修 士 論 文

リアルタイムJava検証のための検証支援環境に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

曽我 徹典

2007年3月

修 士 論 文

リアルタイムJava検証のための検証支援環境に関する研究

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　DEFAGO Xavier 助教授
審査委員　青木利晃 特任助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

510057 曽我 徹典

提出年月: 2007 年 2 月

概 要

本稿は, リアルタイム Java に対して, 検証方法とそのための検証支援環境を提案する.

組込みソフトウェアにおいて, リアルタイム性の保証と資源の制約は高信頼性を持つために遵守されるべきである. しかし, 開発期間が変わらないまま, 肥大化していくソフトウェアに開発コストと開発者の負担は大きくなっている. 本研究では, 将来的に組込みソフトウェア開発に使用されるであろうプログラム言語リアルタイム Java に着目した. そして, リアルタイム Java の不具合を検証するため, 検証方法の提案とそのための検証支援環境の一部を実験的に実装し, リアルタイム Java を例題として実験した. その結果, 検証に必要な状態遷移モデルを作り出すための要素を作りだすことができた.

目 次

第1章　はじめに

本研究をはじめる動機となった背景と目的を述べる.

1.1　背景

本研究は, 組込みソフトウェア開発の実行基盤となるリアルタイム Java に対して, プログラムの不具合を発見するための検証支援環境を提案する.

組込みソフトウェアにおいて, リアルタイム性の保証と資源の制約は高信頼性を持つために遵守されるべきである. しかし, 開発期間が変わらないまま, 肥大化していくソフトウェアに開発コストと開発者の負担は大きくなってきており, ソフトウェアの品質に影響を与えている.

近年, このようなソフトウェアの肥大化への対抗手段について, 議論が盛んである. 現在, このような問題を解決するために形式的手法が注目されている. その中でもモデル検査は必要な予備知識が比較的少なく, 開発現場での適応が期待されている. また, リアルタイム性の保証と資源の制約は, プログラマが注意深く明示的にプログラムを記述するがある. この作業は, 職人芸的な要素が強く, やはり開発コストとソフトウェアの品質の両面で大きな問題となっている. これらの作業を最大限に自動化し, プログラマによる指定を最低限に抑えるための実行基盤を用いることも対抗手段の 1 つである.

1.2　目的

本研究の目的は, 組込みソフトウェアに対するそのような取り組みをふまえ, 組込みソフトウェアの開発に適応することができるための検証支援環境を提案し, その一部分を実験によって評価することである. 本研究で提案する検証支援環境は, 組込みソフトウェアがリアルタイム Java で開発されることを考えられており, 作られたソフトウェアを開発者が大きな負担をかけることなく検証をおこなうことができるためのツールである.

1.3　アプローチ

本研究は, リアルタイム Java を検証するための検証支援環境の全体像を考え, 最初の操作となるリアルタイム Java から検証に必要な状態遷移モデルを作りだすためのプログラ

ムストラクチャを作りだすことができるかを確かめることである. ここでいうプログラムストラクチャとは, クラスファイルを解析してえられるバイトコード命令のうち, 逐次実行をおこなう部分を区間ごとに分けた木構造である. この構造は, 逐次実行される部分ごとにバイトコード命令が分割されている. これは, リアルタイム Java の実行環境での 1 つ 1 つのバイトコード命令の実行時間を計ることで, プログラムストラクチャに時間を割り当てることができるようにするためである.

検証支援環境を用いた検証の全体の流れを説明する.

リアルタイム Java のクラスファイルを入力として, バイトコード処理ライブラリ BCEL を用いてプログラムストラクチャに変換する. リアルタイム Java の実行環境での 1 つ 1 つのバイトコード命令の実行時間を計った実行時間表を作成する. プログラムストラクチャと実行時間表を合わせることでプログラムストラクチャに時間を割り当てる. 時間付きのプログラムストラクチャからパラメトリックタイムストラクチャを作る. パラメトリックタイムストラクチャとは, 本研究で検証に用いるパラメトリックモデル検査の状態遷移モデルである. パラメトリックモデル検査は, 時間を定数として決めておきたくない部分を変数としておき, 検証した結果として時間の不等式を返す, モデル検査手法である. マルチスレッドのプログラムの場合, それぞれのスレッドでパラメトリックタイムストラクチャが作られるので, これを合成する. ここまでの操作でできたパラメトリックタイムストラクチャに, 検証性質を書いて, パラメトリックモデル検査ツールで検証をおこなう. パラメトリックモデル検査での検証によって時間の不等式をえられる. その不等式から要求に合う適切な値を決めてプログラムを修正する.

このような検証支援環境ならば, 開発者は設計・分析の段階を変更することなく検証をおこなうことができるようになる.

1.4 結果

本研究では, 提案した検証支援環境のうち, リアルタイム Java のクラスファイルからプログラムストラクチャを作りだすツールを実験的に実装した.

ツールは, リアルタイム Java に対して部分的な解析しかできなかったため, 本研究では通常の Java のクラスファイルを解析の対象とした. 実験の結果として, このツールによって, BCEL を使用して Java のクラスファイルを解析して, プログラムストラクチャを作りだすことができる, とわかった. これは, この実験に限って, 検証支援環境を作るための最初の操作をおこなうことができるという意味である. また, プログラムストラクチャを作る上で, 問題となる Native メソッドや Abstract メソッドなどを発見することができた. これらの問題点については解決策を考察した. 本研究で作りだしたプログラムストラクチャは, パラメトリックタイムストラクチャを作りだすためには不完全である. このため, 検証をおこなうためには, パラメトリックタイムストラクチャを作りだすことのできるプログラムストラクチャを作りだす必要がある.

第2章　リアルタイムJava

リアルタイムシステムは, システムが定められた時間制約を満たして動作する性質を持つ. リアルタイム Java とは, そのようなリアルタイムシステムの要件であるリアルタイム性を記述することのできる Java である. リアルタイム性の記述は, RTSJ によって定められている. RTSJ のうち, 本研究に関係する主な仕様について説明する. また, 本研究で用いたリアルタイム Java の実行環境についても述べる.

2.1　Javaリアルタイム仕様RTSJの概要

RTSJ は, ハードリアルタイムアプリケーションもソフトリアルタイムタイムアプリケーションもサポートするように設計されている.

RTSJ の指針原則として以下が定められている.

- 特定の Java 環境に対する適用の可能性：

  Java 開発キット, 組込み Java アプリケーション環境, あるいは $Java^{TM}2$ プラットフォームマイクロエディッション ($J2ME^{TM}$) の特定のバージョンなどに, RTSJ の使用を制限するような仕様を含んではならない.

- 互換性：

  RTSJ の実装上で, 既存の適切に書かれたリアルタイムではない Java プログラムの動作を, RTSJ は阻害してはならない.

- Write Once, Run Anywhere：

  RTSJ は, 「Write Once, Run Anywhere」の重要性を認識しなければならない. しかし, リアルタイムプログラムで WORA を達成することの困難さも認識しなければならないし, 予測可能性を犠牲にしてまでバイナリ移植性を高めたり維持したりするように試みてはならない.

- 現状の実践と先進の機能：

  RTSJ は, 将来の実装が先進の機能を含めることを可能とすると同時に, 現在のリアルタイムシステムで実践されていることを扱えなければならない.

- 予測可能な実行：

  RTSJ はすべてのトレードオフにおいて, 予測可能な実行を保持することを最優先としなければならない. これにより, 典型的な汎用コンピューティングパフォーマンスが犠牲になる場合があるかもしれない.

- 文法の拡張をしない：

  ツール開発者の作業を容易にする. その結果, 早い段階で実装が提供される可能性を増すために, RTSJ は Java 言語に対して新たな予約後を導入, つまり文法の拡張をしてはならない.

- 実装上の決定における差を許す：

  多くの実装上の決定に関して, RTSJ の実装は異なる. 効率あるいは非効率なアルゴリズムの仕様, 時間と空間の効率でのトレードオフ, 最適源の実装に必要とされていないスケジューリングアルゴリズムを含んだり, バイトコードを実行するためのコードパス長の違いなどがある. RTSJ は, そのようなアルゴリズムや特定の時間定数を必須とすべきではない. しかし, 実装のセマンティクスが満たされることを要求する. RTSJ は顧客の要求を満たすのに適した実装を作成するための柔軟性を実装者に提供する.

本研究において重要な仕様については, 次に詳細を述べていく.

2.2 優先度に基づくスケジューリング

タスクをスイッチするオペレーティングシステムのある機能は, ポリシとメカニズムをうまく切り分けている. あるタスクの状態を実際にセーブし, 次のタスクをスタートさせるコンポーネントはディスパッチャと呼ばれる. 次にどのタスクを走らせるかを決定するコンポーネントをスケジューラという.

ディスパッチャは伝統的に簡単で高速である. システムが急に何かを先にしなければならないとする. そのときには, 緊急の仕事をするタスクにスイッチする. ディスパッチャで使われる時間は, 純粋なオーバヘッドのようにみえる. リアルタイムシステムのソフトウェアは, このオーバヘッドをゼロに近づける努力をおこなう. ディスパッチャそのものには知的な機能はない. リアルタイムシステムを作るプログラマは, ディスパッチャは速く, そして単純であることを望む.

ところが, スケジューラはそうではない. リアルタイムではないシステムは, FIFO の順番で CPU 時間をタスクに配るスケジューラに満足している. しかし, ほとんどのリアルタイムシステム固定優先度に基づくスケジューラを使用する. RTSJ でも, 固定優先度でプリエンティブなスケジューリングが最小の要求事項である. 単純な固定優先度に基づくスケジューラは, 最高の優先度を持つ, 実行可能なタスクを最優先に走らせることができる

ことを保証する. 優先度は, リアルタイムシステムにおける「重要度」の目安を表す. これは, 与えられた時間でどのタスクが一番重要であるか, ということを表す. リアルタイムシステムでもう 1 つ重要なスケジューリングがある. それはデッドラインに基づくスケジューリングである. これは, それぞれのタスクの優先度の代わりに, 現在のタスクを完了するのに必要な時間に従ってスケジューリングをおこなう.

2.2.1 プリエンプティション

システムが並行タスクごとに, 少なくとも 1 つのプロセッサを持っていない限り, オペレーティングシステムはときどきプリエンプトして, ほかのタスクがプロセッサをアクセスできるようにする. あるオペレーティングシステムは, ある区間はプリエンプティションを遅らせる機能を持っている. リアルタイムのオペレーティングシステムは, プリエンプティションによる遅延とプリエンプティションサービス時間が問題となる. プリエンプティションによる遅延は, プリエンプティションが遅れる最大の時間である. プリエンプティションサービス時間は, プリエンプティションがはじまり, プリエンプティションが終わるまでの時間である.

リアルタイムの目的を持っているどのようなシステムでも, 実行可能な優先度が最大のタスクを実行するために, 現在走っているタスクをプリエンプティションする. たとえば, タスクのスイッチを持っているタスクは, タスクのスイッチが行われると, 直ちに実行される. ただし, 優先度が同じであるか, または優先度がそれより高いタスクがすでに走っていない場合である. これは優先度の定義ではない. つまり, スケジューラが, プリエンプティブであるか, またはサポートしていない場合を考える. このとき, タスクがブロックすることによってプリエンプティションされてもよいと申しでれば, 次に走るタスクは優先度で選ぶ以外に方法はないと考えられる.

2.2.2 優先度逆転

固定優先度のスケジューラは, プログラマによって割り付けられた優先度にしたがって厳密にスケジュールをおこなう. これに対するものが動的優先度に基づくスケジューラである. これはスケジューラがタスクの優先度を変更することを許す. 厳密な固定優先度のスケジューラというのは, 使われなくなってきている. その理由は, 固定優先度のスケジューラは優先度逆転を引き起こすからである. その結果, 固定優先度は「優先度逆転以外は優先度が固定」であるということを意味するようになった.

図 2.1 は, 優先度逆転の例である. タスクは 3 つあり, 優先度が高い順にタスク A, B, C としている. 資源は, X としている. この操作は以下の通りである.

1. タスク C が, 資源 X にロックをかける.
2. タスク B が, C にプリエンプトして長時間計算する.

3. タスク A が, B をプリエンプトして, X にロックをかけようとする. しかし, ロックをかけることはできない. すでにタスク C が資源 X にロックをかけているからである.

タスク A は最高の優先度を持っているが, タスク C が資源 X を解放するまで先に進むことはできない. 一方, タスク C は, B が A にプロセッサ時間を渡すまで資源を解放できない. この操作では, タスク A の優先度は, タスク C と同じになってしまう.

この状況が優先度逆転である. 優先度逆転は, 優先度の高いタスクが優先度の低いタスクよりも, あたかも優先度が低いようにスケジュールされることから, このようにいわれる.

優先度逆転は, 注意深いプログラマによって制御することができる. しかし, そのようなことが起こることが予測されているか, またはそのように設計されている場合だけである. つまり, それ以外では, 優先度逆転は見逃されることになる. そして, 配備されたシステムでは, 発見はきわめて困難となる. これは, 図 2.1 のように, 優先度の低いタスクが資源を確保しているとき, 優先度の高いタスクがその資源を必要とする, というような特別なイベントが続いたときに起こる. この状況は, 起こりそうもないイベントの組み合わせを考えなくてはならない. これは, テストを行っているときは眠っていて, システムとして配備されたときに断続的に起こるバグとして現れる.

このような事態を避けるために, 優先度継承プロトコルを実装する方法がある.

ロックをかけようと待っているタスクが, ロックをかけているタスクより優先度が高いとする. すると, 優先度継承プロトコルを実装するロックは, 待機中のタスクの優先度をロックをかけているタスクに割りあてる. そして, タスクが資源を解放したときに, 元の優先度に戻す.

図 2.2 は, 優先度継承の例である. タスクと資源に関しては図 5.1 と同じである. この操作は以下の通りである.

1. タスク C が, 資源 X にロックをかける.
2. タスク B が, C にプリエンプトして長時間計算する.
3. タスク A が, B をプリエンプトして, X にロックをかけようとする.
4. タスク C に, A の優先度を与える. この操作でタスク A が, C が X をアンロックしたときに, X をロックできるようにする. よって, タスク A は, B で待たされることなくロックをかけて, アンロックする間にプログラムを実行することができる.

いくつかの研究によると, 合理的な共通な仮定のもとでは, レイトモノトニック優先順位割当てアルゴリズムが使用される場合には, 32 個のユニークな優先順位レベルは, 最適に近いスケジューリング効率に対する妥当な選択であることを示している. また, 256 個の優先順位レベルは, より高い効率性を提供する. この仕様の実装は, Java 仮想マシンの外で実行されるロジックを持つシステム上に存在する. そして, そのシステムの動作のためにより高い, より低い, あるいは両方の優先順位を必要とするかもしれません. その妥協として, この RTSJ では最低 28 個のユニークな優先順位を要求している.

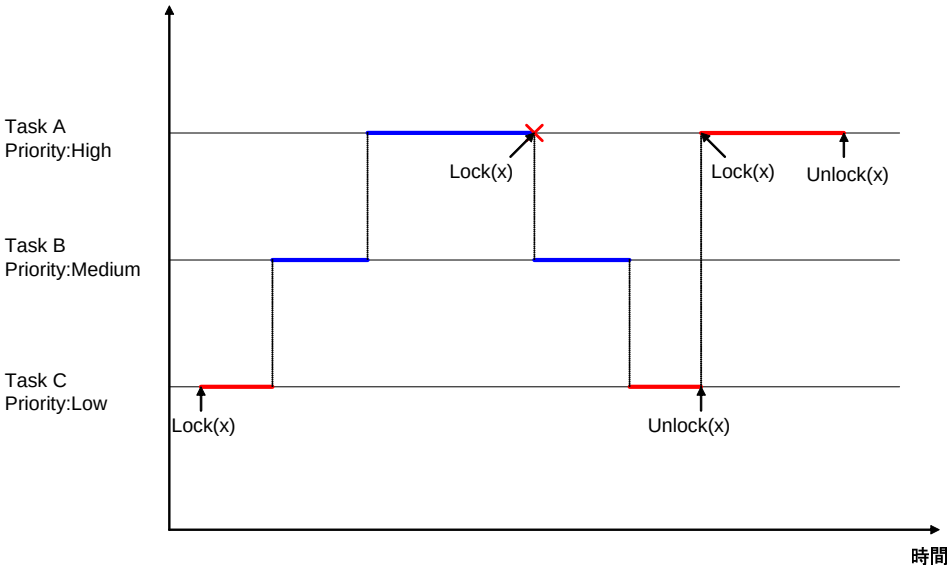


図 2.1: 優勢度逆転の例

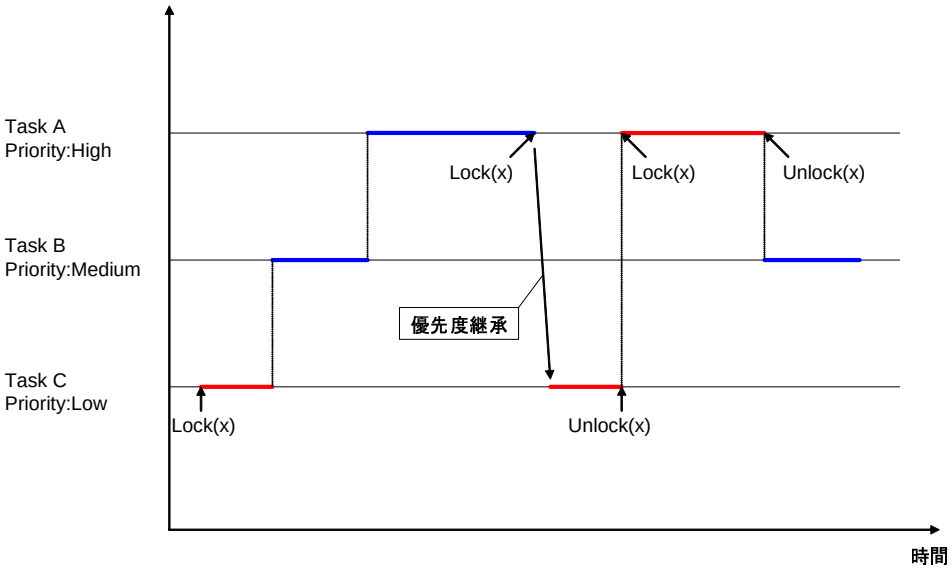


図 2.2: 優勢度継承の例

2.3 リファレンス実装

現在, 通常の Java 環境ではリアルタイム Java を実行することはできない. 本研究では無償で利用することのできる TimeSys 社の RTSJ リファレンス実装を用いた. ほかにも有料であるが, Sun MicroSystems 社の Java RTS, IBM 社の Apogee の Aphelion と WebSphere Real Time がある.

2.3.1 TimeSys RTSJ リファレンス実装

実行環境として x86 系 CPU と LinuxOS が必要である. LinuxOS は RedHat9 が推奨される. 本研究で用いた GPL 版は無償であるため機能制限が設けられている. たとえば, 優先度継承プロトコルが実装されていない.

第3章　パラメトリックモデル検査

パラメトリックモデル検査は, モデル検査の一種である. パラメトリックモデル検査は, 状態遷移モデルと検証性質から, 検証性質を満たす条件式を導出する. 本研究では, 検証支援環境に Chaiwat の考案したパラメトリックモデル検査を使用することを想定している. 以後のパラメトリックモデル検査については, Chaiwat の考案したパラメトリックモデル検査のことである.

3.1　パラメトリックモデル検査の概要

モデル検査は状態遷移モデルと検証性質をモデル検査ツールの入力として与える. これで状態遷移モデルが, 検証性質を満たすかどうかを検証する. 検証性質を満たさない場合, その反例を結果として返す.

パラメトリックモデル検査でも同様に, 状態遷移モデルと検証性質をモデル検査ツールに入力する. パラメトリックモデル検査は時間性質について扱うことができる. パラメトリックモデル検査では, 状態から状態の遷移間に定数と変数を割り当てることができる. 遷移間を変数で割り当てた箇所は, 検証結果として変数が取りうる値を条件式で返す. そして, 条件式に合う値を選ぶことで, 検証性質を満たすことができる. 検証性質を満たす値をモデル検査のあとに決める. この方法ならば, 条件式を満たす範囲の値を取ることができるならば, 状態遷移モデルの修正をおこなう必要がない.

第4章　検証全体の流れ

リアルタイム Java からパラメトリックモデル検査による検証までを 3 つの操作に区分けして説明をする. 検証の全体の流れは図 4.1 の通りである.

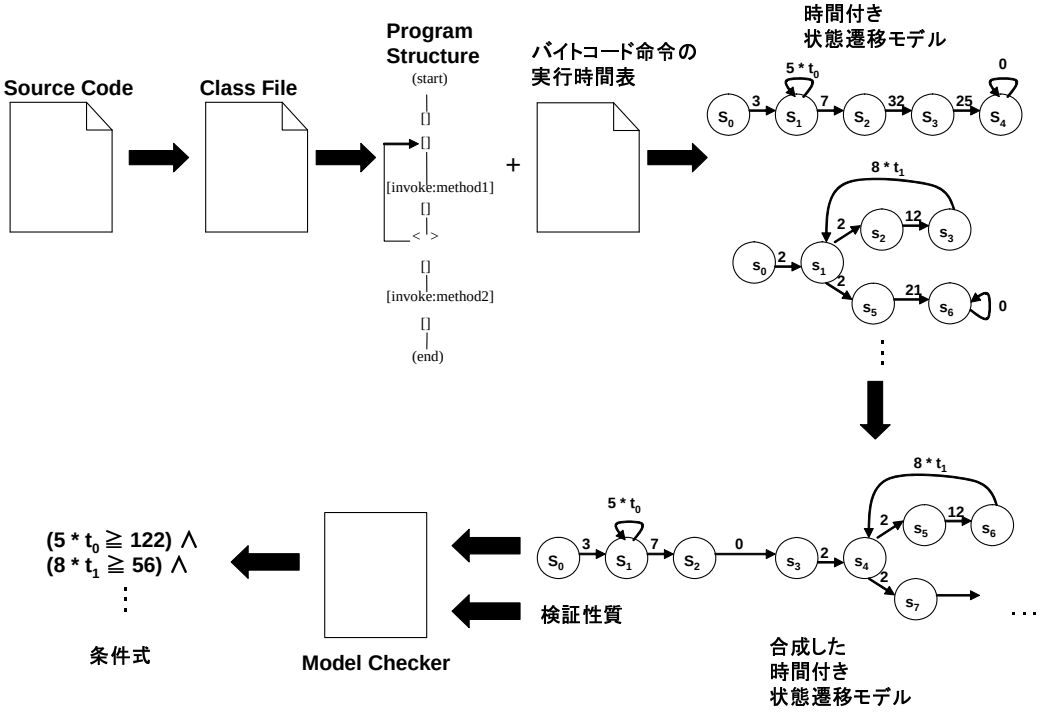


図 4.1: 検証支援環境を用いた検証の流れ

4.1　リアルタイム Java からプログラムストラクチャを作成する

本研究が検証をおこなうのは, リアルタイム Java である. そのために, リアルタイム Java をコンパイルして作られる, クラスファイルが検証支援環境の入力に必要となる. クラスファイルは, バイトコードと呼ばれる機種に依存しない中間言語で構成されている.

プログラムの実行は, Java 仮想マシンの中のインタプリタによってネイティブコードに変換されてからおこなわれる. 本研究では, クラスファイルを解析してえられるバイトコードを元にしてプログラムストラクチャを作りだしている.

プログラムストラクチャとは, クラスファイルを解析してえられるバイトコード命令のうち, 逐次実行をおこなう部分を区間ごとに分けた木構造である. この構造は, 逐次実行される部分ごとにバイトコード命令が分割されている. これは, リアルタイム Java の実行環境での 1 つ 1 つのバイトコード命令の実行時間を計ることで, プログラムストラクチャに時間を割り当てることができるようにするためである. これは, プログラムストラクチャを作ることで, リアルタイム Java の実行環境が異なっていても, それぞれの実行環境での実行時間を調べることでプログラムストラクチャに時間を割り当てることができるからである. 図 4.2 がプログラムストラクチャへの時間の割り当て方になる.

本研究では, タスクのデッドラインを検証性質と想定しているため, 時間情報は, プログラムストラクチャに割り当てることになる.

4.2 プログラムストラクチャからパラメトリックタイムストラクチャを作成する

本研究で検証に用いることを想定しているモデル検査は, パラメトリックモデル検査である. パラメトリックモデル検査は, 入力にパラメトリックタイムストラクチャを用いる. そのために, パラメトリックタイムストラクチャを作る操作をおこなう.

リアルタイム Java の実行環境でのバイトコード命令の実行時間を調べて, バイトコード命令の実行時間表をつくる. この実行時間表を使ってプログラムストラクチャに時間を割り当てていく. そして, 時間が割り当てられたプログラムストラクチャからパラメトリックタイムストラクチャを作りだす. この時点でのパラメトリックタイムストラクチャは, リアルタイム Java のスレッドごとに作りだされることを想定している.

本研究において, 具体的な実行環境でのバイトコード命令の実行時間の求めかたについては取り扱わない. また, 時間を割り当てられたプログラムストラクチャからパラメトリックタイムストラクチャを作りだす操作についても同様である.

4.3 パラメトリックタイムストラクチャから検証をおこなう

ここまでの操作で, スレッドごとのパラメトリックタイムストラクチャが作られている. 次は, スレッドごとのパラメトリックストラクチャを合成する. このとき, スケジューリングについて考慮する必要があると考える.

合成されたパラメトリックタイムストラクチャとパラメトリック CTL で書いた検証性質をモデル検査器にかけて検証をおこなう. 検証の結果は, 不等式として返ってくる. この不等式から開発者が要求に合う適切な値を変数に対してきめる. そして, 適切な値をプ

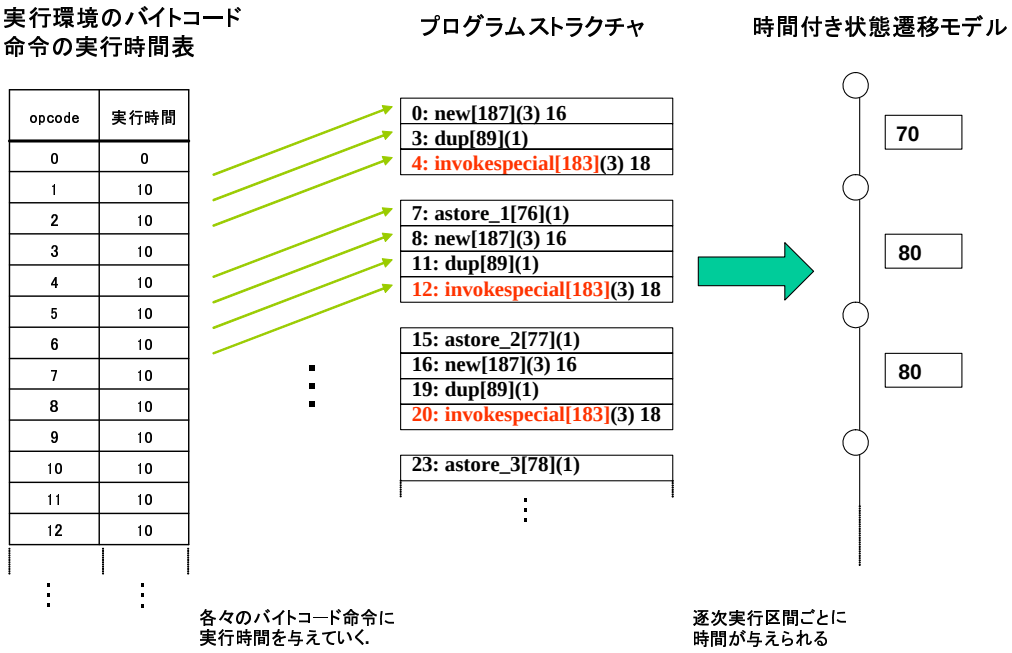


図 4.2: プログラムストラクチャへの時間の割り当て方

ログラムに反映させることで, 不具合と性能の向上を図ることができるようになる.

本研究において, 具体的にパラメトリックタイムストラクチャを合成する方法については取り扱わない.

第5章　クラスファイルの解析によるプログラムストラクチャの作成方法

4 章で提案した検証支援環境のうち, 最初の操作である, リアルタイム Java からプログラムストラクチャを作りだす方法について具体的に提案する.

本研究では, クラスファイルの解析にバイトコード処理ライブラリ BCEL を使用する. そして, クラスファイルを解析してえられるバイトコード命令からプログラムストラクチャを作りだすために注目すべき点について述べる.

5.1　バイトコード処理ライブラリ BCEL

BCEL は, バイナリ形式のクラスファイルを解析, 生成, 操作を可能とすることを目的としているライブラリである. 本研究においては, プログラムストラクチャを作りだすために, クラスファイルを分析するために BCEL を利用している.

5.2　プログラムストラクチャを作る手順

図 5.1 がクラスファイルを解析するための流れとなっている. 詳細は以下に述べていく.

はじめに, 1 つのクラスファイルを入力する. BCEL には, 与えられたクラスファイルを解析するための ClassParser というクラスがある. この ClassParser にクラスファイルを入力することで, クラスファイルの解析の基本型となる JavaClass というクラスのオブジェクトを返す. この JavaClass のオブジェクトは, 基本的に, フィールド, メソッド, スーパクラスや実装されたインタフェースへのシンボリックリファレンスで構成される. 以上の操作を以下にコードとして示す.

```
ClassParser clazzParser_ = new ClassParser(args[0]);
JavaClass javaClazz_ = clazzParser_.parse();
```

次に, ClassGen を JavaClass で初期化して, ClassGen のオブジェクトを生成する. ClassGen は, コンスタントプールを取得するために生成した. コンスタントプールとは, クラスファイル内で参照される様々な文字定数, クラス名やインタフェース名, フィールド名,

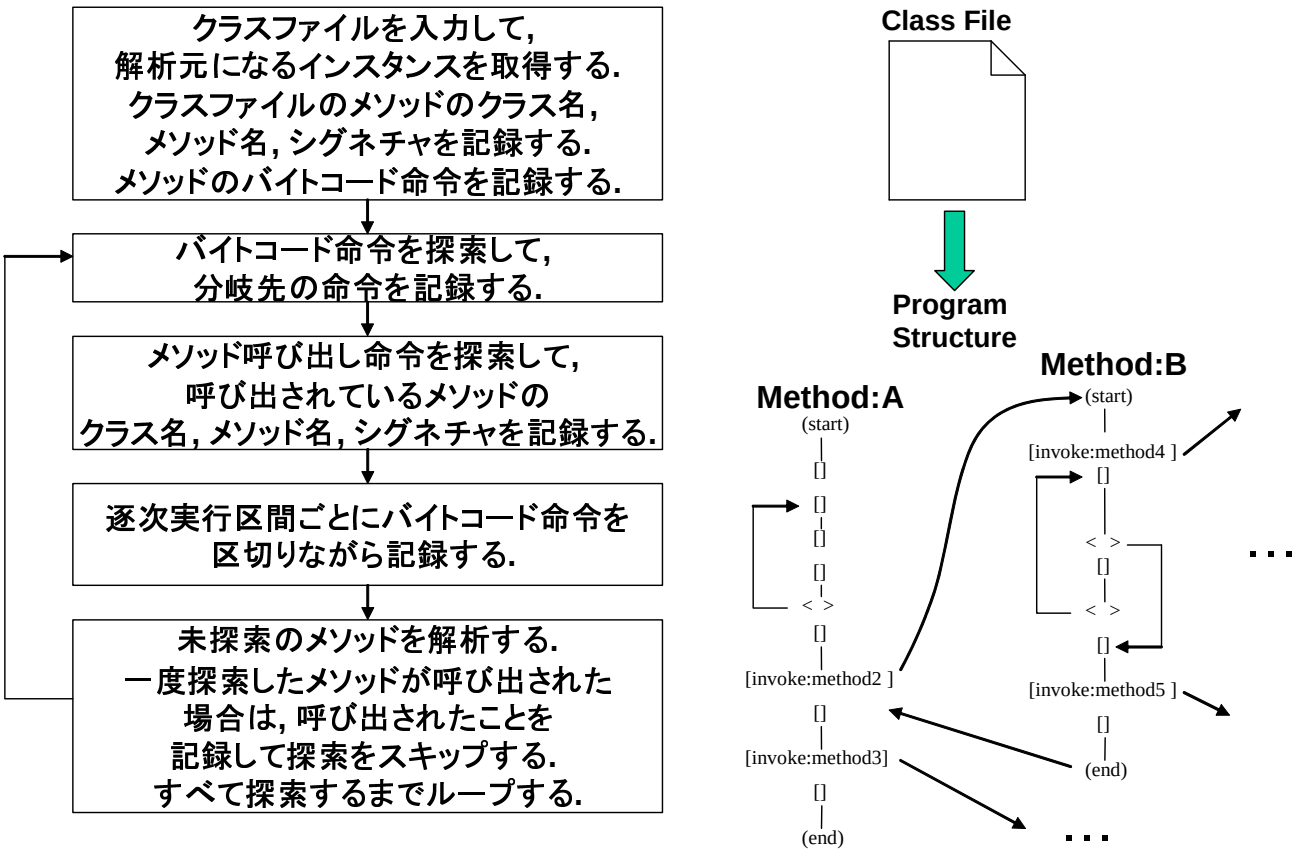


図 5.1: プログラムストラクチャをつくる流れ

その他の定数を表現するための構造体テーブルである. また, JavaClass のオブジェクトから, 入力されたクラスファイルのメソッドの情報を取得する. 以上の操作を以下にコードとして示す.

```
ClassGen clazzGen_ = new ClassGen(javaClazz_);
ConstantPoolGen constantPoolGen_ = clazzGen_.getConstantPool();
Method[] methods_ = javaClazz_.getMethods();
```

Method から入力したクラスファイルのクラス名, メソッド名, シグネチャを取得する. ここでのシグネチャは, メソッド名を含まず, 仮パラメータの数と型のみを指す. メソッドのクラス名, メソッド名, シグネチャの 3 つの情報から, メソッドを一意に決定することができる. クラス名, メソッド名, シグネチャのそれぞれの情報を格納するために, ArrayList を使用した. InstructionList を取得するために MethodGen を生成している. InstructionList は, メソッドの操作を表す命令ハンドラのセットからできている. 命令ハンドラは InstructionHandle というクラスで定義されている. 命令ハンドラは, 特に断らない限りは, バイトコード命令を意味する. 以上の操作を以下にコードとして示す.

```
MethodGen[] methodGens_ = new MethodGen[methods_.length];
ArrayList<String> classNameList_ = new ArrayList<String>();
ArrayList<String> methodNameList_ = new ArrayList<String>();
ArrayList<String> signatureList_ = new ArrayList<String>();
for (int i = 0; i < methods_.length; i++) {
  //メソッドを特定するための 3 つの情報をそれぞれの ArrayList に格納.
  //入力するクラスファイルのみ, Method から取得する.
  classNameList_.add((methods_[i].getName()));
  methodNameList_.add((methods_[i].toString()));
  signatureList_.add((methods_[i].getSignature()));
  methodGens_[i] = new MethodGen(methods_[i], methods_[i].getName(), constantPool-
Gen_);
}
```

生成した MethodGen から, InstructionList を取得する. InstructionList は, 個々の命令を扱うことができないので, 個々の命令ハンドラごとに ArrayList に格納して, 命令ハンドラのリストをつくる. 命令ハンドラのリストは 1 つのメソッドに相当する. クラスファイルのすべてのメソッドは, ArrayList で実現した. 2 次元配列である命令ハンドラのテーブルに格納する.

命令ハンドラのリストを取得する前に, MethodGen が Abstract メソッド, Native メソッドまたはそうでないメソッドであるかを判定する. 現在は判定の結果として, Abstract メ

ソッドと Native メソッドには, メソッドを特定する情報を持つ, 空の命令ハンドラのリストを格納する. また, 命令ハンドラのリストに Abstract メソッドと Native メソッドを認識するための特別な値を設定する. メソッドを特定する情報は, メソッドのクラス名, メソッド名, シグネチャの 3 つの情報を連結してつくる.

メソッドがバイトコード命令を持つ場合は, MethodGen から InstructionList を取得して, 個々の命令ハンドラを命令ハンドラのリストに格納する. 命令ハンドラのリストにメソッドのすべての命令ハンドラを格納したら, 命令ハンドラのテーブルに命令ハンドラのリストを格納する.

また, 1 度呼び出したメソッドを再度探索しないようにするために, Hashtable で実現した探索表にメソッドの記録をする. Key にメソッドを特定する情報を使用し, 命令ハンドラのリストの参照を value とした. 再度呼び出された命令の場合, 命令ハンドラのリストの参照を命令ハンドラテーブルに格納する. 以上の操作を以下にコードとして示す.

```
InstructionList[] instructionLists_ = new InstructionList[methods_.length];
InstructionHandleTable instructionHandleTable_ = new InstructionHandleTable();
Hashtable<String, InstructionHandleList> recursivelyInvokedMethodHashtable_ = new
Hashtable<String, InstructionHandleList>();
for (int j = 0; j < instructionLists_.length; j++) {
  // methodIdentify_は, メソッドごとに渡す.
  methodIdentify_ = classNameList_.get(j)+methodNameList_.get(j)+signatureList_.get(j);
  // メソッドの Abstract Method の判定と InstructionHandleTable への格納.
  if (methodGens_[j].isAbstract()) {
    instructionHandleTable_.setAbstractMethod(j, methodIdentify_, recursivelyInvoked-
MethodHashtable_);
  }
  // メソッドの Native Method の判定と InstructionHandleTable への格納.
  else if (methodGens_[j].isNative()) {
    instructionHandleTable_.setNativeMethod(j, methodIdentify_, recursivelyInvokedMethod-
Hashtable_);
  }
  // メソッドの InstructionHandleTable への格納.
  else {
    instructionHandleTable_.setMethod(j, methodIdentify_, recursivelyInvokedMethod-
Hashtable_, methodGens_[j]);
  }
}

// setAbstractMethod
```

```
public void setAbstractMethod(int index, String metID, Hashtable<String, InstructionHandleList>
reInvMetHashtable) {
  InstructionHandleList insHanList_ = new InstructionHandleList(metID);
    insHanList_.setAbstractMethodCheckingNumber();
    insHanTable_.add(insHanList_);
    reInvMetHashtable.put(metID, insHanTable_.get(index));
  }

//setNativeMethod
  public void setNativeMethod(int index, String metID, Hashtable<String, InstructionHandleList>
reInvMetHashtable) {
    InstructionHandleList insHanList_ = new InstructionHandleList(metID);
    insHanList_.setNativeMethodCheckingNumber();
    insHanTable_.add(insHanList_);
    reInvMetHashtable.put(metID, insHanTable_.get(index));
  }

// setMethod(命令ハンドラのテーブル用)
  public void setMethod(int index, String metID, Hashtable<String, InstructionHandleList>
reInvMetHashtable, MethodGen metGen) {
    InstructionList insList_ = metGen.getInstructionList();
    InstructionHandleList insHanList_ = new InstructionHandleList(metID);
    insHanList_.setInstructionHandles(insList_.getInstructionHandles());
    insHanTable_.add(insHanList_);
    reInvMetHashtable.put(metID, insHanTable_.get(index));
  }
```

命令ハンドラのテーブルに, クラスファイルの命令ハンドラのリストを入れ終わったら, 命令ハンドラのテーブルから分岐先の命令を探索する. 分岐先の命令は Hashtable に格納する. Key は, 探索しているメソッドのメソッドを特定する情報と分岐先の命令ハンドラのアドレスを連結した値を使用した. value は, 分岐先の命令である.

分岐先の命令は, バイトコード命令のうち逐次実行をおこなう部分を区間に分けるために使用する. 区間に分けるとは, 分岐先の命令をバイトコード命令の中に見つけた場合, バイトコード命令を部分的なバイトコード命令として記録するということである. 分岐先の命令以外にも, 条件分岐命令, 無条件分岐, 複合条件分岐, メソッド呼び出し命令, リターン命令が逐次実行区間に分ける命令である. 表 5.1 は, 分岐命令先以外のバイトコード命令を逐次実行区間に分ける命令である.

また, 命令ハンドラのテーブルからメソッド呼び出し命令も探索する. メソッド呼び出

表 5.1: 分岐命令先以外のバイトコード命令を逐次実行区間に分ける命令

命令の種類	命令名
条件分岐	ifeq
	iflt
	ifle
	ifne
	ifgt
	ifge
	ifnull
	ifnonnull
	if_icmpeq
	if_icmpne
	if_icmplt
	if_icmpgt
	if_icmple
	if_icmpge
	if_acmpeq
	if_acmpne
複合条件分岐	tableswitch
	lookupswitch
無条件分岐	goto
	goto_w
	jsr
	jsr_w
	ret
メソッド呼び出し	invokevirtual
	invokeinterface
	invokespecial
	invokestatic
リターン命令	ireturn
	lreturn
	freturn
	dreturn
	areturn
	return

し命令が見つかった場合には, そのメソッドが今まで 1 度も呼ばれていないことを確認する. 呼ばれていない場合は, 呼び出されたメソッドのクラス名, メソッド名, シグネチャをそれぞれの ArrayList に格納する. 呼び出したメソッドの ConstantPoolGen を記録する. これは, 区間に分けたバイトコード命令に, 次に向かう区間を設定するために使用する. 以上の操作を以下にコードで示す.

```
// 探索をおこなうメソッドを示すための値
Integer searchingMethodIndex_ = new Integer(0);
// 探索をおこなうメソッドの数.
Integer foundMethodNumber_= new Integer(classNameList_.size());
Hashtable<String, InstructionHandle> targetHashtable_ = new Hashtable<String,
InstructionHandle>();
Hashtable<String, ConstantPoolGen> constantPoolGenHashtable_ = new Hashtable<String,
ConstantPoolGen>();
// 入力したクラスファイルのメソッドについて探索をおこなう. for (int k = 0; k < found-
MethodNumber_.intValue(); k++) {
  instructionHandleTable_.setTargetHashtableAndInvokedMethod(searchingMethodIndex_,
constantPoolGen_, classNameList_, methodNameList_, signatureList_, targetHashtable_,
constantPoolGenHashtable_);
  // 次のメソッドを探索するために, 値を 1 つ増加する.
  ++searchingMethodIndex_;
}

// setTargetHashtableAndInvokedMethod
public void setTargetHashtableAndInvokedMethod(Integer sMetIndex, ConstantPoolGen
conPoolGen, ArrayList<String> claNameList, ArrayList<String> metNameList, ArrayList<String>
sigList, Hashtable<String, InstructionHandle> tarHashtable, Hashtable<String, ConstantPoolGen>
conPoolGenHashtable) {
// insHanTable_.get(sMetIndex.intValue()).getSize() は, 探索するメソッドの Instruction-
Handle の数.
// opcode_には, バイトコード命令が持つユニークな値であるオペコードを代入する.
  for (int i = 0; i < insHanTable_.get(sMetIndex.intValue()).getSize(); i++) {
    short opcode_ = insHanTable_.get(sMetIndex.intValue()).getInstructionHandle(i).
getInstruction().getOpcode();
// 条件分岐と無条件分岐の判断 (ret を除く).
    if ((IFEQ <= opcode_ && opcode_ <= IF_ACMPNE) || IFNULL == opcode_ ||
IFNONNULL == opcode_ || GOTO == opcode_ || opcode_ == JSR || GOTO_W
== opcode_ || JSR_W == opcode_) {
```

```
        // 分岐先の命令とアドレスを記録する.
        InstructionHandle targetInstructionHandle_ = ((BranchHandle)(insHanTable_.
get(sMetIndex.intValue()).getInstructionHandle(i))).getTarget();
        Integer targetAddress_ = new Integer(targetInstructionHandle_.getPosition());
        // 分岐先の命令を tarHashtable(targetHashtable) に格納する.
        // targetKey_を生成する. tarHashTable(targetHashtable_) の Key.
        // メソッドを特定する情報+分岐先の命令のアドレス (String に変換).
        String targetKey_ = insHanTable_.get(sMetIndex.intValue()).getMethodID() + tar-
getAddress_.toString();
        tarHashtable.put(targetKey_, targetInstructionHandle_);
    }

    // 複合条件分岐の判断.
    else if (opcode_ == TABLESWITCH || opcode_ == LOOKUPSWITCH) {
        // Select クラスの getTargets() で複合条件分岐の分岐先の複数の命令を取ってくる
ことができる.
        InstructionHandle[] targetInstructionHandles_ = ((Select)(insHanTable_.
get(sMetIndex.intValue()).getInstructionHandle(i).getInstruction())). getTargets();
        // 分岐先の「複数」の命令を tarHashtable(targetHashtable) に格納する.
        for (int j = 0; j < targetInstructionHandles_ .length; j++) {
            // 分岐先の命令とアドレスを記録する.
            Integer targetAddress_ = new Integer(targetInstructionHandles_[j].getPosition());
            // targetKey_を生成する. tarHashTable(targetHashtable_) の Key.
            // メソッドを特定する情報+分岐先の命令のアドレス (String に変換).
            String targetKey_ = insHanTable_.get(sMetIndex.intValue()).getMethodID() +
targetAddress_.toString();
            tarHashtable.put(targetKey_, targetInstructionHandles_[j]);
        }
    }

    // claNameList, metNameList, sigList に呼び出されたメソッドの情報をそれぞれを
格納する.
    // fMetIndex(foundMethodIndex_) は, メソッドが呼び出されたため, 1 つ加算する.
    else if (INVOKEVIRTUAL <= opcode_ && opcode_ <= INVOKEINTERFACE) {
        // Interval の次の区間を割り当てるために, メソッド呼び出し命令のある constant-
Pool を記録する.
        conPoolGenHashtable.put(insHanTable_.get(sMetIndex.intValue()).getMethodID(),
conPoolGen);
```

```
        claNameList.add(((InvokeInstruction)insHanTable_.get(sMetIndex.intValue()).
getInstructionHandle(i).getInstruction()). getLoadClassType(conPoolGen).toString());
        metNameList.add(((InvokeInstruction)insHanTable_.get(sMetIndex.intValue()).
getInstructionHandle(i).getInstruction()). getMethodName(conPoolGen));
        sigList.add(((InvokeInstruction)insHanTable_.get(sMetIndex.intValue()).
getInstructionHandle(i).getInstruction()). getSignature(conPoolGen));
    }
  }
}
```

ここまでの操作で, 入力されたクラスファイルに書かれたメソッドに関して, プログラムストラクチャを作るための準備ができた.

次は, 入力されたクラスファイルから呼び出されたメソッドに対して, プログラムストラクチャを作るための準備をおこなっていく.

まずは, 呼び出されたメソッドの JavaClass を取得する. JavaClass の取得には, 呼び出されたメソッドのクラス名を使用する.

JavaClass を取得したあとの操作は, 入力したクラスファイルを解析したときと基本的な流れは変わらない. ただし, クラスファイルの場合と違い, メソッドは 1 つずつ解析されていく. JavaClass からプログラムストラクチャを作りだすための準備までを一貫しておこなう.

まず, メソッドが今まで呼ばれたかどうかを判定する. 呼ばれている場合, メソッドの記録をおこなっている Hashtable から Key に合致する value を命令ハンドラのテーブルに格納する.

はじめて呼び出されたメソッドと判断された場合, 入力したクラスファイル同様に ClassGen を JavaClass で初期化をおこない, ConstantPoolGen を取得して, Method の配列を取得する. 呼び出されたメソッドに関しては, メソッド名とシグネチャの情報からメソッドの特定をおこなう. もしも, Method の中に合致するメソッドが存在しない場合, 呼び出されたメソッドが所属するスーパクラスを解析して, 呼び出されたメソッドの探索をおこなう. スーパクラスの名前は, JavaClass から得る. メソッド名とシグネチャに関しては変更はない.

呼び出されたメソッドが特定できたならば, MethodGen を生成する. この後の操作も入力したクラスファイルと同様である. この操作を呼び出されるメソッドが無くなるまで探索する. 以上の操作を以下にコードで示す.

```
// 呼び出されたメソッドに関して, MethodGen の生成をおこなう.
// 発見済みのメソッドの数を更新する.
foundMethodNumber_ = classNameList_.size();
MethodGen methodGen_ = null;
```

```
// メソッドの発見に対する flag. true ならメソッドが発見されたことを示す.
boolean methodFlag_ = false;
while (searchingMethodIndex_.intValue() < foundMethodNumber_.intValue()) {
  methodIdentify_ = classNameList_.get(searchingMethodIndex_.intValue())+methodNameList_.
get(searchingMethodIndex_.intValue())+ signatureList_.get(searchingMethodIndex_.intValue());
  // 発見済みのメソッドかどうかを判断する.
  // 発見済みである場合, recursivelyInvokedMethodHashtable_から InstructionHandleList_
の参照を instructionHandleTable_に格納する.
  if (recursivelyInvokedMethodHashtable_.containsKey(methodIdentify_)) {
    instructionHandleTable_.add(recursivelyInvokedMethodHashtable_.get(methodIdentify_));
  }
  else {
    javaClazz_ = instructionHandleTable_.getJavaClass(classNameList_.get(searchingMethodIndex_.
intValue()));
    // Method の判定をおこなって, MethodGen を生成する.
    while (methodFlag_ == false) {
      clazzGen_ = new ClassGen(javaClazz_);
      constantPoolGen_ = clazzGen_.getConstantPool();
      methods_ = javaClazz_.getMethods();

      for (int l = 0; l < methods_.length && methodFlag_ == false; l++) {
        if((methods_[l].getName().equals(methodNameList_.get(searchingMethodIndex_.intValue())))
&& (methods_[l].getSignature().equals(signatureList_.get(searchingMethodIndex_.intValue()))))
{
          methodGen_ = new MethodGen(methods_[l], methods_[l].getName() , con-
stantPoolGen_);
          methodFlag_ = true;
        }
      }
      // Methods の中に合致するメソッド名とシグネチャがない場合, スーパクラスを探
索する.
      if (methodFlag_ == false) {
        JavaClass superClazzJavaClazz_ = instructionHandleTable_.getJavaClass(javaClazz_.
getSuperclassName());
        javaClazz_ = superClazzJavaClazz_;
      }
    }
    methodFlag_ = false; // 判定が終わったらフラグをおろす.
```

```
    // メソッドの判定をおこない, MethodGen から InstructionHandle を取得し, InstructionHandleTable に格納していく.
    // メソッドの Abstract Method の判定と InstructionHandleTable への格納.
    if (methodGen_.isAbstract()) {
      instructionHandleTable_.setAbstractMethod(searchingMethodIndex_.intValue(), methodIdentify_, recursivelyInvokedMethodHashtable_);
    }
    // メソッドの Native Method の判定と InstructionHandleTable への格納.
    else if (methodGen_.isNative()) {
      instructionHandleTable_.setNativeMethod(searchingMethodIndex_.intValue(), methodIdentify_, recursivelyInvokedMethodHashtable_);
    }
    // メソッドの InstructionHandleTable への格納.
    else {
      instructionHandleTable_.setMethod(searchingMethodIndex_.intValue(), methodIdentify_, recursivelyInvokedMethodHashtable_, methodGen_);
      instructionHandleTable_.setTargetHashtableAndInvokedMethod(searchingMethodIndex_, constantPoolGen_, classNameList_, methodNameList_, signatureList_, targetHashtable_, constantPoolGenHashtable_);
    }
  }
  // 次のメソッドを探索するために, 1 つ値を増加する.
  ++searchingMethodIndex_;
  // 発見済みのメソッドの数を更新する.    foundMethodNumber_ = classNameList_.size();
}


  // getJavaClass public JavaClass getJavaClass(String claName) throws ClassNotFoundException {
  return(Repository.lookupClass(claName));
}
```

この時点でプログラムストラクチャを作成するための準備が整った. プログラムストラクチャのデータ構造は, ArrayList で実装した 3 次元配列である. 図 5.2 がプログラムストラクチャの概念図になる.

逐次実行区間に区切るために, 命令ハンドラのテーブルを探索する. 命令ハンドラのテーブルを作るときと同様に, 探索したメソッドは記録しておく. プログラムストラクチャを作りだすときに, すでに探索したメソッドであるかどうかを判断するためである. バイト

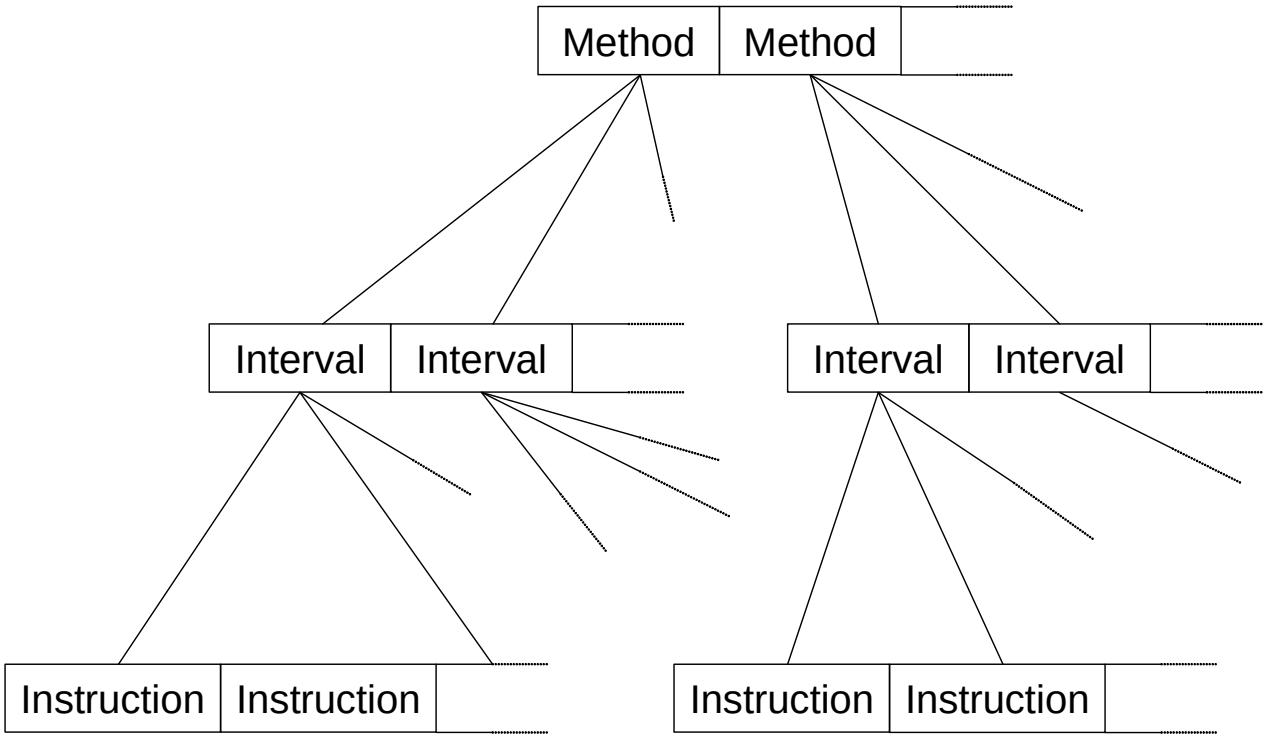


図 5.2: プログラムストラクチャのデータ構造

コードを区切る規則は, 以下の通りである.

1. 逐次実行区間に分ける命令を発見するまで, 逐次実行命令であるバイトコード命令を配列に記録していく. メソッドの最後のバイトコード命令までたどり着いたときは, バイトコード命令を格納してきた配列に格納する.
2. 逐次実行区間に分ける命令であるバイトコード命令を発見した場合, 逐次実行区間に分ける命令の種類別に, 区間に分ける.
   - 条件分岐命令, 無条件分岐, 複合条件分岐命令, メソッド呼び出し命令, リターン命令の場合.
     逐次実行区間に分ける命令まで探索して, バイトコード命令を格納してきた配列に, 逐次実行区間に分ける命令であるバイトコード命令を記録する. 次に探索するバイトコード命令からは, 新しい配列にバイトコード命令を格納していく.
   - 分岐先の命令の場合.
     逐次実行区間に分ける命令まで探索して, バイトコード命令を格納してきた配列までで区切る. 逐次実行区間に分ける命令は新しい配列に格納して, 残りのバイトコード命令の探索を続ける. 配列が空の場合は, そのまま逐次実行区間に分ける命令を格納して, 残りのバイトコード命令の探索を続ける.

1つのメソッドのバイトコード命令を最後まで探索したら, 区切られた命令ハンドラをプログラムストラクチャとして記録する. 以上の操作を以下にコードとして示す.

```
ProgramStructure programStructure_ = new ProgramStructure();
Hashtable<String, IntervalTable> recursivelyIntervalInvokedMethodHashtable_ = new
Hashtable<String, IntervalTable>();
// メソッドごとに InstructionHandle を調べて, プログラムストラクチャに格納していく.
//for (int p = 0; p < foundMethodNumber_.intValue(); p++) {
  // メソッド判定用に instructionHandleList_からメソッドを特定する情報を取得する.
  methodIdentify_ = instructionHandleTable_.getInstructionHandleList(p).getMethodID();
  // すでに区間を区切ったメソッドは, プログラムストラクチャに記録だけしてスキップ
する.
  if(recursivelyIntervalInvokedMethodHashtable_.containsKey(methodIdentify_)) {
    programStructure_.add(recursivelyIntervalInvokedMethodHashtable_.get(methodIdentify_));
  }
  // メソッドの Abstract Method の判定と ProgramStructure への格納.
  else if (instructionHandleTable_.getInstructionHandleList(p).isAbstract()) {
    programStructure_.setIntervalAbstractMethod(p, methodIdentify_, recursivelyInter-
valInvokedMethodHashtable_);
  }
  // メソッドの Native Method の判定と ProgramStructure への格納.
```

```
  else if (instructionHandleTable_.getInstructionHandleList(p).isNative()) {
    programStructure_.setIntervalNativeMethod(p, methodIdentify_, recursivelyInterval-
InvokedMethodHashtable_);
  }
  // メソッドの ProgramStructure への格納.
  else {
    programStructure_.setIntervalMethod(p, methodIdentify_, recursivelyIntervalInvoked-
MethodHashtable_, instructionHandleTable_.getInstructionHandleList(p), targetHashtable_);
  }
}

// setIntervalAbstractMethod public void setIntervalAbstractMethod(int metIndex, String
metID, Hashtable<String, IntervalTable> reInterInvMetHashtable) {
  IntervalTable interTable_ = new IntervalTable(metID);
  interTable_.setIntervalAbstractMethodCheckingNumber();
  interTable_.setAbstractMethod();
  progStructure_.add(interTable_);
  reInterInvMetHashtable.put(metID, progStructure_.get(metIndex));
}

// setIntervalNativeMethod
public void setIntervalNativeMethod(int metIndex, String metID, Hashtable<String, IntervalTable>
reInterInvMetHashtable) {
  IntervalTable interTable_ = new IntervalTable(metID);
  interTable_.setIntervalNativeMethodCheckingNumber();
  interTable_.setNativeMethod();
  progStructure_.add(interTable_);
  reInterInvMetHashtable.put(metID, progStructure_.get(metIndex));
}

// setIntervalMethod
public void setIntervalMethod(int metIndex, String metID, Hashtable<String, IntervalTable>
reInterInvMetHashtable,    InstructionHandleList insHanList, Hashtable<String, InstructionHandle>
tarHashtable) {
  IntervalTable interTable_ = new IntervalTable(metID);

  for(int i = 0; i < insHanList.getSize(); i++) {
    interTable_.setMethod(i, insHanList, tarHashtable);
```

```
      // InterTable 内で更新された i(=insHanIndex) を ProgramStructure で取得する.
      i = interTable_.getCount();    }
   // interList_が空でない, かつ分岐先の命令の場合の処理.
   // IntervalList を生成して, 分岐先の命令を格納する.
   // そして, IntervalTable に格納する.(つまり, 最後の命令だけが格納された IntervalList
ができる)
   if (interTable_.getTargetInstructionHandle() != null) {
      IntervalList interList_ = new IntervalList();
      interList_.setInstructionHandle(interTable_.getTargetInstructionHandle());
      interTable_.add(interList_);
   }
   progStructure_.add(interTable_);
   reInterInvMetHashtable.put(metID, progStructure_.get(metIndex));
}

// setMethod(プログラムストラクチャ用)
public void setMethod(int insHanIndex, InstructionHandleList iHList, Hashtable<String,
InstructionHandle> tHashtable ) {
   IntervalList interList_ = new IntervalList();

   // メソッドのすべての InstructionHandle を判断していく.
   // while(判断している InstructionHandle の Index < InstructionHandle)
   while(insHanIndex < iHList.getSize()) {
      // InstructionHandle を判断するための opcode_を設定.
      // 判断する InstructionHandle の短縮変数を設定.
      // 分岐先の命令であるかを判断するための一部である instructionHandleAddress_を
生成.
      // 分岐の命令であるかを判断する targetKet_を生成する.

      Short opcode_ = iHList.getInstructionHandle(insHanIndex).getInstruction().getOpcode();
      InstructionHandle checkingInstructionHandle_ = iHList.getInstructionHandle(insHanIndex);
      Integer instructionHandleAddress_ = new Integer(iHList.getInstructionHandle(insHanIndex).
getPosition());
      String targetKey_ = iHList.getMethodID() + instructionHandleAddress_.toString();

      // 1. 分岐先の命令がある場合, interList_に分岐先の命令 (insHanIndex-1 の命令であ
る) を格納する.
      if (targetInstructionHandle_ != null) {
```

```
        interList_.setInstructionHandle(targetInstructionHandle_);
        targetInstructionHandle_ = null; // 初期化
    }

    // 2-1. 分岐先の命令であるかどうかを判断する.
    if (tHashtable.containsKey(targetKey_)) {
        // 区切るまでの interList_が空であるかどうかを判断する.
        // interList_が空である場合, interList_に分岐先の命令を格納する.
        if (interList_.isEmpty()) {
            interList_.setInstructionHandle(checkingInstructionHandle_);
            //interList_が空, かつ最後の命令の場合の処理である.
            if (insHanIndex == (iHList.getSize() - 1)) {
                intervalTable_.add(interList_);
                break; // setMethod を抜ける.
            }
        }

        // interList_が空でない場合, 今まで格納してきた interList_を intervalTable_に格
納する.
        // 分岐先の命令は一時的な置き場所である targetInstructionHandle_に記録する.
        // intervalTable_に interList_を格納するので, IntervalTable から抜ける必要があ
る.
        else {
            intervalTable_.add(interList_);
            targetInstructionHandle_ = checkingInstructionHandle_;
            break;
        }
    }

    // 2-2. リストの終端になる InstructionHandle の場合の判断.
    // 条件分岐・無条件分岐・リターン命令・メソッド呼び出し・最後の命令.
    else if ((IFEQ <= opcode_ && opcode_ <= IF_ACMPNE) || IFNULL == opcode_
|| IFNONNULL == opcode_ || (GOTO <= opcode_ && opcode_ <= JSR) || opcode_
== RET || GOTO_W == opcode_ || JSR_W == opcode_ || TABLESWITCH ==
opcode_ || LOOKUPSWITCH == opcode_ || (IRETURN <= opcode_ && opcode_
<= RETURN) || (INVOKEVIRTUAL <= opcode_ && opcode_ <= INVOKEINTER-
FACE) || insHanIndex == (iHList.getSize() - 1)) {
        interList_.setInstructionHandle(checkingInstructionHandle_);
```

```
        intervalTable_.add(interList_);
        break;
      }
/**       // 2-3. 逐次実行命令の処理.
      // interList_に InstructionHandle を格納する.
      else {
        interList_.setInstructionHandle(checkingInstructionHandle_);
      }
      ++insHanIndex;
    }
    searchingInstructionHandleIndex_ = insHanIndex;
}
```

プログラムストラクチャの逐次実行区間が次に向かう区間を記録していく. 次に向かう区間は, メソッドを特定する情報と区間のインデックスを連結した情報で表す. メソッド呼び出し命令の場合は, 呼び出されたメソッドを特定する情報と最初の区間のインデックスを連結した情報となる. 条件分岐や複合条件分岐の場合は, 向かう区間は複数になる. 以上の操作を以下にコードとして示す.

```
recursivelyIntervalInvokedMethodHashtable_.clear();
programStructure_.setIntervalNextInterval(constantPoolGenHashtable_, recursivelyInter-
valInvokedMethodHashtable_);

// setNextInterval
public void setIntervalNextInterval(Hashtable<String, ConstantPoolGen> conPoolGen-
Hashtable, Hashtable<String, IntervalTable> reInterInvMetHashtable) {
    InstructionHandle intervalLastInstructionHandle_ = null;
    InstructionHandle targetInstructionHandle_ = null;
    InstructionHandle[] targetInstructionHandles_ = null;
    boolean nextIntervalFlag_ = false;
    short opcode_ = 0;
    String methodID_ = null;
    String nextInter_ = null;
    ConstantPoolGen conPoolGen_ = null;

    for (int i = 0; i < progStructure_.size(); i++) {
      methodID_ = progStructure_.get(i).getMethodID();
```

```
    if (reInterInvMetHashtable.containsKey(methodID_)) {
      // すでに探索したメソッドの場合, スキップする.       }
    else {
      for (int j = 0; j < progStructure_.get(i).getSize(); j++) {
        if (progStructure_.get(i).isAbstract()) {
          nextInter_ = ABSTRACT;
          progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
          reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
        }
        else if (progStructure_.get(i).isNative()) {
          nextInter_ = NATIVE;
          progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
          reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
        }
        else

          // 区間の最後の命令から次の区間がどこであるかを判断して記録する.
          // InstructionHandle ごとの処理をおこなって, nextInterval_に次の向かう区
間を格納する.
          intervalLastInstructionHandle_ = progStructure_.get(i).getIntervalList(j).
getInstructionHandle(progStructure_.get(i).getIntervalList(j).getSize() -1);
          opcode_ = intervalLastInstructionHandle_.getInstruction().getOpcode();

          // 1-1 条件分岐
          if ((IFEQ <= opcode_ && opcode_ <= IF_ACMPNE) || IFNULL == op-
code_ || IFNONNULL == opcode_) {
            // 条件分岐が false の場合の次の区間を代入して, 格納する.
            // 次の区間は, メソッドを特定する情報+区間の Index である.
            nextInter_ = progStructure_.get(i).getMethodID() + (j+1);
            progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
            // 分岐先の命令を探索するための短縮変数を設定.
            targetInstructionHandle_ = ((BranchHandle)intervalLastInstructionHandle_).
getTarget();

            // 区間の中の分岐先の命令を探索する.
            for (int loopC1 = 0; loopC1 < progStructure_.get(i).getSize(); loopC1++)
{
              if (nextIntervalFlag_ == true) {
```

```
                    break;
                }
                // 条件分岐命令の true の場合の命令を探索して, 格納する.
                // 分岐先の命令は IntervalList の最初に格納されている.
                else if (targetInstructionHandle_ == progStructure_.get(i).getIntervalList(loopC1).
getInstructionHandle(0)) {
                    nextInter_ = progStructure_.get(i).getMethodID() + loopC1;
                    progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
                    nextIntervalFlag_ = true;
                }
            }
            nextIntervalFlag_ = false;
            reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
        }

        // 1-2 無条件分岐
        else if (GOTO == opcode_ || opcode_ == JSR || GOTO_W == opcode_
|| JSR_W == opcode_) {
            targetInstructionHandle_ = ((BranchHandle)intervalLastInstructionHandle_).
getTarget();
            // 区間の中の分岐先の命令を探索する.
            for (int loopC2 = 0; loopC2 < progStructure_.get(i).getSize(); loopC2++)
{
                if (nextIntervalFlag_ == true) {
                    break;
                }
                // 分岐先の命令は IntervalList の最初に格納されている.
                else if (targetInstructionHandle_ == progStructure_.get(i).getIntervalList(loopC2).
getInstructionHandle(0)) {
                    nextInter_ = progStructure_.get(i).getMethodID() + loopC2;
                    progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
                    reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
                    nextIntervalFlag_ = true;
                }
            }
            nextIntervalFlag_ = false;
            reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
        }
```

```
            // 1-3 複合条件分岐命令
            else if (opcode_ == TABLESWITCH || opcode_ == LOOKUPSWITCH) {
              // 複数の分岐先の命令を targetInstructionHandles_格納する.
              targetInstructionHandles_ = ((Select)(intervalLastInstructionHandle_.getInstruction())).
getTargets();
              // targetInstructionHandles_から個々の分岐先の命令のある区間を探索する.
              for (int multiTarIndex = 0; multiTarIndex < targetInstructionHandles_.length;
multiTarIndex++) {
                // 区間の中の分岐先の命令を探索する.
                for (int loopC3 = 0; loopC3 < progStructure_.get(i).getSize(); loopC3++)
{
                  if (nextIntervalFlag_ == true) {
                    break;
                  }
                  else if (targetInstructionHandles_[multiTarIndex] == progStructure_.get(i).
getIntervalList(loopC3).getInstructionHandle(0)) {
                    nextInter_ = progStructure_.get(i).getMethodID() + loopC3;
                    progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
                    nextIntervalFlag_ = true;
                  }
                }
                nextIntervalFlag_ = false; // 初期化
              }
              reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
            }

            // 1-4ret 命令
            else if (opcode_ == RET) {
              // 区間の中の分岐先の命令を探索する.
              for (int loopC4 = 0; loopC4 < progStructure_.get(i).getSize(); loopC4++)
{
                // jsr, jsr_w のある区間すべてを次の区間として格納する.
                if (JSR == progStructure_.get(i).getIntervalList(loopC4).getInstructionHandle(0).
getInstruction().getOpcode() || JSR_W == progStructure_.get(i).getIntervalList(loopC4).
getInstructionHandle(0).getInstruction().getOpcode()) {
                  nextInter_ = progStructure_.get(i).getMethodID() + loopC4;
                  progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
```

```
                }
              }
              reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
            }

            // 1-5 メソッド呼び出し命令
            else if (INVOKEVIRTUAL <= opcode_ && opcode_ <= INVOKEINTER-
FACE) {
              conPoolGen_ = conPoolGenHashtable.get(progStructure_.get(i).getMethodID());
              // 呼び出されたメソッドを特定する情報+最初の区間 (0 番目) を連結した値
を次の区間に設定する,
              nextInter_ = ((InvokeInstruction)intervalLastInstructionHandle_.getInstruction()).
getLoadClassType(conPoolGen_).toString() + ((InvokeInstruction)intervalLastInstructionHandle_.
getInstruction()).getMethodName(conPoolGen_) + ((InvokeInstruction)intervalLastInstructionHandle_.ge
+ 0;
              progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
              reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
            }

             // 1-6 リターン命令
            else if (IRETURN <= opcode_ && opcode_ <= RETURN) {
              // 命令の名前を設定する.
              nextInter_ = intervalLastInstructionHandle_.getInstruction().toString();
              progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
              reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
            }

            // 1-7 逐次実行命令
            else {
              // 次の区間は, メソッドを特定する情報+区間の Index である.
              nextInter_ = progStructure_.get(i).getMethodID() + (j+1);
              progStructure_.get(i).getIntervalList(j).setNextInterval(nextInter_);
              reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
            }
          }
        }
      }
    }
```

}

バイトコード命令の実行時間表から, 区間に時間を割り当てる. 今回は, バイトコード命令の実行時間は仮の時間を設定した. 以上の操作を以下にコードとして示す.

```
recursivelyIntervalInvokedMethodHashtable_.clear();
programStructure_.setIntervalTime(recursivelyIntervalInvokedMethodHashtable_);

// setIntervalTime
public void setIntervalTime(Hashtable<String, IntervalTable> reInterInvMetHashtable)
{
  short opcode_ = 0;
  String methodID_ = null;

  for (int i = 0; i < progStructure_.size(); i++) {
    methodID_ = progStructure_.get(i).getMethodID();
    if (reInterInvMetHashtable.containsKey(methodID_)) {
      // すでに探索したメソッドの場合, スキップする.
    }
    else {
      for (int j = 0; j < progStructure_.get(i).getSize(); j++) {
        for (int k = 0; k < progStructure_.get(i).getIntervalList(j).getSize(); k++) {
          opcode_ = progStructure_.get(i).getIntervalList(j).getInstructionHandle(k).
                    getInstruction().getOpcode();
          switch (opcode_) {
          case NOP: //0
            progStructure_.get(i).getIntervalList(j).setTime(NOP_TIME);
            break;
          case ACONST_NULL: // 1
            progStructure_.get(i).getIntervalList(j).setTime(NOP_TIME);
            break;
(中略)
          case JSR_W: // 201
            progStructure_.get(i).getIntervalList(j).setTime(JSR_W_TIME);
            break;
          }
        }
```

```
            reInterInvMetHashtable.put(methodID_, progStructure_.get(i));
          }
        }
      }
    }
```

第6章　実験的実装

5章で提案したBCELを用いたクラスファイルの解析によるプログラムストラクチャを作りだすツールを実験的に実装した.

ツールを実装する課程でいくつかの問題が発見された. そのため, リアルタイムJavaに対してのツールの適応ができなかったため, 通常のJavaのクラスファイルにツールを適応させている.

結果としては, 本研究の事例対象に限って, プログラムストラクチャを作ることに成功した.

6.1　事例対象としたJavaのプログラムの動作

Javaによるマルチスレッドのプログラムである. main()メソッドで, スレッドを3つ生成する. スレッドは, スレッドの名前と5秒づつずれた値で初期化する. 3つのスレッドを生成したら, それぞれのスレッドをスタートする. スレッドは初期値として与えただけスレッドを停止する. その後, スレッドの停止時間が終わったら, スレッドの名前とスレッドの提示時間を表示させて終了する.

6.2　プログラムストラクチャの作成

事例対象としたJavaのクラスファイルをツールにかけた結果を部分的に抜きだして, バイトコード命令とプログラムストラクチャで比較したのが図6.1である.

このバイトコード命令は, 事例としたプログラムのmain()メソッドである. バイトコード命令を逐次実行区間に分ける命令で区切ることができていることがわかる. このことにより, 本研究で事例としたクラスファイルに限り, また, ツールの実装過程の中で見つかった問題を考えない場合, プログラムストラクチャの作成がおこなえたことが確認できた.

この事例では, プログラムストラクチャが1845個見つかった. そのうち2回以上呼び出されたメソッドの数は1185個であった. 呼び出されたNativeMethodの数が100個であった呼び出されたAbstractメソッドの数が44個である.

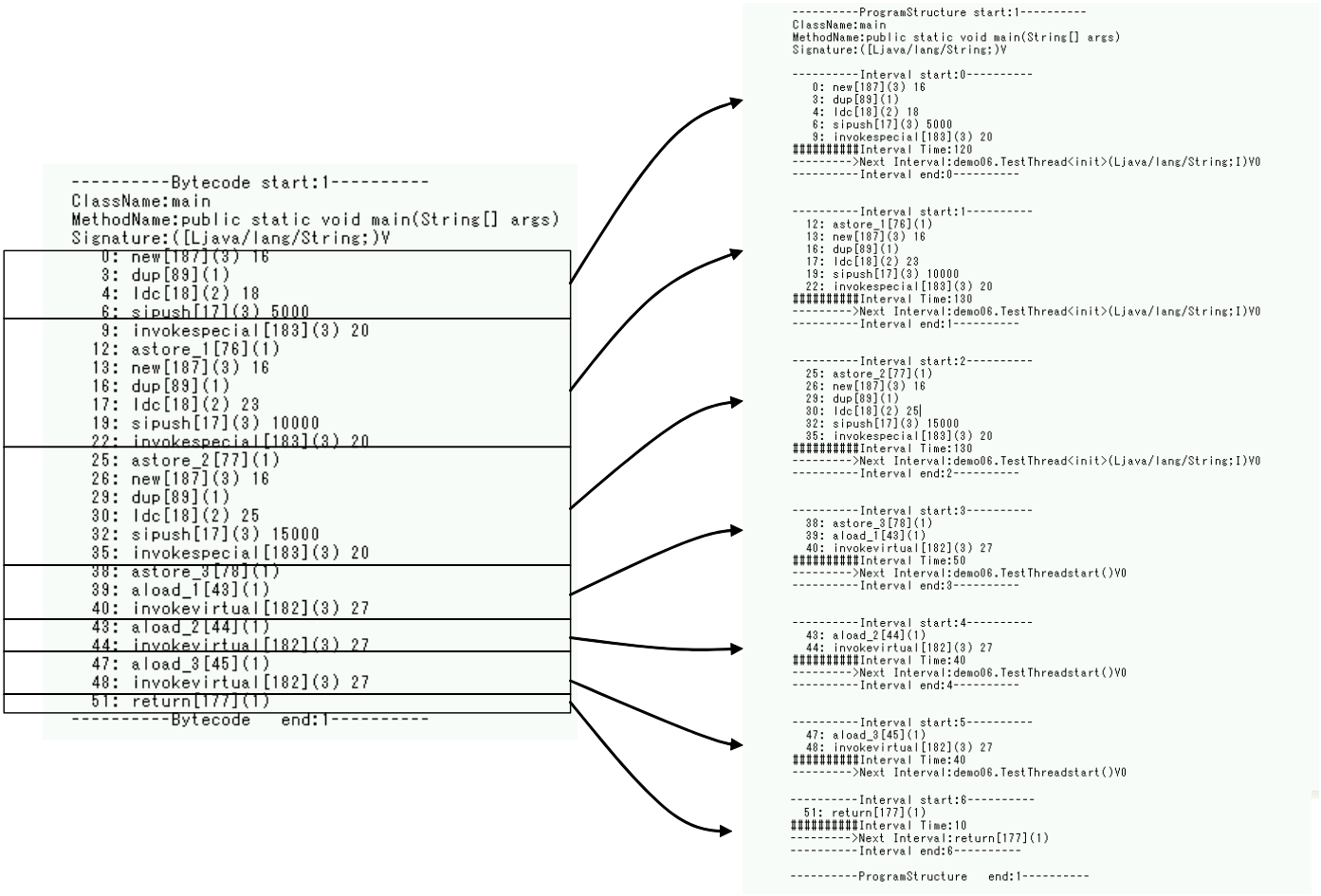


図 6.1: バイトコード命令とプログラムストラクチャの比較

6.3 問題の発見

ツールを実装する過程で見つかった問題を挙げる.

- Native メソッドの処理.

  Native メソッドは, プラットフォーム依存のコードによって実装されている. 典型的に C, C++, FORTRAN, アセンブリ言語といった, ほかのプログラミング言語によって記述される.

  ほかの言語で書かれるためにバイトコード命令を持たない. そのため, 本研究で提案した方法では, プログラムストラクチャを作ることはできない. 現在はバイトコード命令を持たないメソッドとして, プログラムストラクチャに格納している.

- abstract メソッドの処理

  abstract メソッドは, シグネチャ(メソッドの名前とパラメータの数と型), 戻り型, また, あるならば throws 節を指定し, 実装を指定せずにメンバとしてメソッドを導入するものである.

  実装をしていないため, abstract メソッドそのものはバイトコード命令を持たない.

  また, abstract メソッドには, 実装されているメソッドがあるはずである. つまり, abstract メソッドを発見した場合は, abstract メソッドを実装しているメソッドを見つけにいく必要がある. 現在, ツールでは abstract メソッドを発見した場合, バイトコード命令を持たないメソッドとして, プログラムストラクチャに格納して, それ以上の処理はおこなっていない.

- アドレスをローカル変数から取得する無条件分岐命令 ret

  分岐命令は, バイトコード命令のアドレスの相対的な値で分岐する. しかし, 無条件分岐命令 ret は, 分岐先のアドレスをローカル変数から取りだす. これは, プログラムの実行時でなくては分岐先を一意に決めることができないかもしれない, ということである. プログラムの実行時に, 分岐先のアドレスを取り出すローカル変数の値が変わる可能性があるからである. 現在, ret が現れるクラスファイルに関しては対応ができておらず, 本研究で実験的に実装したツールでは, プログラムストラクチャを作りだすことができない.

第7章　考察

実験的に実装したツールによって, 今回事例としたクラスファイルから作りだされたプログラムストラクチャとツールを作る過程で発見した問題点について考察をおこなう.

7.1　プログラムストラクチャを作りだすツールの実験的な実装について

本研究で提案したプログラムストラクチャを実験的に実装することで, 実験をおこなった範囲に限って, プログラムストラクチャを作りだすことができることを確認した. この結果から, クラスファイルからプログラムストラクチャへの変換についての問題点を見つけることができた. また, 発見された問題に対応するための方法について模索することができた.

7.2　発見された問題点について

Native メソッドは, その実行に関して, リアルタイム Java の実行環境の実行時間を計り, その値を割り当てるという方法が考えられる. また, 別の方法としては, 実装されている言語をバイトコード命令に置き換えるコンバータを作る, といった方法も考えられる.

Abstract メソッドは, 実装されているメソッドを見つけることが静的な解析に限った場合, 困難であると考える. 考えられる対応策としては, Abstract メソッドのメソッドのシグネチャなどの情報から, 実装される可能性のあるメソッドを多重分岐として扱う, という方法が考えられる.

アドレスをローカル変数から取得する無条件分岐命令 ret は, 静的な解析に限ると, メソッド内で ret の取りうる分岐先を調べて, 多重分岐として扱う, という方法で対処できる.

第8章　おわりに

8.1　まとめ

本研究では, リアルタイム Java に対する検証支援環境の提案をおこなった. 検証支援環境の最初の操作であるプログラムストラクチャを作りだす方法について具体的な提案をおこなった. プログラムストラクチャを作りだすツールの実験的な実装をおこなった. プログラムストラクチャを作りだすツールによって, 本研究で扱った事例に関しては, プログラムストラクチャを作りだすことができた. 発見された問題点に関しては, 対応できる方法を模索し, 指針を考えた.

8.2　今後の課題

本研究で実験的に実装したプログラムストラクチャをつくりだすツールは検証をおこなうために必要なパラメトリックタイムストラクチャを作りだすには不完全である. そのため, 今回の実験で対応できなかった項目について対処する必要がある.

検証支援環境を完成させるには, プログラムストラクチャを作りだしたあとの操作についても考えていく必要がある. リアルタイム Java の実行環境でのバイトコード命令の時間を計る方法, 時間付けしたプログラムストラクチャからのパラメトリックタイムストラクチャの作り方, スレッドごとにつくられたパラメトリックタイムストラクチャの合成方法が挙げられる.

謝辞

本論文の執筆にあたって終始熱心な御指導を頂きました片山卓也教授に深く感謝申し上げます. また, 貴重な御意見と有意義な議論を共にして頂きました Chaiwat Sathawornwichit さん, ならびにソフトウェア基礎講座の皆様に厚く感謝を申し上げます.

付録

本研究において事例に用いたプログラムを載せる.

```
class TestThread extends Thread {
  private String thname;
  private int delay;

  public TestThread(String str, int del) {
    thname = str;
    delay = del;
  }
  public void run() {
    try {
      Thread.sleep(delay);
    } catch (InterruptedException e) {
    }
    System.out.println(thname + ":" + delay);
  }
}

public class ThreeThreadTest {
  public static void main(String args[]) {
    TestThread t1, t2, t3;

    t1 = new TestThread("Thread 1", 5000);
    t2 = new TestThread("Thread 2", 10000);
    t3 = new TestThread("Thread 3", 15000);
    t1.start();
    t2.start();
    t3.start();
  }
```

```
}
```

```
----------Bytecode start:0----------
ClassName:<init>
MethodName:public void <init>()
Signature:()V
   0: aload_0[42](1)
   1: invokespecial[183](3) 8
   4: return[177](1)
----------Bytecode   end:0----------

----------Bytecode start:1----------
ClassName:main
MethodName:public static void main(String[] args
Signature:([Ljava/lang/String;)V
   0: new[187](3) 16
   3: dup[89](1)
   4: ldc[18](2) 18
   6: sipush[17](3) 5000
   9: invokespecial[183](3) 20
  12: astore_1[76](1)
  13: new[187](3) 16
  16: dup[89](1)
  17: ldc[18](2) 23
  19: sipush[17](3) 10000
  22: invokespecial[183](3) 20
  25: astore_2[77](1)
  26: new[187](3) 16
  29: dup[89](1)
  30: ldc[18](2) 25
  32: sipush[17](3) 15000
  35: invokespecial[183](3) 20
  38: astore_3[78](1)
  39: aload_1[43](1)
  40: invokevirtual[182](3) 27
  43: aload_2[44](1)
  44: invokevirtual[182](3) 27
  47: aload_3[45](1)
  48: invokevirtual[182](3) 27
  51: return[177](1)
----------Bytecode   end:1----------

----------Bytecode start:2----------
ClassName:java.lang.Object
MethodName:<init>
Signature:()V
   0: return[177](1)
----------Bytecode   end:2----------

----------Bytecode start:3----------
ClassName:demo00.TestThread
MethodName:<init>
Signature:(Ljava/lang/String;I)V
   0: aload_0[42](1)
   1: invokespecial[183](3) 12
   4: aload_0[42](1)
   5: aload_1[43](1)
   6: putfield[181](3) 15
   9: aload_0[42](1)
  10: iload_2[28](1)
  11: putfield[181](3) 17
  14: return[177](1)
----------Bytecode   end:3----------
```

```
----------Bytecode start:4----------
ClassName:demo00.TestThread
MethodName:<init>
Signature:(Ljava/lang/String;I)V
   0: aload_0[42](1)
   1: invokespecial[183](3) 12
   4: aload_0[42](1)
   5: aload_1[43](1)
   6: putfield[181](3) 15
   9: aload_0[42](1)
  10: iload_2[28](1)
  11: putfield[181](3) 17
  14: return[177](1)
----------Bytecode   end:4----------

----------Bytecode start:5----------
ClassName:demo00.TestThread
MethodName:<init>
Signature:(Ljava/lang/String;I)V
   0: aload_0[42](1)
   1: invokespecial[183](3) 12
   4: aload_0[42](1)
   5: aload_1[43](1)
   6: putfield[181](3) 15
   9: aload_0[42](1)
  10: iload_2[28](1)
  11: putfield[181](3) 17
  14: return[177](1)
----------Bytecode   end:5----------

----------Bytecode start:6----------
ClassName:demo00.TestThread
MethodName:start
Signature:()V
   0: aload_0[42](1)
   1: getfield[180](3) 375
   4: ifeq[153](3) -> aload_0
   7: new[187](3) 204
  10: dup[89](1)
  11: invokespecial[183](3) 403
  14: athrow[191](1)
  15: aload_0[42](1)
  16: iconst_1[4](1)
  17: putfield[181](3) 375
  20: aload_0[42](1)
  21: getfield[180](3) 385
  24: aload_0[42](1)
  25: invokevirtual[182](3) 455
  28: aload_0[42](1)
  29: invokespecial[183](3) 428
  32: return[177](1)
----------Bytecode   end:6----------
```

図 8.1: 事例のバイトコードの一部その 1

```
----------Bytecode start:7----------
ClassName:demo00.TestThread
MethodName:start
Signature:()V
   0: aload_0[42](1)
   1: getfield[180](3) 375
   4: ifeq[153](3) -> aload_0
   7: new[187](3) 204
  10: dup[89](1)
  11: invokespecial[183](3) 403
  14: athrow[191](1)
  15: aload_0[42](1)
  16: iconst_1[4](1)
  17: putfield[181](3) 375
  20: aload_0[42](1)
  21: getfield[180](3) 385
  24: aload_0[42](1)
  25: invokevirtual[182](3) 455
  28: aload_0[42](1)
  29: invokespecial[183](3) 428
  32: return[177](1)
----------Bytecode   end:7----------

----------Bytecode start:8----------
ClassName:demo00.TestThread
MethodName:start
Signature:()V
   0: aload_0[42](1)
   1: getfield[180](3) 375
   4: ifeq[153](3) -> aload_0
   7: new[187](3) 204
  10: dup[89](1)
  11: invokespecial[183](3) 403
  14: athrow[191](1)
  15: aload_0[42](1)
  16: iconst_1[4](1)
  17: putfield[181](3) 375
  20: aload_0[42](1)
  21: getfield[180](3) 385
  24: aload_0[42](1)
  25: invokevirtual[182](3) 455
  28: aload_0[42](1)
  29: invokespecial[183](3) 428
  32: return[177](1)
----------Bytecode   end:8----------
```

```
----------Bytecode start:9----------
ClassName:java.lang.Thread
MethodName:<init>
Signature:()V
   0: aload_0[42](1)
   1: invokespecial[183](3) 405
   4: aload_0[42](1)
   5: iconst_0[3](1)
   6: putfield[181](3) 374
   9: aload_0[42](1)
  10: iconst_0[3](1)
  11: putfield[181](3) 376
  14: aload_0[42](1)
  15: aconst_null[1](1)
  16: putfield[181](3) 387
  19: aload_0[42](1)
  20: aconst_null[1](1)
  21: putfield[181](3) 386
  24: aload_0[42](1)
  25: iconst_0[3](1)
  26: putfield[181](3) 370
  29: aload_0[42](1)
  30: new[187](3) 207
  33: dup[89](1)
  34: invokespecial[183](3) 405
  37: putfield[181](3) 379
  40: aload_0[42](1)
  41: aconst_null[1](1)
  42: aconst_null[1](1)
  43: new[187](3) 213
  46: dup[89](1)
  47: invokespecial[183](3) 414
  50: ldc[18](2) 6
  52: invokevirtual[182](3) 417
  55: invokestatic[184](3) 421
  58: invokevirtual[182](3) 416
  61: invokevirtual[182](3) 415
  64: lconst_0[9](1)
  65: invokespecial[183](3) 447
  68: return[177](1)
----------Bytecode   end:9----------

----------Bytecode start:10----------
ClassName:java.lang.IllegalThreadStateException
MethodName:<init>
Signature:()V
   0: aload_0[42](1)
   1: invokespecial[183](3) 13
   4: return[177](1)
----------Bytecode   end:10----------
```

図 8.2: 事例のバイトコード命令の一部その 2

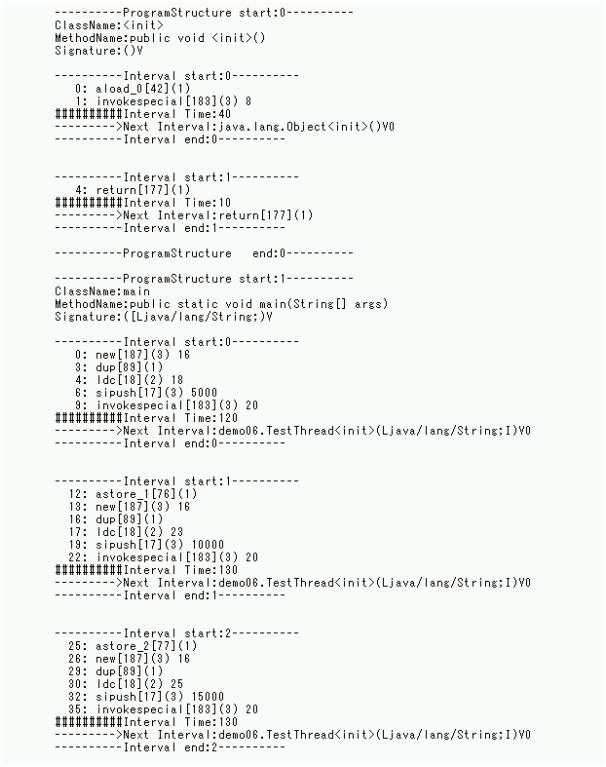
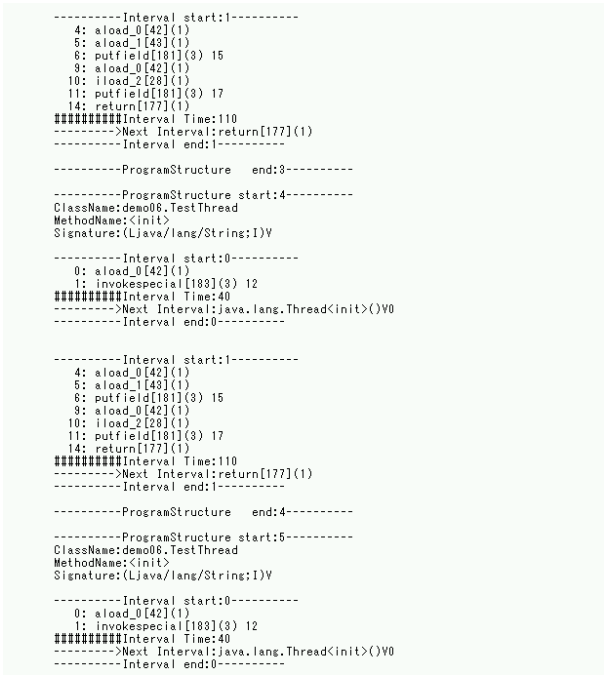


図 8.3: 事例のプログラムストラクチャの一部その 1



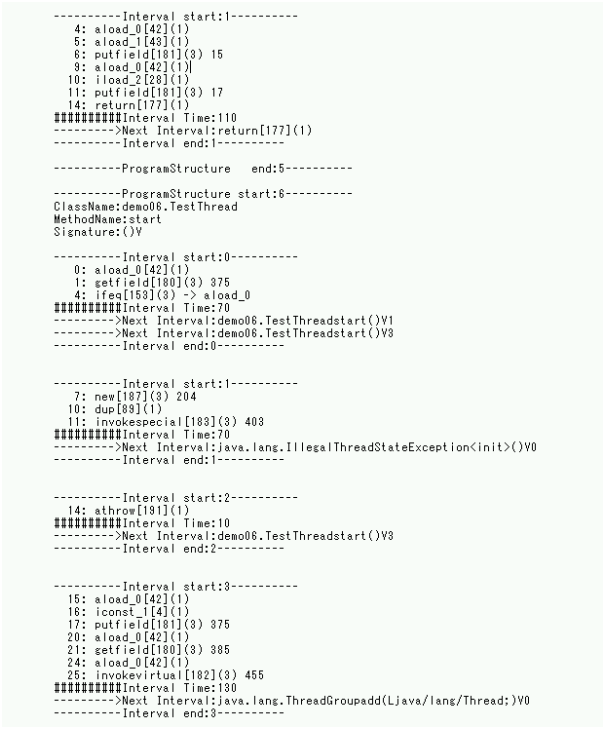


図 8.4: 事例のプログラムストラクチャの一部その 2

参考文献

[1] Chaiwat Sathawornwichit, Takuya Kataya, A Parametric Model Checking Approach for Real-Time Systems Design, 2005.

[2] Jakaruta BCEL Project.

http://jakarta.jp/bcel/.

[3] RTSJ Main Page.

http://rtsj.org/.

[4] ボレラ, ゴスリン, ブロスゴル, ディビル, フウ, ハーディン, ターンブル, (音川英之, 白川洋充 訳), リアルタイム JAVA プログラミング, ピアソン・エデュケーション, 2003.

[5] ピーター C. ディブル, (柴田芳樹 監訳, 富士ゼロックス情報システム 訳), Java リアルタイム仕様, ピアソン・エデュケーション, 2000.

[6] ティム・リンドホルム, フランク・イェリン, (村上雅章 訳), Java 仮想マシン仕様 第2版, ピアソン・エデュケーション, 2001.