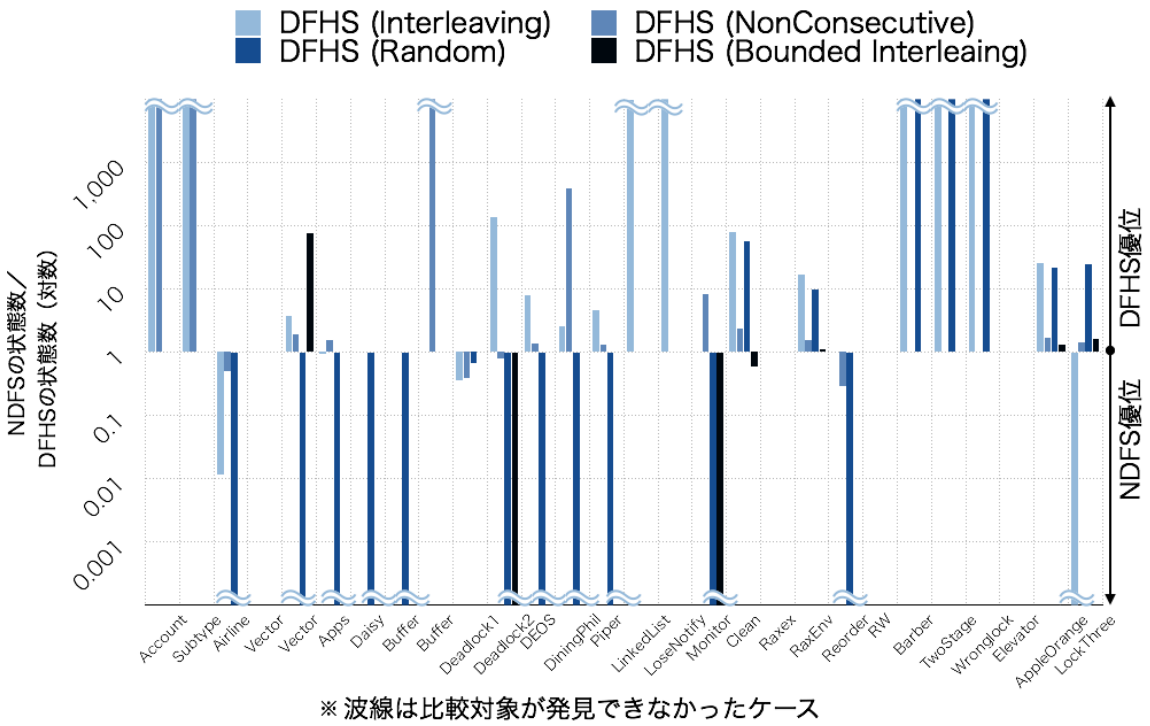

図 6.10: NDFS と DFHS 各枝刈り関数の比較 (枝選択最適化なし)

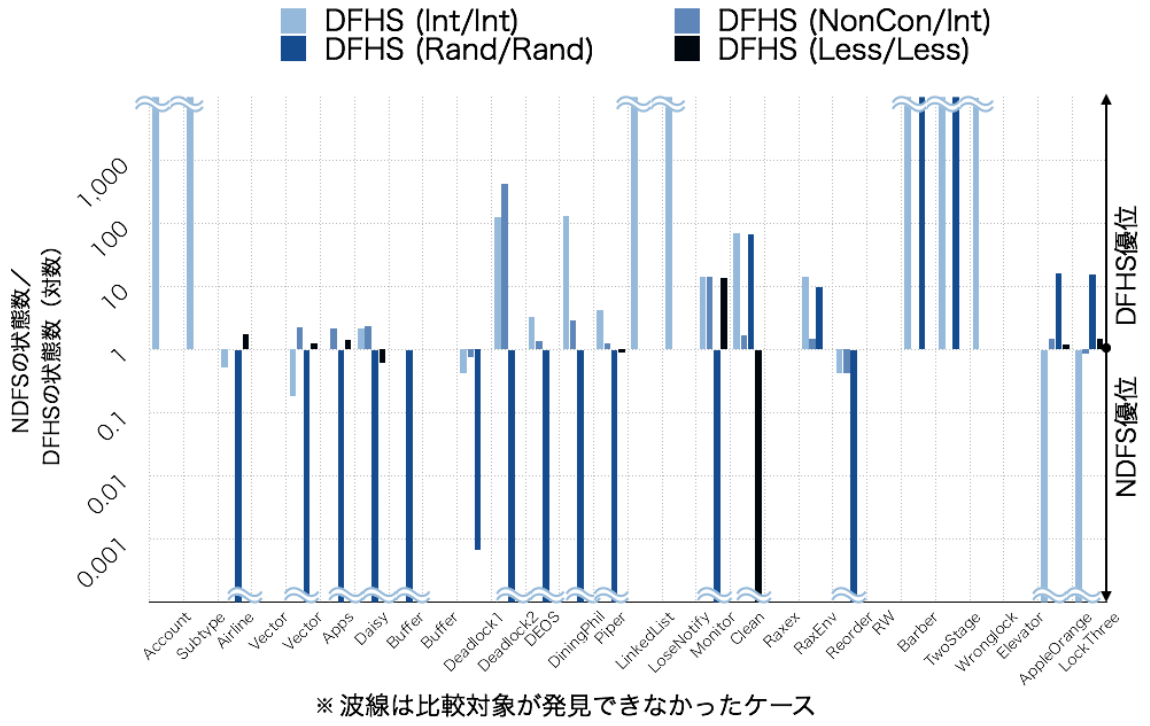

図 6.11: NDFS と DFHS 各枝刈り関数の比較 (枝選択最適化あり)

ラム中 21 のプログラムについて NDFS より良い結果となっている．このうち 7 件については，NDFS ではタイムアウトまたはメモリオーバーフローによって不具合を発見することができなかったケースである．

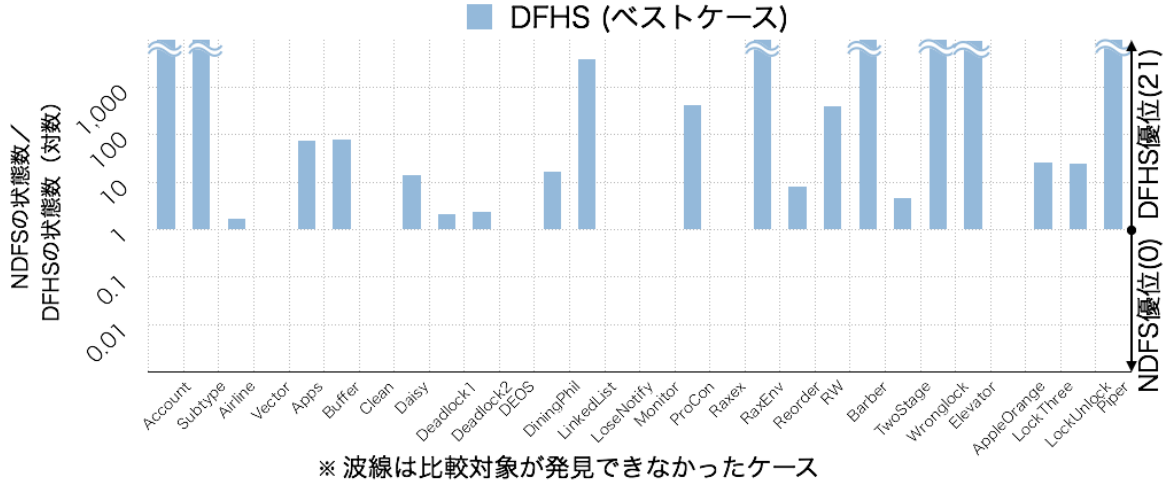

図 6.12: NDFS と DFHS (ベストケース，公平性最適化なし) の比較

最後に，公平性最適化の効果について評価する．図 6.13 に，公平性最適化を適用しない場合を 1 としたときの適用した場合の状態数の比を比較したグラフを示す．この結果から，12 プログラム中 11 プログラムについて優位な結果となった．そのうち 3 件は DFHS の (1) 枝刈り，(2) 枝選択順序最適化のみでは不具合を発見できなかったケースである．

LTL 検証に関しては，LTL 式の複雑さが探索空間（同期積）の大きさに大きく影響する．公平性最適化によって，LTL 式から公平性の記述部分を省くことで，探索空間を削減することか可能となり，その結果，多くのケースで性能が向上したと考えられる．

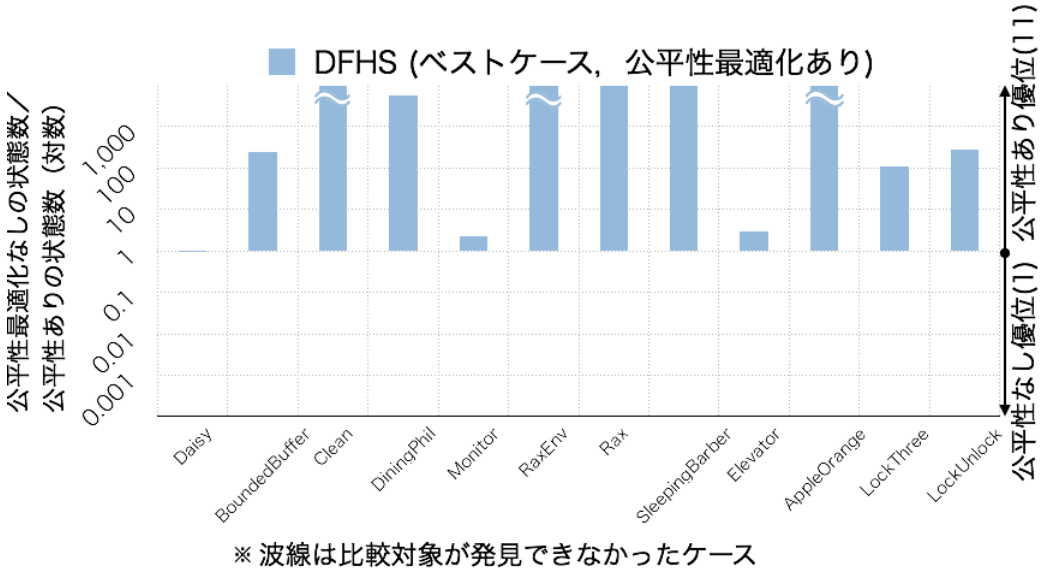

図 6.13: NDFS と DFHS (ベストケース，公平性最適化あり) の比較

6.4 優先度算出関数と枝刈り関数の関係

本節では，従来の最良優先探索における優先度算出関数と DFHS の枝刈り関数の関係について考察する．

6.4.1 最良優先探索による DFHS のエミュレーション

本章では，提案手法である DFHS を，既存研究である最良優先探索と対比させて論じてきた．ここで，従来の最良優先探索の枠組みを用いて DFHS を (ほぼ) 実現することが可能であることに注意しておく．

枝刈り関数を f とすると，次のように定義された最良優先探索手法の優先度算出関数 h を用いて，f をエミュレートすることができる．ただし，f は true または false を返し，true は探索の打ち切りを，false は継続を意味する．また，h は整数を返し，値が小さいほど探索優先度が高いことを意味する．

$$h(S) = \begin{cases} -d(S) & (f(S) = \text{true のとき}) \\ M - d(S) & (f(S) = \text{false のとき}) \end{cases}$$

ここで，$d(S)$ は状態 S の探索空間における深さを現す．また，M は定数であり，考えている状態空間の任意の状態 S に対して $M > d(S)$ が成り立つようにとる．

上記のエミュレーションと，実際の DFHS を比較すると，直接の子供の展開が行われる順に差異がある．エミュレーションでは，各状態の直接の子供が全て展開されてから子供の探索に移るのに対し，DFHS では子供が 1 つずつ選ばれて探索される．したがって，厳密には同一の探索を実施するわけではないが，方針としては同じであり，特にスレッド数が少ない場合には実行効率にも大きな差は認められない．

このことは，しかし，本研究の新規性を否定してはいない．実際，従来研究で提案されているヒューリスティック関数のほとんど [12, 16, 31] は，各状態をある観点から「評価」し，その「評価値」の高い順に探索を実施するために用いられてきた．本研究により，同じ観点 – たとえば「ブロックしているスレッド数」「インターリーブの回数」– を用いても，その「評価の高い順に探索」するのではなく，その「評価が低くなった時点で打ち切る」ことにより，不具合発見が可能になるケースが多数あることが実証された．ヒューリスティック探索の有効性が大きく向上したということができる．

6.4.2 優先度算出関数の枝刈り関数への変換

従来手法の優先度算出関数も，提案手法の枝刈り関数も，ともに与えられた状態から不具合に至る見込みの高さを評価するものである．したがって，優先度算出関数 h をベースに，枝刈り関数 f を作成することができる．

通常は，h が立脚している観点 (たとえば「インターリーブが頻繁に発生する」) を抽出して，f として適切な実装を考えることになるが，機械的に作成することも可能である．例えば，ヒューリスティック値が閾

値 α を超えた場合に探索を打ち切る関数

$$f(S) = \begin{cases} \text{true} & (h(S) > \alpha \text{ のとき}) \\ \text{false} & \text{そうでないとき} \end{cases}$$

あるいは，数回の状態遷移におけるヒューリスティック値の向上が閾値 α より少ない場合に探索を打ち切る関数

$$f(S) = \begin{cases} \text{true} & (h(p(S,n)) - h(S) < \alpha \text{ のとき}) \\ \text{false} & \text{そうでないとき} \end{cases}$$

などが考えられる．ここで，探索空間において状態 S から n 回親をたどって到達する状態を $p(S,n)$ で表している．実験で使用した DFHS の BlockedNum ポリシー関数は，最良優先探索の MostBlocked 関数から，後者の方法で作成したものである．

従来手法に基づくさまざまなヒューリスティック関数が提案されているので，上述の方法で作成した枝刈り関数を適用することによりヒューリスティック探索の多様性が広げられたと言うことができる．

第7章　まとめと今後の課題

7.1　まとめ

本研究では，ソフトウェアモデル検査による不具合発見の効率化を目的とし，新しいヒューリスティック探索手法である DFHS を提案した．提案手法は，既存のヒューリスティック探索による「不具合に到達する可能性が高い状態から優先的に探索する」というアプローチと異なり,「不具合に到達する可能性が低い状態を探索から外す」という枝刈り考え方に基づき探索の効率化を図った．従来手法と提案手法は，表裏の関係にあるため，従来手法で効率化が困難な問題に対して提案手法が有利にはたらく可能性がある．これを検証するために，Java PathFinder (JPF) に対して本手法の実装を行った上で，検証ツール評価用プログラムを用いて，JPF 標準の深さ優先探索および，既存のヒューリスティック探索の一つである最良優先探索との比較実験を実施した．実験の結果，安全性検証，LTL 検証ともに，既存手法よりも多くのケースで DFHS が早期に不具合を発見できることを示し，DFHS による効率化が実現できる可能性が十分に高いことを実証した．

本研究による貢献を以下にまとめる．

安全性検証に対する新しいヒューリスティック探索手法の確立 ヒューリスティック探索はどのヒューリスティック関数がどの不具合に有効であるかを事前に見抜くことは現実的には困難であるため，利用可能な計算資源・時間等に応じ，単一または複数のヒューリスティック関数を選択して適用することになる．計算機資源の制約により，従来手法もしくは DFHS のいずれかを選択して適用するような場合は，各ポリシーの比較結果が示すように，3 件中 2 件のポリシーにおいて DFHS が優位であるため，DFHS によって効率化が実現できる可能性が十分に高い．

一方，計算機資源が潤沢に活用可能で，従来手法，DFHS 含めて複数のヒューリスティックを並列に検査を走らせることが可能であった場合は，実験結果に基づくベストケースの比較結果が示すように 25 のプログラムのうち，17 のプログラムについて，最良優先探索と比較して DFHS が優位な結果を示している．そのため，この場合でも従来技術のみによる並行検査に比べて，効率化が実現できる可能性が十分に高い．近年のクラウド環境など (特に短時間の) 計算資源の入手の容易さを考えるとこのような利用法も十分現実的であるといえる．

以上の効果は，以下の特徴を持つ DFHS という新しいヒューリスティック探索手法を確立することにより実現可能となった．

- 従来のヒューリスティック関数から機械的に枝刈り関数を作ることができる
- 実験結果より，DFHS が従来のヒューリスティック探索より平均的に早く不具合を見つける

LTL 検証へのヒューリスティック探索の導入 従来のヒューリスティック探索は，深さ優先探索をベースとする一般的な LTL 検証アルゴリズムと組み合わせること自体が不可能であった．これに対して，提案手法は，LTL 探索アルゴリズムと組み合わせて利用することができる．28 プログラムを用いて評価実験の

結果より，ヒューリスティック探索による効率化を行わない LTL 検証アルゴリズムとの比較の結果，まず，従来アルゴリズム（1 通り）と各枝刈り関数（8 通り）を比較した場合は，8 件中 6 件の枝刈り関数について，DFHS がより多くのプログラムで不具合を早期に発見できた．また，従来アルゴリズムと DFHS のベストケースとを比較した場合は，28 プログラム中 21 プログラムで不具合を早期に発見した．さらに，DFHS の公平性検証の最適化手法を適用することにより，公平性最適化を実施しない場合と比較して，12 プログラム中 11 プログラムについて，不具合を早期に発見できた．これによって，従来の LTL 検証アルゴリズムに提案手法によるヒューリスティック探索を組み合わせることで，効率化できる可能性が十分に高いことを示した．

実用ツールの提供 提案手法を実際の Java プログラムに対して利用可能な実装を提供したことが貢献である．Java のモデル検査ツールとして広く認識されている JPF に対して，DFHS が実際に利用できる実装を提供したことにより，単に理論として有効であることを示したのみならず，実用への道を開いたといえる．

7.2　今後の課題

今後，本研究を以下のような方向に発展させることを検討している．

各種手法の組み合わせ適用 様々な探索手法を組み合わせたり，複数の手法を複数マシンで並行して実施したりすることで全体最適化を図る．例えば，最良優先探索によってキュー溢れが多数発生した場合に，DFHS に切り替えて再探索することで，両者の利点を活かし，トータルの不具合発見効率の最適化を目指すアプローチが考えられる．

枝刈りポリシーの観点拡大 スレッドの状態に限らず，従来研究のヒューリスティック探索で検討されているその他のヒューリスティックを導入・検証する．

品質保証の観点からの評価 ヒューリスティック探索手法は探索の網羅性を保証しないため，従来のテストにおけるカバレッジのような定量的品質指標の適用を検討する．

謝辞

本研究を進めるにあたっては，大変多くの方々のご指導，ご協力を頂きました．ここに感謝の意を表します．

トップエスイーの修了制作での本研究の立ち上げから今日に至るまで，本研究のご指導を賜りました鶴見大学田辺良則教授には，研究の方向性に関する議論，実験の進め方，論文執筆など，月に数回の打ち合わせを通じて，多大なるご指導・ご助言を頂きました．進捗が思わしくない状況においても熱心にご指導頂きました．ここに深く感謝申し上げます．また，博士課程入学から三年間，主任指導教員としてご指導頂きました電気通信大学石川冬樹客員准教授には，研究の差別化ポイント，研究の拡大などに関するご指導，論文投稿の方向性に関するご助言を頂きました．大変感謝致します．また，指導教員として，研究に関するアドバイス，中間発表，研究審査でのご指導を賜りました電気通信大学大須賀昭彦教授，田中健次教授に深く感謝致します．また，ご多忙にもかかわらず，博士論文の審査員を引き受けて頂き，論文や発表内容に対して多くの有益なご指導・ご指摘を頂きました電気通信大学多田好克教授，田原康之准教授に感謝申し上げます．また，発表練習にご参加頂き，ご指摘，アドバイスを頂きました皆様に感謝致します．

企業での業務と並行して研究を推進するにあたり，株式会社日立製作所の方々には多大なるご支援を頂きました．博士課程に入学する機会を与えて頂くとともに，日頃の業務負荷へのご配慮を賜りました藤井由紀夫氏，藤林昭氏，廣井和重氏，森谷真寿美氏，茂岡知彦氏に感謝致します．また，業務面でご支援を頂きました中村秀樹氏，田中晶氏，村田大二郎氏，是木玄太氏，長井昭祐氏，長野岳彦氏を始めとする職場の皆様に深く御礼申し上げます．

最後に，博士課程の三年間，様々な面で協力し，支えて頂いた家族に感謝致します．

参考文献

[1] Baier, C. and Katoen, J.: *Principles of model checking*, MIT Press (2008).

[2] Ball, T. and Rajamani, S. K.: Automatically Validating Temporal Safety Properties of Interfaces, *8th International SPIN Workshop Toronto, Canada*, Lecture Notes in Computer Science, Vol. 2057, Springer, pp. 103–122 (2001).

[3] Barnat, J., Brim, L. and Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking, *18th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society, pp. 106–115 (2003).

[4] Behrmann, G., David, A. and Larsen, K. G.: A Tutorial on Uppaal, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, Lecture Notes in Computer Science, Vol. 3185, Springer, pp. 200–236 (2004).

[5] Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P. and Yi, W.: UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems, *Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop*, Lecture Notes in Computer Science, Vol. 1066, Springer, pp. 232–243 (1995).

[6] Clarke, E. M., Grumberg, O. and Peled, D.: *Model checking*, MIT Press (2001).

[7] Courcoubetis, C., Vardi, M. Y., Wolper, P. and Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties, *Computer Aided Verification, 2nd International Workshop*, Lecture Notes in Computer Science, Vol. 531, Springer, pp. 233–242 (1990).

[8] Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic, *World Congress on Formal Methods*, Lecture Notes in Computer Science, Vol. 1708, Springer, pp. 253–271 (1999).

[9] Couvreur, J., Duret-Lutz, A. and Poitrenaud, D.: On-the-Fly Emptiness Checks for Generalized Büchi Automata, *Model Checking Software, 12th International SPIN Workshop*, Lecture Notes in Computer Science, Vol. 3639, Springer, pp. 169–184 (2005).

[10] Cuong, N. A. and Cheng, K. S.: Towards Automation of LTL Verifcation for Java Pathfnder, *National University of Singapore* (2008).

[11] Duret-Lutz, A., Poitrenaud, D. and Couvreur, J.-M.: On-the-fly Emptiness Check of Transition-Based Streett Automata, *Automated Technology for Verification and Analysis, 7th International Symposium*, Lecture Notes in Computer Science, Vol. 5799, Springer, pp. 213–227 (2009).

[12] Edelkamp, S., Lluch-Lafuente, A. and Leue, S.: Trail-Directed Model Checking, *Electr. Notes Theor. Comput. Sci.*, Vol. 55, No. 3, pp. 343–356 (2001).

[13] Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A. and Aljazzar, H.: Survey on Directed

Model Checking, *Model Checking and Artificial Intelligence, 5th International Workshop*, Lecture Notes in Computer Science, Vol. 5348, Springer, pp. 65–89 (2008).

[14] Formal Systems (Europe) Ltd.: FDR2.

[15] Geldenhuys, J. and Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm, *Theor. Comput. Sci.*, Vol. 345, No. 1, pp. 60–82 (2005).

[16] Groce, A. and Visser, W.: Heuristics for Model Checking Java Programs, *International Journal on Software Tools for Technology Transfer*, Vol. 6, No. 4, pp. 260–276 (2004).

[17] Hart, P. E., Nilsson, N. J. and Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Trans. Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107 (1968).

[18] Havelund, K. and Pressburger, T.: Model Checking JAVA Programs using JAVA PathFinder, *STTT*, Vol. 2, No. 4, pp. 366–381 (2000).

[19] Henzinger, T. A., Jhala, R., Majumdar, R. and Sutre, G.: Lazy Abstraction, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 58–70 (2002).

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

[21] Khyzha, A., Parízek, P. and Păsăreanu, C. S.: Abstract Pathfinder, *The Java Pathfinder Workshop* (2012).

[22] Lombardi, M.: https://bitbucket.org/michelelombardi/jpf-ltl.

[23] LTSA: Labelled Transition System Analyser, http://www.doc.ic.ac.uk/ltsa/.

[24] Magee, J. and Kramer, J.: *Concurrency - state models and Java programs (2. ed.)*, Wiley (2006).

[25] Manna, Z. and Pnueli, A.: *Temporal verification of reactive systems - safety*, Springer (1995).

[26] McMillan, K. L.: *Symbolic Model Checking*, Kluwer (1993).

[27] Musuvathi, M. and Qadeer, S.: Iterative Context Bounding for Systematic Testing of Multithreaded programs, *SIGPLAN 2007 Conference on Programming Language Design and Implementation*, ACM, pp. 446–455 (2007).

[28] Owen, T.: *Heuristics: Intelligent Search Strategies for Computer Problem Solving* by Judea Pearl Addison-Wesley Publishing Company, Massachusetts, USA, 10 1985 (£43.95), *Robotica*, Vol. 6, No. 2, p. 165 (1988).

[29] Parízek, P. and Lhoták, O.: Randomized Backtracking in State Space Traversal, *18th International SPIN Workshop*, Lecture Notes in Computer Science, Vol. 6823, Springer, pp. 75–89 (2011).

[30] Renault, E., Duret-Lutz, A., Kordon, F. and Poitrenaud, D.: Three SCC-Based Emptiness Checks for Generalized Büchi Automata, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference*, Lecture Notes in Computer Science, Vol. 8312, Springer, pp. 668–682 (2013).

[31] Rungta, N. and Mercer, E. G.: A Meta Heuristic for Effectively Detecting Concurrency Errors, *4th International Haifa Verification Conference*, Lecture Notes in Computer Science, Vol. 5394, Springer, pp. 23–37 (2008).

[32] Rungta, N. and Mercer, E. G.: Clash of the Titans: tools and techniques for hunting bugs in concurrent programs, *7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ACM (2009).

[33] Russell, S. J. and Norvig, P.: *Artificial Intelligence - A Modern Approach (Third Edition)*, Pearson Education

(2010).

[34] Schwoon, S. and Esparza, J.: A Note on On-the-Fly Verification Algorithms, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, Lecture Notes in Computer Science, Vol. 3440, Springer, pp. 174–190 (2005).

[35] Sun, J., Liu, Y., Dong, J. S. and Pang, J.: PAT: Towards Flexible Verification under Fairness, *Computer Aided Verification, 21st International Conference*, Lecture Notes in Computer Science, Vol. 5643, Springer, pp. 709–714 (2009).

[36] Vardi, M. Y. and Wolper, P.: Automata-Theoretic Techniques for Modal Logics of Programs, *J. Comput. Syst. Sci.*, Vol. 32, No. 2, pp. 183–221 (1986).

[37] Visser, W., Havelund, K., Brat, G. P., Park, S. and Lerda, F.: Model Checking Programs, *Autom. Softw. Eng.*, Vol. 10, No. 2, pp. 203–232 (2003).

関連論文の印刷公表の方法及び時期

1. 前岡淳，田辺良則，石川冬樹,
   論文題目「Java PathFinder における探索打ち切りポリシーを用いたヒューリスティック探索」,
   平成 25 年 8 月コンピューターソフトウェア (日本ソフトウェア科学会論文誌)，Vol.30, No.3, pp.109-122
   (本文の 4.1 節，4.2 節，5 章，6 章に関連)
2. Jun Maeoka, Yoshinori Tanabe, Fuyuki Ishikawa,
   "Depth-First Heuristic Search for Software Model Checking",
   2015/6 IEEE/ACIS International Conference on Computer and Information Science (ICIS 2015),
   Studies in Computational Intelligence (Springer)
   (4.3 節，5 章，6 章に関連)