JAIST Repository
https://dspace.jaist.ac.jp/

Title	Cellプロセッサ用プログラム検証法
Author(s)	レ, ディンスアン
Citation	
Issue Date	2009-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/8163
Rights	
Description	Supervisor:青木利晃, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

Cell プロセッサ用プログラム検証法

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

Le Dinh Xuan

2009 年 3 月

修 士 論 文

Cell プロセッサ用プログラム検証法

指導教官　青木利晃 特任准教授

審査委員主査　青木利晃 特任准教授
審査委員　小川瑞史 教授
審査委員　岸知二 特任教授

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

0710078　Le Dinh Xuan

提出年月: 2009 年 2 月

Copyright © 2009 by Le Dinh Xuan

概 要

最近，複数のコアから構成されたマルチコア・プロセッサの開発の発展により，ハイパフォーマンス計算の技術が PC と家族電気商品の世界までにも普及している．代表的に Cell プロセッサにより，普通のプログラマでもマルチコア上のソフトウェア開発ができ，普通の人でも Cell プロセッサが入っている PlayStation3 でハイパフォーマンスのゲームなどを体験できる．しかし，ハイパフォーマンスのために，Cell プロセッサは複雑なアーキテクチャを持ち，プログラマはそのアーキテクチャの全ての特徴に気を付けながら多くの手動最適化手段を適用してプログラムを作成する必要がある．結局，Cell プロセッサ用プログラムに対して振舞いの正しさの検証とパフォーマンスの解析・視覚化の作業は極めて複雑になり，従来の手法いわゆるテスティング，デバッグ，シミュレーションなどは効果がなくなる．解決策として，本研究はモデル検査技術を用い，プログラムの正確さとパフォーマンスの両方が検証できる枠組みを提案する．

Cell プロセッサ用のプログラムの正確さとパフォーマンスを有効に検証する要求は本研究の動機になった．Cell/B.E. アーキテクチャには２つの並列化階層，つまり SPE||SPE||PPE のコア間階層と SPU||MFC のコア内階層があり，それぞれの階層に対して違うアルゴリズムと最適化技術がある．その中，本研究はまずコア内の並列化階層に着目し，SPU と MFC の両方の利用率を高めるための２重バッファリングという最適化アルゴリズムを考察しようとした．その考察内容は，Cell/B.E. アーキテクチャの上で実現した２重バッファリングのアルゴリズムの正確さとその有効性，つまり２重バッファリングによってどの程度パフォーマンスを向上できるかということである．そして，考察の手段はモデル検査によって SPU，MFC とその間の通信仕組みを基盤モデルとしてモデル化し，２重バッファリングのアルゴリズムを応用モデルとしてモデル化し，異なる設定の基盤モデルと異なるアルゴリズムの応用モデルを組み合わせて正確さとパフォーマンスを検証することである．

Cell/B.E. アーキテクチャと２重バッファリングのアルゴリズムを提案した枠組みによってモデル化し，正確さとパフォーマンスの検証を行った．結果は，アルゴリズムの誤りの１つ，及び MFC の非決定性によるパフォーマンスの落下各パターンが検出できた．

目 次

第1章　はじめに **1**

第2章　技術的概観 **4**
2.1 Cell/B.E. アーキテクチャ . . . 4
2.1.1 全体の構成 . . . 4
2.1.2 PowerPC Processor Element (PPE) . . . 5
2.1.3 Synergistic Processor Element (SPE) . . . 6
2.2 モデル検査技術 . . . 7
2.2.1 モデル検査の概要 . . . 7
2.2.2 (システムの振舞いの正しさの検証) . . . 8
2.2.3 時間オートマトン . . . 8

第3章　検証問題と既存の検証技術 **9**
3.1 Cell 用プログラムのパフォーマンスの問題 . . . 9
3.2 SPIN . . . 9
3.2.1 モデル検査ツール Spin . . . 9
3.2.2 記述言語 Promela . . . 9
3.3 時間オートマトンの実装 Uppaal . . . 10

第4章　Cell プロセッサ向け検証手法 **11**
4.1 各枠組みの共用基盤：Spin モデルチェッカ . . . 11
4.1.1 システムのモデル化 . . . 11
4.1.2 正確さの検証方法 . . . 12
4.2 待機オートマトンの枠組み . . . 14
4.2.1 待機オートマトンの概念 . . . 14
4.2.2 待機オートマトンの実現 . . . 16
4.2.3 パフォーマンスの検証方法 . . . 19
4.3 時間オートマトンの枠組み . . . 19
4.3.1 時間オートマトンの翻案 . . . 19
4.3.2 ティック付きリ時間オートマトンのセマンティックス . . . 20
4.3.3 リ時間オートマトンの実現 . . . 23

4.3.4 パフォーマンスの解析・検証方法 25

第5章 実験結果と考察 **26**
5.1 実験の設定 26
5.1.1 Cell 用プログラム：画像色反転 26
5.1.2 2 重バッファリングの最適化技術 26
5.1.3 パフォーマンスを影響するパラメーター 27
5.2 待機オートマトンの枠組みによるパフォーマンス検証 27
5.2.1 アルゴリズムの誤りの検出 27
5.2.2 逐次実行の場合と平行実行の場合のパフォーマンス 27
5.3 時間オートマトンの枠組みによるパフォーマンス検証 28

第6章 おわりに **29**
6.1 本研究のまとめ 29
6.2 今後の課題 30

付 録A 用語 **32**

付 録B API **33**
B.1 MFC パッケージ 33
B.2 Wait パッケージ 34
B.3 Clock パッケージ 34

図 目 次

2.1 Cellプロセッサの構成概要 . . . 5
2.2 PPEの構成要素 . . . 6
2.3 SPEの構成要素 . . . 7

4.1 検証モデルの構成 . . . 11
4.2 待機オートマトン． (a) オートマトン， (b) 実行列 . . . 14
4.3 2プロセスのシステムの実行列クラスのパフォーマンス順序 . . . 16
4.4 WA状態遷移，同期化点と同期化 . . . 17
4.5 並行テスト・アンド・セット . . . 18
4.6 リ時間オートマトンの例． (a)MFC， (b)SPU . . . 21
4.7 ティック付きReTAのセマンティックス． (a) 物理的観点， (b) プログラムの観点 . . . 22
4.8 リ時間オートマトンのCTAS . . . 24

5.1 画像色反転の作業分担 . . . 26
5.2 MFCとSPUの処理振舞い，１バッファの場合 . . . 26
5.3 MFCとSPUの処理振舞い，２バッファの場合 . . . 27
5.4 待機オートマトン状態とパフォーマンス変域 . . . 28

表 目 次

4.1 MFC コマンドの実行順序を検証する例 13
4.2 時間オートマトンとリ時間オートマトンの対応関係 21

5.1 ReTA の枠組みによって検証したパフォーマンス 28

第1章　はじめに

最近，複数のコアから構成されたマルチコア・プロセッサの開発の発展により，ハイパフォーマンス計算の技術がPCと家族電気商品の世界までにも普及している．代表的にCellプロセッサ [1] により，普通のプログラマでもマルチコア上のソフトウェア開発ができ，普通の人でもCellプロセッサが入っているPlayStation3でハイパフォーマンスのゲームなどを体験できる．しかし，ハイパフォーマンスのために，Cellプロセッサは複雑なアーキテクチャを持ち，プログラマはそのアーキテクチャの全ての特徴に気を付けながら多くの手動最適化手段を適用してプログラムを作成する必要がある．結局，Cellプロセッサ用プログラムに対して振舞いの正しさの検証とパフォーマンスの解析・視覚化の作業は極めて複雑になり，従来の手法いわゆるテスティング，デバッグ，シミュレーションなどは効果がなくなる．解決策として，本研究はモデル検査技術を用い，プログラムの正確さとパフォーマンスの両方が検証できる枠組みを提案する．

背景　2001年からソニー，SCE，IBMと東芝の四社は共同研究でCell Broadband Engine Architecture(Cell/B.E.アーキテクチャ)を開発開始し [2]，2006年発売したPlayStation3にCell Broadband Engine Microprocessor(Cellプロセッサ)が搭載され[1]，2008年までCellプロセッサは続けて発展していた[2]．Cellプロセッサは異なる2種類の9個のコア，つまり1個のPowerPC Processor Element(PPE)と8個のSynergistic Processor Element(SPE)から構成される．SPEのコアはまたデータ転送を担当するMemory Flow Controller(MFC)とデータ処理を担当するSynergistic Processor Unit(SPU)から構成される．そして，SPUは自分のメモリしか直接アクセスできず，主記憶にアクセスするのにMFCを通じてデータ転送する必要がある．この高度な並列化，高度な役割分担のシステムに対して，プログラムの正確さとパフォーマンスは多くの要素によって影響され，プログラムの振舞いが要求仕様を満たすか，プログラムがシステムの機能を有効に活用できるかなどの性質を検証することは問題になる．

Cellプロセッサと Heterogeneous Multi-core Processor(HMCP) の開発よりもっと前から，モデル検査という従来の検証技術が存在した．モデル検査は，ある検証対象になったシステムに対してシステムの形式モデルと形式要求仕様を作成して自動的に取り扱うことにより，対象システムが要求仕様を満足することを検証する技術の1つである．その中，

[1]Sony Computer Entertainment Inc. の Cell Broadband Engine の技術情報公開ホームページ（2009年1月27日）： http://cell.scei.co.jp/index_j.html

[2]COMPUTERWORLD.jp：IBM、Cellベースのブレード・サーバ「BladeCenter QS22」を発表（2008年05月14日）：http://www.computerworld.jp/topics/ibm/107589.html

1) 形式モデルは有限オートマトンで表現し，2) 形式要求仕様は時相論理で記述し，3) 検証作業は有限オートマトンの全ての状態に対して要求仕様の各論理式を検査することである．ただし，モデル検査を実現するときに 2 つの選択肢があり，Spin モデルチェッカ [?] のように有限オートマトンの各状態を明示して直接メモリに保存し，もしくは SMV モデルチェッカ [?] のように有限オートマトンの状態集合と状態遷移を暗黙にブール論理に変換して二分決定図 (BDD) の形で保管することである．状態の明示表現の方には，モデル検査作業はそれの状態を直接それぞれ検査するので，有限オートマトンを実行パス毎に調べる形になり，検証できる性質は線形時相論理 (LTL) で表現する．一方，状態の暗黙表現の方には，モデル検査作業は有限オートマトンの状態全集合を表す BDD の上の操作によって検査するので，検証できる性質は計算木論理 (CTL) で表現する．その上，実時間の概念と実時間に関する要求仕様を検証するために，有限オートマトンを時間変数と遅延遷移によって拡張した時間オートマトンを用いたモデル検査技術もある．

動機・アプローチ　Cell プロセッサ用のプログラムの正確さとパフォーマンスを有効に検証する要求は本研究の動機になった．Cell/B.E. アーキテクチャには 2 つの並列化階層，つまり SPE‖SPE‖PPE のコア間階層と SPU‖MFC のコア内階層があり，それぞれの階層に対して違うアルゴリズムと最適化技術がある．その中，本研究はまず[3]コア内の並列化階層に着目し，SPU と MFC の両方の利用率を高めるための 2 重バッファリングという最適化アルゴリズムを考察しようとした．その考察内容は，Cell/B.E. アーキテクチャの上で実現した 2 重バッファリングのアルゴリズムの正確さとその有効性，つまり 2 重バッファリングによってどの程度パフォーマンスを向上できるかということである．そして，考察の手段はモデル検査によって SPU，MFC とその間の通信仕組みを基盤モデルとしてモデル化し，2 重バッファリングのアルゴリズムを応用モデルとしてモデル化し，異なる設定の基盤モデルと異なるアルゴリズムの応用モデルを組み合わせて正確さとパフォーマンスを検証することである．

関連研究　モデル検査の分野には，複雑なシステムの振舞いの正しさを検証する代表的なツールとしてSpin モデルチェッカ [?] があり[4]，時間に関するパフォーマンスを検証する代表的なツールとしてUppaal モデルチェッカ [?] がある．しかし，どちらでも複雑なシステムに対して時間に関するパフォーマンスを検証することに最適なツールではない．Spin の方は実時間の概念が載っていないので時間に関するパフォーマンスを直接検証することができない．一方，Uppaal のモデル記述言語は低いレベル，つまり時間オートマトン [?] のレベルでの言語であるので，複雑なシステムとアルゴリズムを記述することは実用性がないと考えられる．

[3]その後にコア間の並列化階層に移動すると予定したが，時間の制限によってこの予定の作業を今後の課題にした．

[4]以上の「背景」のところに述べた通りに，モデル検査のもう 1 つの代表的なツールとして SMV モデルチェッカがあるが，そのモデル記述言語は低いレベル，つまりオートマトンのレベルでの言語であるので，Cell プロセッサのような複雑にシステムに対しては実用性がないと考えられる．

提案手法　本研究はCellプロセッサ用のプログラムの正確さとパフォーマンスの両方を検証できる2つの枠組みを提案する．これらの枠組みは，Cellの複雑なアーキテクチャと最適化アルゴリズムを記述するために，Spinモデルチェッカとそのモデル記述言語Promelaを採用する．そして，パフォーマンスを検証するために，1つ目の枠組みは待機オートマトンのアイドル・ビジー状態の概念，2つ目の枠組みは時間オートマトンの実時間の概念を導入してPromelaとSpinを拡張する．すなわち，1つ目の枠組みは各プロセスのアイドル・ビジー状態のフラグ，2つ目の枠組みは時間変数と時間制御用のモデル化要素をPromelaに追加する．それらの拡張は，Promelaの既存要素のマクロとして，もしくはSpinの生成したCコードのベリファイヤーを編集して実現される．ただし，待機オートマトンの枠組みの検証能力の狭い限界により，本研究には時間オートマトンの枠組みを中心にして開発し，待機オートマトンの枠組みを評価のベースラインとしてのみ取り扱う[5]．

考察結果　Cell/B.E. アーキテクチャと2重バッファリングのアルゴリズムを提案した枠組みによってモデル化し，正確さとパフォーマンスの検証を行った．結果は，アルゴリズムの誤りの１つ，及びMFCの非決定性によるパフォーマンスの落下各パターンが検出できた．

論文の構成　本論文は6章から成る．第1章は研究の全体図を紹介する．第2章は背景としてCell/B.E. アーキテクチャとモデル検査技術を紹介する．第3章は本研究の動機とした検証の要求，及びその関連研究としてSpinとUppaalの２つのモデルチェッカを紹介する．第4章は本研究の提案手法について説明する．第5章は行った実験とその考察について説明する．第6章は論文のまとめと今後の課題を述べる．

[5]本研究の履歴により，最初は待機オートマトンの枠組みを開発しようとしていたが，待機オートマトンを実現した後にその枠組みの狭い限界が分かった．

第2章　技術的概観

2.1　Cell/B.E. アーキテクチャ

Cell（セル，正式名称はCell Broadband Engine）はソニー・SCE・IBM・東芝によって開発されたマイクロプロセッサである.「PLAYSTATION 3」を初め，一部のサーバーやワークステーション，また2009年1月現在，米エネルギー省が保有する世界最速のスーパーコンピューター「Roadrunner」やHDTV受像機などの様々な製品に採用されている.

Cell はマルチコア CPU で，1つの CPU の中に9個のプロセッサコアをもつ．1個の汎用的なプロセッサコアと，8個のシンプルなプロセッサコアを組み合わせたヘテロジニアスマルチコア（ヘテロジニアス：Heterogeneous，非対称，異種混合）．汎用プロセッサコアはPowerPC Processor Element (PPE) と呼ばれ，8個のコアはSynergistic Processor Element (SPE) と呼ばれる．オペレーティングシステム (OS) はLinuxなどをサポートする．また，仮想マシン支援機能が搭載されており，複数の仮想マシン上で複数のOS（ゲストOS）を互いに干渉させること無く走らせることができる．通常のOSが動作するスーパーバイザーモードの上位にハイパーバイザーモードがあり，仮想マシンを管理する最上位OSはこのハイパーバイザーモードで動作している.

PPE，SPE共にPowerPC G5やPentium 4，Athlon 64など高度なアウト・オブ・オーダー実行機能や分岐予測機構を持つCPUと異なり，命令を並び替えたりするような複雑なスケジューリング機構を搭載しないことでコアを単純化し，高クロック化を実現している．そのため，複雑な条件分岐を伴う整数演算能力は最近のパソコン用CPUに比べ劣るが，強力で高度に並列化された演算機能を備え，数値解析やシミュレーション，動画，音声処理などにおいては，複数のコアを並列に動作させることによって，Cellの性能を発揮させることができる.

エンディアン（メモリ配置）順序はPowerPCアーキテクチャではビッグエンディアン，リトルエンディアン双方に切り替え可能だが，Cellではビッグエンディアンで統一している.

2.1.1　全体の構成

ここまで，Cellがヘテロジニアス・マルチコアプロセッサであり，制御系プロセッサコアと演算系プロセッサコアで得意とする処理が異なるということを解説しました．ここからは実際にCellがどのような物理構成になっているかについて解説します.

Cell プロセッサの構成概要を図 2.1 に示します．

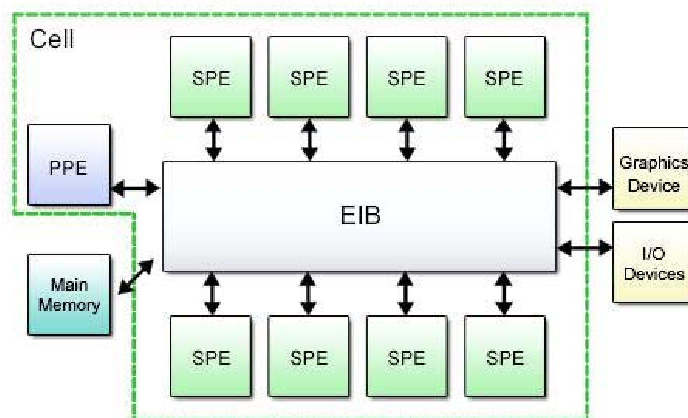

図 2.1: Cell プロセッサの構成概要

Cell は，1 基の制御系プロセッサコア (PPE) と 8 基の演算系プロセッサコア (SPE) で構成されます．各プロセッサコアは，EIB (Element Interconnect Bus) と呼ばれる高速なバスで接続されています．また，EIB はメインメモリや外部入出力デバイスとも接続されていて，各プロセッサコアは EIB を経由してデータアクセスをおこないます．それでは，各プロセッサコアの中身について見ていくことにしましょう．

2.1.2 PowerPC Processor Element (PPE)

Cell は，PPE と呼ばれるプロセッサコアを 1 基搭載しています．PPE は，64 ビット PowerPC アーキテクチャと同等の機能を有した汎用プロセッサであり，従来のプロセッサと同様にオペレーティング・システムやアプリケーションの実行がおこなえます．また，オペレーティング・システムの役割であるメインメモリや外部デバイスへの入出力制御に加えて，SPE を制御する役割も担っています．このように，PPE は処理の制御を主におこなうため「制御系プロセッサ」ともいえます．PPE の構成要素を図 2.2 に示します．

(1) PowerPC Processor Unit (PPU)
PPU は，PPE の演算処理をおこなう核となるユニットで，PowerPC アーキテクチャをベースとした命令セットを持ちます．また，1 次キャッシュとして 32KB の命令キャッシュと 32KB のデータキャッシュを搭載しています．

(2) PowerPC Processor Storage Subsystem (PPSS)
PPSS は，PPU からメインメモリへのデータアクセスを制御するユニットです．また，PPU からのメモリアクセスを高速化させるために 512KB の 2 次キャッシュを搭載しています．

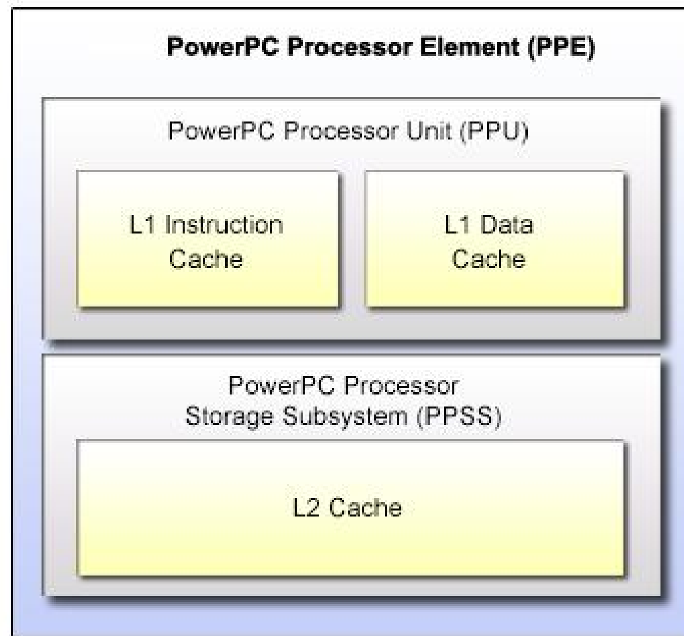

図 2.2: PPE の構成要素

2.1.3 Synergistic Processor Element (SPE)

Cell は，SPE と呼ばれるプロセッサコアを 8 基搭載しています．SPE は，PPE のような複雑なプログラム制御よりも，計算を単純に繰り返すマルチメディア系の処理を得意とする「演算系プロセッサコア」です．Cell は，この 8 基の SPE を有効に活用することにより高い計算能力を発揮します．SPE の構成要素を図 2.3 に示します．

(1) Synergistic Processor Unit (SPU)
SPU は，SPE の演算処理をおこなう核となるユニットで，各 SPE 上に搭載されています．SPU は，PPU とは異なる独自の命令セットを持ちます．また，各 SPU は，LS と呼ばれる専用メモリを搭載しています．

(2) Local Store (LS)
LS は，各 SPU に専用の，容量が 256K バイトのメモリです．また，LS は各 SPU から直接参照できる唯一のメモリでもあります．各 SPU が，メインメモリや他の SPE 上の LS にアクセスするためには，次に解説する MFC と呼ばれるユニットを利用しなくてはいけません．

(3) Memory Flow Controller (MFC)
MFC は，メインメモリや他の SPE などとデータをやり取りするためのユニットです．また，SPU は，チャネルと呼ばれるインタフェースを介して MFC に対してデータ転送などを依頼します．

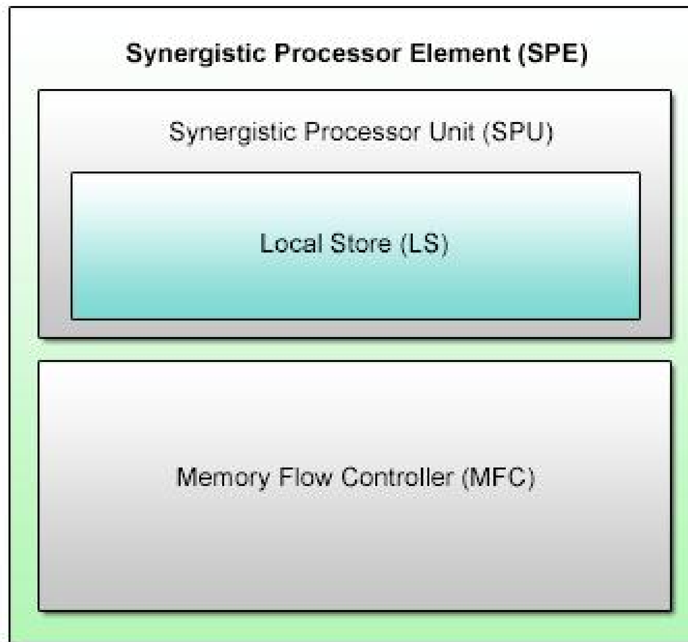

図 2.3: SPE の構成要素

2.2 モデル検査技術

2.2.1 モデル検査の概要

モデル検査（英：Model Checking）とは，形式システムをアルゴリズム的に検証する手法である．ハードウェアやソフトウェアの設計から導出されたモデルが形式仕様を満足するかどうか検証する．仕様は時相論理の論理式の形式で記述することが多い．

モデル検査はハードウェア設計に適用されることが最も多い．ソフトウェアには非決定性があるため，アルゴリズム的な手法だけでは完全ではなく，証明も反証もできない場合がある．

モデルはハードウェア記述言語や専用の言語で記述されたソースコードの形態となるのが一般的である．そのようなプログラムは有限状態機械に対応付けられ，ノード（接点）とエッジから成る有向グラフで表される．原子命題の集合は各ノードと対応し，一般にメモリ要素の内容に対応する．ノードはシステムの状態を表し，エッジはある状態から他の状態への遷移可能性を意味する．その場合，原子命題は実行のある時点で保持される基本的属性を表す．

問題を形式的に表現すると次のようになる．時相論理の論理式 p で表される属性について，初期状態 s のモデル M があるとき，$M, s \models p$ が成り立つかどうかを決定する．ハードウェアの場合のように M が有限であれば，モデル検査はグラフ探索に帰着できる．

実世界の問題を扱おうとしたとき，モデル検査は状態組み合わせ爆発問題（状態の数が

膨大となって検査不能となること）に直面する．これに対応する方法はいくつか存在する．

1. 記号的アルゴリズムは有限状態機械のグラフを作らず，命題論理の論理式によって暗黙のうちにグラフを表現する．Ken McMillan (1992 年）の研究により，二分決定木の利用が一般的となった．最近では，SAT solver （充足可能性問題参照）をグラフ探索に使用するようになってきた．

2. 半順序法は明確にグラフの形式をとっている場合に，考慮すべき並行プロセス群の独立した同時動作の数を削減することができる．考え方の基本は，A と B のどちらが先に実行されるかが証明に無関係なら（AB でも BA でも証明に関係しない場合），それを考慮しないということである．

3. 抽象インタプリタはシステムをまず単純化してから証明しようとする．単純化されたシステムは本来のシステムと等価な属性を持たないので，詳細化の工程が必須となる．一般に抽象化は「健全」でなければならない（抽象化されたシステムで証明された属性は本来のシステムでも真であること）．プログラムの抽象化の例として，ブーリアン以外の変数を無視し，ブーリアンの変数とプログラムの制御構造だけを考慮する場合がある．このような抽象化は劣悪のように見えるが，相互排他の属性を証明するには十分である．

モデル検査は当初，離散状態系の論理的正当性を調べるために開発されたが，リアルタイムシステムや限定された形式のハイブリッドシステムにも対応できるよう拡張されてきている．

2.2.2 (システムの振舞いの正しさの検証)

2.2.3 時間オートマトン

本節では，オートマトンの拡張である時間オートマトンに関して述べる．時間オートマトン間の通信は，同じイベントで時間オートマトンを同期遷移させることにより表現される．実時間システムの仕様記述や，ソフトウェアプロセスの記述において，特に，ソフトウェアプロセスの記述には，同じイベントによる通信でデータのやりとりを表現するがあるため，イベントに入力と出力の区別が必要となる．しかし，Alur らが提案した時間オートマトンでは，イベントにおける入力と出力の区別が存在しない．

第3章　検証問題と既存の検証技術

3.1　Cell用プログラムのパフォーマンスの問題

最適化（2重バッファリング）の有効性非決定性が高いコア間通信が複雑 $p \cap q, q \vee p$ アルゴリズムの正確さの検証も必要

3.2　SPIN

3.2.1　モデル検査ツールSpin

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

3.2.2　記述言語Promela

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

3.3 時間オートマトンの実装Uppaal

UPPAAL はリアルタイムシステムの妥当性検査（グラフィカルなシミュレーションによって）と検証（自動的なモデル検査によって）を行なうツールボックスで，主にグラフィカル・ユーザインタフェースとモデルチェッカーエンジンの２つの機能から構成されています． ユーザインタフェースは Java で実装されており，ユーザーワークステーションの上で実行されます． 実行には Java １．２以上を必要とします． エンジン部分はデフォルトでユーザインタフェースと同じコンピュータの上で実行されますが，サーバー上で実行する事もできます．

時間オートマトンを用いてシステムをモデル化するとは，ミュレートし，その上で特性を検証する事という意味です．時間オートマトンとは時間を持つ有限の状態マシンです．システムはロケーションで構成されているプロセッサーのネットワークから成り立っています． ロケーション間の遷移はシステムの振る舞いを定義しています． シミュレーションステップではインタラクティブに，システムが思い通りに機能するかを調べていきます． また到達可能性をチェックすることができるので，ある状態へ到達可能かどうかがわかります．これはモデル検査と呼ばれ，基本的にシステムの全ての起こりうる動的な振る舞いを網羅的に調査します．

UPPAAL では，標準的な時間オートマトンに対して拡張を行った拡張時間オートマトンを検証モデルとしている．標準的な時間オートマトンに対して UPPAAL 拡張時間オートマトンが追加している主な特徴を以下に示す．

`bounded integer variable` インバリアントやガード制約，変数の代入文に含めることが可能な整数変数．とり得る値の範囲の指定が可能．

`urgent synchronisation` この同期が可能な状態では時間経過が不可能となる同期．

`urgent location` ロケーション上での時間経過が不可能であるロケーション．

`commited location` ロケーション上での時間経過が不可能であり，次の瞬間にこのロケーションから抜け出す必要のあるロケーション．

これらの拡張点のうち，整数変数については抽象化による状態数の削減が可能であると考えられる．しかし，以降で述べる抽象化手法は標準的な時間オートマトンが持つクロック変数の抽象化であり，整数変数の抽象化は 6. の考察で述べている．

第4章　Cellプロセッサ向け検証手法

本章はCellプロセッサ用プログラムの正確さとパフォーマンスの両方を検証する2つの枠組み及びその対応のパフォーマンス解析・検証方法について説明する．これらの枠組みは従来ツールSpinモデルチェッカを基盤にし，プログラムの正確さの検証はSpinの基礎機能を用いて行い，プログラムのパフォーマンスの検証はSpinを拡張して行う．

4.1　各枠組みの共用基盤：Spinモデルチェッカ

4.1.1　システムのモデル化

本研究の検証対象になったシステムは固定のCell/B.E.アーキテクチャとその上で動くCell用のプログラムという2つの部分から構成されるので，図4.1のように2つの部分モデルに分けてモデル化する．

基盤モデル Cell/B.E.アーキテクチャの固有特徴を表現するモデルである．このモデルは応用モデルへのインタフェースを提供する．

応用モデル 各Cell用のプログラムの振舞いを表現するモデルである．このモデルは基盤モデルで提供したインタフェースを用いてCell用のプログラムの振舞いを表現する．

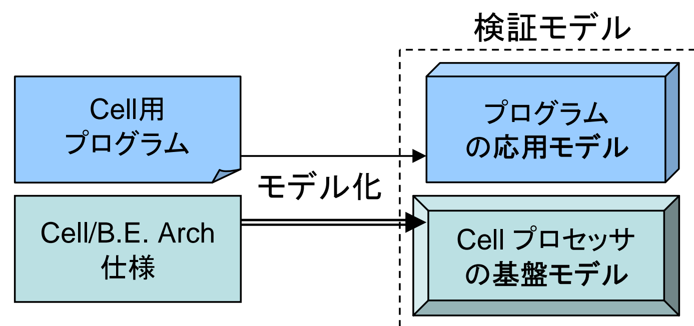

図4.1: 検証モデルの構成

SpinのモデルR記述言語Promelaの要素はプログラミング言語Cの要素と似ているので，応用モデルの作成はほぼCell用プログラムの言語変換に過ぎない．以下は基盤モデルへCell/B.E.アーキテクチャのモデル化方法について説明する．

時間の制限のため，本研究は１つのSPEのみモデル化した．SPEの各要素に対応したモデル化方法は以下のようである．

- 各並列コンポーネント，つまりSPUとMFCをそれぞれPromelaプロセス`SPU()`と`MFC()`でモデル化する．ただし，SPUのプロセスの処理内容はユーザによって定義される．
- MFCコマンドの実行順序の非決定性により，MFCコマンド・キューは必ずFIFOキューではないので，Promelaのメッセージ・バッファが使えなく，１つのコマンドの配列`SPUCmdQ[]`と１つのキュー管理のプロセス`MFCCmdReceiver()`によってモデル化する．
- SPUがMFCコマンドを発行する仕組みをコマンドのランデブー･チャンネル`MFCCmd_ch`を通して`MFCCmdReceiver()`へコマンドのメッセージを送ることでモデル化する．
- MFCのみの振舞いを検査するために，ランダムのコマンドを発行するSPUプロセスを用意する．このプロセスはPromelaの非決定選択によってモデル化する．

4.1.2 正確さの検証方法

通常，システムの振舞いの要求仕様（の否定）をLTLで記述してPromelaの`never`文でモデルに挿入して検証する．しかし，Cellのように複雑なシステムに対して，LTLで記述できる要求仕様が少ない．その代わりに，要求仕様を検査するサブルーチン（Promelaの`inline`）又は検査用プロセスを定義し，その中に届いてはいかない状態を`assert(false)`でマークすることによって検証することが多い．

典型的な要求仕様は，MFCがDMA転送コマンドを実行する順序に関する要求である．MFCコマンドの実行順序は非決定性があるため，SPUが発行したコマンド（`get`，`put`）を間違った順序で実行される恐れがあるので，例えば，

MFCは処理対象データの i 番目の塊をバッファにもらったら（`get(buf,data[i])`），次の塊をバッファにもらう（`get(buf,data[i+1])`）までには，必ず現在のバッファのデータを i 番目の塊の番地へ送っておく（`put(buf,data[i])`）かつその以外のDMA転送コマンドを実行しない	(S4.1)

という要求がある．言い換えると，`get(buf,data[i])`, `put(buf,data[i])`, `get(buf,data[i+1])`, `put(buf,data[i+1])`,...のようにMFCコマンドが実行される必要なことである．ここで，もし処理データの番地を無視して

MFCは`get`コマンドと`put`コマンドを交互に並んで実行する	(S4.2)

という要求に簡略化したら，$Pget =$「`get`コマンドを実行した」，$Pput =$「`put`コマン

ドを実行した」と $Pnil$ =「MFC コマンドを実行していない」という 3 つの原子論理式によって LTL 式

$$\Box[Pget \rightarrow \mathfrak{X}(Pnil\ \mathfrak{U}\ Pput)] \ \wedge \ \Box[Pput \rightarrow \mathfrak{X}(Pnil\ \mathfrak{U}\ Pget)] \tag{4.1}$$

で表現できる．しかし `get(buf,data[i])`，`put(buf,data[i+1])`，`get(buf,data[i+1])`，`put(buf,data[i])`，... という反例があるので，LTL 式 4.1 で表現した要求 S4.2 は不十分である．

表 4.1: MFC コマンドの実行順序を検証する例

```
  1  MFCCmd_t last_MFCcmd;
  2  bool first_MFCcmd = true;
  3
  4  inline check_MFCcmd(last,curr){
  5  if
  6  ::first_MFCcmd -> first_MFCcmd = false;
  7    MFCCmd_assign(last, curr);
  8  ::else -> if
 *9    ::isGet(last) && isPut(curr) && last.ea == curr.ea
*10    ||isPut(last) && isGet(curr) && last.ea + BufSize == curr.ea
 11    -> MFCCmd_assign(last, curr);
*12    ::else -> assert(false);
 13    fi;
 14  fi;
 15  }
-----------------------------------------------------------------
 16  inline MFCPutDelay(n){
 17    check_MFCcmd(last_MFCcmd,pMFC_current_cmd);
 18  }
 19  inline MFCGetDelay(n){
 20    check_MFCcmd(last_MFCcmd,pMFC_current_cmd);
 21  }
```

要求 S4.1 を十分に検証するために，表 4.1 のように，MFC パッケージが提供した API（付録 B.1 を参照），及びユーザが新たに定義する MFC コマンド実行順序の検査サブルーチン `check_MFCcmd()` によって，モデル化する必要になる．すなわち，各 MFC コマンドの監視点 `MFCPutDelay()`, `MFCPutDelay()` に，検査サブルーチン `check_MFCcmd()` を挿入する．また，`check_MFCcmd()` は現在 MFC コマンド `pMFC_current_cmd` と過去 MFC コマンドを保存するユーザの変数 `last_MFCcmd` によって MFC コマンドの実行順序を検査する．その中，処理対象データの番地は有効アドレス ae で表し，行 9-10 では `get` と `put` の順序とそのデータの番地に関する検査項目を記述する．

4.2 待機オートマトンの枠組み

4.2.1 待機オートマトンの概念

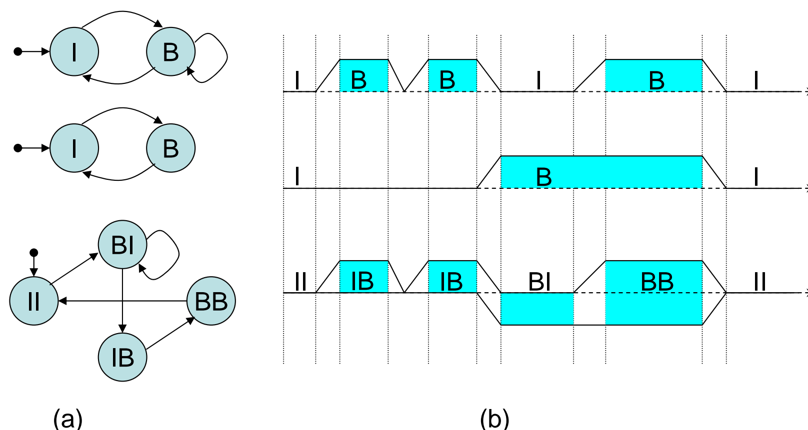

図 4.2: 待機オートマトン． (a) オートマトン， (b) 実行列

「待機オートマトン」(WA) とは図 4.2 単に各プロセスのアイドル・ビジー状態の組み合わせた遷移システムである．従って，１つのプロセスの状態をアイドル（待機中）とビジー（処理中）の 2 つ「安定状態」とその間に遷移している「中間状態」に分ける．物理的に「中間状態」または状態遷移の存在時間は「安定状態」のと比べると比較的に 0 であるので,「中間状態」は低レベルでのモデル化（本研究はPromela）における概念である．その 2 つの状態を持つ n 個のプロセスからなったシステムの状態は 2^n 状態に分け，その間の遷移を含めるとシステムの待機オートマトンが得れる．この枠組みはこの待機オートマトンの上にパフォーマンスを論じる．

定義 4.1 (WA の原子状態). $Q_0 = \{I, B\}$ は待機オートマトンの原子状態の集合である．パフォーマンスに影響するプロセスはこの 2 つの状態，又はその１つを持ち，遷移する．それらのプロセスを**パフォーマンス・プロセス**と呼ぶ．

定義 4.2 (WA の状態のパフォーマンス順序関係). 原子状態集合 Q_0 におけるパフォーマンスの順序関係は

$$I \prec B \tag{4.2}$$

$$p, q \in Q_0. \quad p \preceq q \Leftrightarrow p \prec q \vee p = q \tag{4.3}$$

で定義される．

組み状態 $Q = Q_0^n$ におけるパフォーマンスの順序関係は

$$p_i, q_i \in Q_0. \quad (p_1, p_2, \cdots, p_n) \prec (q_1, q_2, \cdots, q_n)$$

$$\Leftrightarrow \ \forall i \in \overline{1..n}[p_i \preceq q_i] \ \wedge \ \exists j \in \overline{1..n}[p_j \prec q_j] \tag{4.4}$$

$$p, q \in Q_0^n. \quad p \preceq q \Leftrightarrow p \prec q \vee p = q \tag{4.5}$$

で定義される．

定義 4.3 (待機オートマトン). n 個のパフォーマンス・プロセスから構成されたシステムの待機オートマトン W は $W = (Q, T, Q_f)$ で定義される．

- $Q \subseteq Q_0^n$ は状態集合である．システムの状態 q^n は成分のパフォーマンス・プロセスに対応した状態 $q_i^1 \in Q_0$ によって，$q^n = (q_1^1, q_2^1, \cdots, q_n^1)$ で定義される．$q^n = q_1^1 q_2^1 \cdots q_n^1$ と短縮して書くこともある．

- $T \subseteq Q^2$ は遷移の集合である．

- $Q_f \subseteq Q$ は終了状態の集合である．

待機オートマトンの初期状態は全待機状態 $q_0^n = II \cdots$ である．

定義 4.4 (WA の実行列). 待機オートマトン $W = (Q, \delta, Q_f)$ は各状態 $q_i \in Q$ を持つとしたら，$\sigma = (q_1, q_2, \cdots, q_l)$ は W の 1 つの実行列である．$q_1 = q_0 = II \cdots$ かつ $q_l \in Q_f$ であれば，σ は完全実行列と呼ぶ．また，$\sigma = (q_1, q_2, \cdots, q_l)$ の値の集合 $C_s \ni q_i$ を**実行列 σ のクラス**と呼ぶ.

本研究はパフォーマンス・プロセスの個数を 2 個と限定して，パフォーマンスを考察する．定義 4.2 によると 2 プロセスのシステムのの状態における順序関係は半順序であり，IB と BI の間には順序関係がない．しかし，簡単のために本研究は声明 S4.3 を前提とする．

IB と BI のパフォーマンスを一緒とし，両方を代表して IB で記述する． (S4.3)

すると，システム状態のパフォーマンス順序は全順序

$$II \prec IB \prec BB \tag{4.6}$$

になり，実行列クラスのパフォーマンス順序は図 4.3 のように

$$\{II\} \prec \{II, IB\} \prec \{IB\} \prec \{IB, BB\} \prec \{BB\} \tag{4.7}$$

$$\{II\} \prec \{II, BB\} \prec \{BB\} \tag{4.8}$$

$$\{II\} \prec \{II, IB, BB\} \prec \{BB\} \tag{4.9}$$

になる．

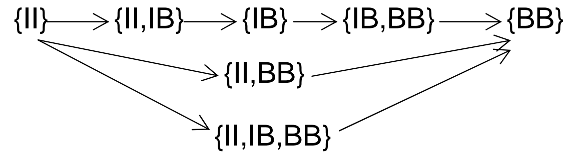

図 4.3: 2 プロセスのシステムの実行列クラスのパフォーマンス順序

4.2.2 待機オートマトンの実現

【モデル化の概要】

待機オートマトンをモデル化するためには，パフォーマンス・プロセス毎にアイドル状態とビジー状態とその間の遷移をモデル化する必要がある．

アイドル状態 プロセスは入力データまたは受信イベントを待っている状態である．それらのことを抽象化して条件 P が成り立つことを待つサブルーチン `waitfor(P)` としてモデル化する．Promela 言語では `(P)->`の文はその同じセマンティックスを持つ．

ビジー状態 プロセスはデータ処理をしている状態である．この枠組みには実時間の概念がないのでこのデータ処理はどの程度時間かかるか分からず，データ処理作業を抽象化して任意の時間で遅延するサブルーチン `delay()` としてモデル化する．Promela 言語では，各プロセスの間の実行選択は非決定的であるので，基本の文，例えば `skip` を実行する前に他のプロセスを任意のステップをかけて実行する可能性があるである．そのセマンティックスは `delay()` の求めたいことと同じである．

WA の状態遷移 状態の遷移はプロセス間通信，システムの構造，アルゴリズムなどによって決まる．それらは他の仕組みによってモデル化され，この枠組みの責任であはない．実際，本研究ではシステムの構造などが SPE の基盤モデルで，アルゴリズムなどが Cell 用プログラム応用モデルでモデル化される．(4.1.1 節を参照)

その上，WA の状態を監視するために各プロセスの WA 状態保管する必要がある．具体的には，各プロセスの `idle` フラグと `busy` フラグを用いて WA 状態を保存する．そして `waitfor()` と `delay()` を拡張し，`waitforf(P,idle)` は `idle`，`delayf(busy)` は `busy` を設定するようにする．

【atomic 文内の delay()】

詳しく考えると `skip` 文だけによって `delay()` をモデル化するのは不十分である．なぜかというと，`delay()` を呼び出すところは `atomic` 文の中にある可能性があり，`atomic` のセマンティックスによって他のプロセスが割り込めなく，`delay()` は 1 ステップで実行完了して「任意時間の遅延」ができなくなる．この問題を回避するために，`delay()` を呼び

出したプロセスの実行を自主的にブロックして実行権を他のプロセスに渡す必要がある．この枠組みは以下のように`delay()`を実現する．

- **自主的ブロック**　ブール型の大域変数`__awake`を定義し，`delay()`のところで`__awake`を`false`にしてからすぐに`__awake`が`true`に成ることを待つ．

```
atomic{ __awake = false; (__awake)-> }
```

- **ブロックの解除**　`__awake`フラグの待つことによってブロックした各プロセスは自分でそのブロックを解除することができない．そのブロックを解除するために，ブロック解除専用プロセス`__Waker()`を定義し，`__awake`フラグを常に`true`に戻す操作を行う．

【waitfor() のフェアネス】

以上の`(P)->`の文による`waitfor(P)`の実現は，待っているプロセスの積極性について考えなかった．従って，

`waitfor(`P`)` は，P が成り立ったら**すぐに**サブルーチンの実行を完了する　(S4.4)

というフェアネスを欠き，P が成り立った後でも`waitfor(`P`)`の呼び出したプロセスに実行権が戻るまでは，任意の時間を経過する場合がある．このフェアネスを保証するために，`waitfor(`P`)`が P の値を検査する操作と他のプロセスが WA 状態を変更する操作を同期化する．

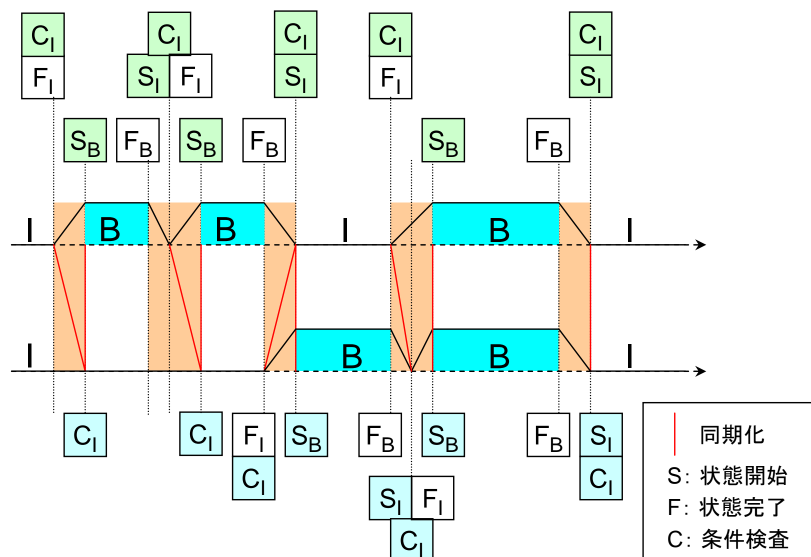

図 4.4: WA 状態遷移，同期化点と同期化

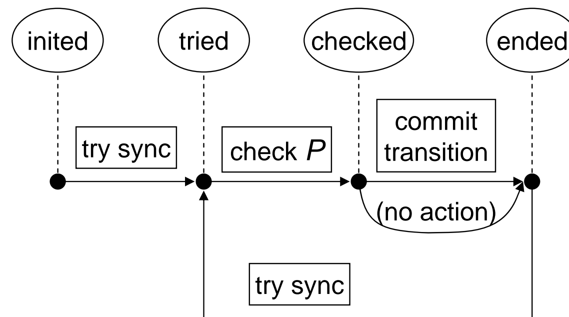

図 4.5: 並行テスト・アンド・セット

- **同期化点の決定**　図 4.4 のように WA 状態が変わる部分は 2 つあり，$I \to B$ の遷移つまり $[F_I, S_B]$ の空間，と $B \to I$ の遷移つまり $[F_B, S_I]$ の空間である．この 2 つの空間の中からそれぞれ 1 つの点を同期化の点として取り扱う必要である．その中，WA 状態の完了点（F_I と F_B）から次の WA 状態の開始点（S_I と S_B）の直前までの空間には P の値を更新する操作が起こりえるのでその直後，つまり S_I と S_B の点でを同期化したほうが安全である．従って，`waitfor`(P) の中で P の値を検査する点，`waitfor`(P) の開始点及び `delay()` の開始点の 3 点を同期化点とする．

- **同期化仕組み：並行テスト・アンド・セット (CTAS)**　同期化点のところでは，`waitfor`(P) の条件 P の検査，かつ WA 状態の変数 `idle`，`busy` の変更という 2 つの作業を同期的に行う必要なので，図 4.5 のように更に `tried`，`checked`，`ended`，という 3 つの下位の同期化点に分けて実現する．`tried` 点で全てのプロセスの CTAS 同期化の開始を待ち，`tried` と `checked` の間で皆は条件 P を検査し，`checked` 点で皆の条件検査の完了を待ち，`checked` と `ended` の間で全てのパフォーマンス・プロセスは P の検査を通したら，皆は WA 状態の変数 `idle`，`busy` を変更し（WA 状態遷移を行い），1 つのパフォーマンス・プロセスは P の検査を通さなかったら何もせずに CTAS 同期化を完了し，`ended` 点で皆の CTAS 同期化の完了を確認する．

  その中，P の検査を通さなくて WA 状態が遷移できないのは，図 4.4 のように，アイドル状態の完了点で P を検査すること $(C_I \equiv F_I)$，又は既に成り立った条件 P に対して `waitfor`(P) を呼び出すこと $(S_I \equiv C_I \equiv F_I)$ などである．その場合，WA 状態の遷移を完了したいプロセスは同期化を再試行する必要である．そのため，この枠組みでは同期化操作を「試行同期化」サブルーチン `trytime`(G) としてモデル化し，`waitfor`(P) と `delay()` は試行錯誤で繰り返して `trytime`(G) を呼び出すことにする．ただし，`trytime`(G) のガード条件 G は `waitfor`(P) の条件 P の反対 $G = \neg P$ である．

4.2.3 パフォーマンスの検証方法

この枠組みでは，パフォーマンスの検証することは各プロセスの idle フラグと busy フラグを用いて LTL 式を書き，応用モデルの never 文に入れて行う．例えば，パフォーマンスが悪い場合「どの実行列でも BB 状態へ届けない」と言う仕様は

$$\Box[(\neg \texttt{pMFC_busy}) \vee (\neg \texttt{pSPU_busy})] \tag{4.10}$$

の LTL 式で表現できる．または逆に，パフォーマンスが良い場合「どの実行列でも BB 状態へ届ける」と言う仕様は

$$\Diamond[\texttt{pMFC_busy} \wedge \texttt{pSPU_busy}] \tag{4.11}$$

の LTL 式で表現できる．

ところが，この枠組みには 2 プロセスのシステムの 4 つの待機状態しかパフォーマンスの情報がないので，表現力の限界はかなり狭い．例えば，式 4.10 より式 4.11 は良いパフォーマンスを表現するが，実際に 4.11 での「届けた BB 状態」の在留時間は全体の経過時間の 1/1000000 であれば，その 2 つの場合のパフォーマンスの差は比較的にないと言える．

4.3 時間オートマトンの枠組み

この枠組みは時間オートマトンの概念（2.2.3 節を参照）をSpin/Promela で実現する．そのため，まず時間オートマトンをSpin に適応させ,「リ時間オートマトン」として再定義する．次はPromela のマクロかつSpin の生成した C コードのベリファイヤーの編集によって，適応させた時間オートマトンを実現する．

4.3.1 時間オートマトンの翻案

この翻案の本旨は時間オートマトンとPromela 言語を役割分担で結合することである．即ち，Promela の部分に非時間的な振舞いを記述させ，時間オートマトンの部分に時間制御の振舞いを記述させる．ただし，ここでは時間オートマトンの定義を使わず，Promela 言語を基礎として時間制御部分を新たに「リ時間オートマトン」として定義する．それで，基礎となった言語を「基礎言語」，その言語で記述した部分（文，式，関数など）を「基礎コード」と呼ぶ．そして，リ時間オートマトンの基底 $\mathcal{F}$ は，基礎言語の基底 $\mathcal{F}_B$ を全部採用して時間変数の集合 $\mathcal{T}$ で拡張したものである[1]．基底 $\mathcal{F}$ の中，気にな

[1]基底 $\mathcal{F}_B$ の拡張を形式に定義するはずである．しかし，そうすると $\mathcal{F}_B$ の各要素の拡張をそれぞれ定義しなければならなくなる．従って，$\mathcal{F}_B$ の形式定義がなければその拡張が定義できない．

る部分は変数集合$\mathcal{V}$とその論理式の集合$\mathcal{B}$である．$\mathcal{V}$は，基礎言語の変数集合$\mathcal{V}_B$と時間変数の集合$\mathcal{T}$を含む．$\mathcal{B}$は，$\mathcal{T}$が入らない基礎言語の従来の要素（$\mathcal{V}_B$など）からなった論理式の集合$\mathcal{B}_B$と，$\mathcal{T}$が入る論理式の集合を含む．

定義 4.5 (リ時間オートマトン). 基底$\mathcal{F}$におけるリ時間オートマトン *(ReTA)*は(N, P, n_0)の組で定義される．

- N　は分岐点（ノード）の集合である．分岐点は，基礎言語で書かれた，時間制御以外の基礎コードの部分を抽象化した概念である．
- $P \subseteq N^2 \times \mathcal{B}^3$　は状態（ピリオド，プレース）の集合である．この状態は物理状態又は安定状態を表現する概念である．ただし，基礎言語の状態の概念と混同しないように，以降はピリオド（時間経過の区間）を用いて記述する．ピリオドは，リ時間オートマトンにおける時間経過を決まる．即ち，ピリオドでは全ての時間変数$t_i \in \mathcal{T}$が同期的に進展し，ピリオド以外では時間が経過しない．そして，ピリオドはまた$(n_{in}, n_{out}, p_{in}, p_{stay}, p_{out})$の組で定義される．
  - $n_{in}, n_{out} \in N$　はピリオドの直前ノードと直後ノードである．実行のとき，コントロールは直前ノードn_{in}からピリオドに入り，またピリオドを出ると直後ノードへ移動する．また，あるピリオドpの直前ノードn_{in}と直後ノードn_{out}に対して，pをn_{in}の直後ピリオドかつn_{out}の直前ピリオドと呼ぶ．
  - $p_{in} \in \mathcal{B}$　はピリオドの入場ガード条件である．この条件が成り立たないとコントロールは直前ノードn_{in}からピリオドに入ることができない．
  - $p_{stay} \in \mathcal{B}$　はピリオドの不変性質である．実行のとき，コントロールが対応のピリオドに滞在する間にはこの性質が常に満足されないとならない．
  - $p_{out} \in \mathcal{B}$　はピリオドの退出ガード条件である．この条件が成り立たないとコントロールはピリオドから直前ノードn_{out}へ出ることができない．
- $n_0 \in N$　は初期ノードである．実行のとき，コントロールはこのノードから始まる．

また，定義4.5で定義したリ時間オートマトンの各概念と従来の時間オートマトンの各概念の対応関係は表4.2で表す.

4.3.2　ティック付きリ時間オートマトンのセマンティックス

分岐点のセマンティックスと基礎言語のセマンティックス　リ時間オートマトンと時間オートマトンのセマンティックスの大きな違う点は，リ時間オートマトンの分岐振舞いが形式に定義されていなく，基礎言語のセマンティックスに依存する．例えば，ある分岐点に対して直後ピリオドの集合が決まったが，実行のときにコントロールがどの直後ピリオドへ移動するか，その選択は決定的か，非決定的かなどは，その分岐点に対応した基礎コード

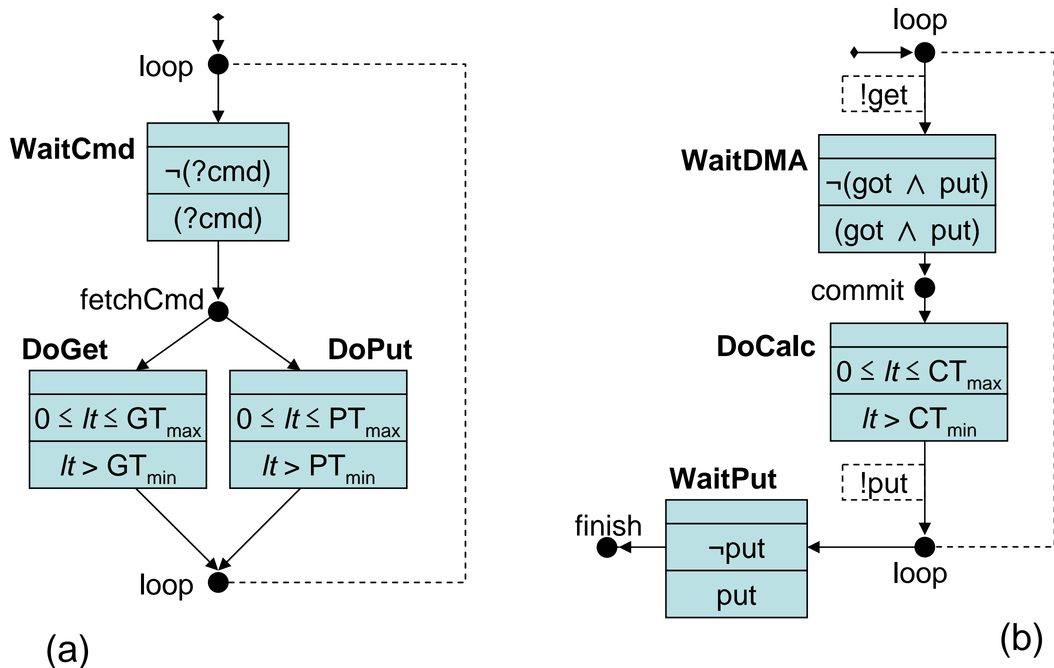

図 4.6: リ時間オートマトンの例． (a)MFC, (b)SPU

表 4.2: 時間オートマトンとリ時間オートマトンの対応関係

時間オートマトン	**リ時間オートマトン**
クロック	時間変数
クロック・リセット	時間変数の一般的な変更
辺	なし.（分岐点とその分岐振舞いのセマンティックス）
なし	分岐点
アクション（辺のラベル）	分岐点で行われる一般的アクション
ロケーション	ピリオド
ロケーションの不変性質	ピリオドの不変性質 p_{stay}
ロケーションから出る全ての辺のガード条件の論理和	ピリオドの退出ガード条件 p_{out}． ピリオドを出たら，コントロールがどの次のピリオドへ移動するのは分岐点に対応した基礎コードに任せる．
なし.（ロケーションの不変性質によるそのロケーションへの遷移可能性のセマンティックス）	ピリオド入場ガード条件 p_{in}

のセマンティックスによって決まる．実際に，図 4.6 における分岐点 fetchCmd の分岐振舞いは MFC コマンド・キューの複雑なアルゴリズムによって決まる．

そして，基礎言語の並行プロセスの概念によって，リ時間オートマトンも並行化され，複数のパフォーマンス・プロセスに対応した各リ時間オートマトンのネットワークを作ることができる．これによって，Cell プロセッサのような平行性が高いシステムを直接モデル化することができる．

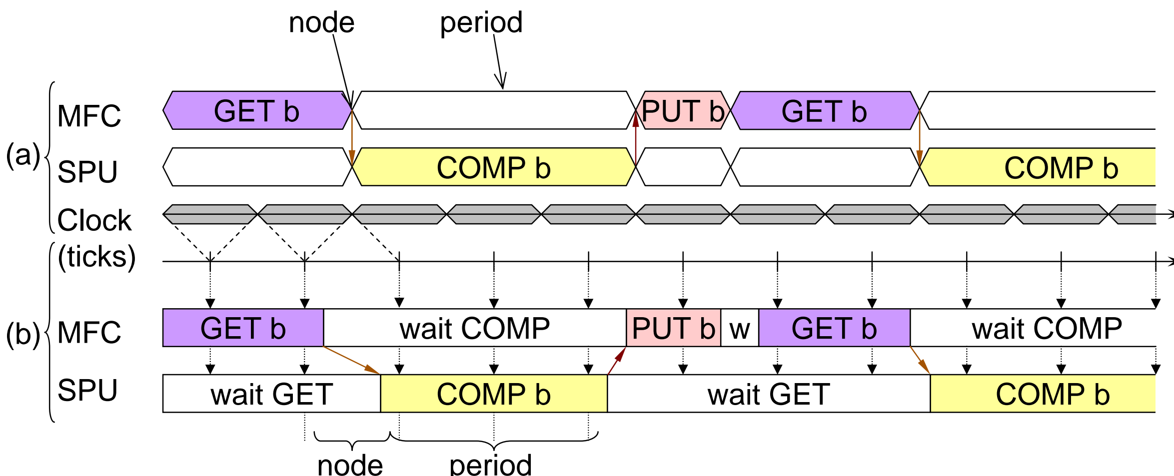

図 4.7: ティック付き ReTA のセマンティックス． (a) 物理的観点， (b) プログラムの観点

ピリオドのセマンティックスとティックのセマンティックス　リ時間オートマトンの時間概念を考えると連続時間、離散時間などが考えられる．だが、本研究は各プロセスの間の相対時間関係を抽象化した「ティック」という概念を用いて時間の進み方を定義する。物理的な観点では，ティックは図 4.7(a) のように経過時間の１つの単位である．この単位は相対的な意味を持つ．例えば，MFC の put，get と SPU の計算の処理時間の比率は $put : get : comp = 1 : 2 : 3$ である場合，それぞれの作業を１ティック，２ティックと３ティックを遅延することでモデル化できる．一方，プログラムの観点では，ティックは図 4.7(b) のように各パフォーマンス・プロセスの時間制御用の同期化点に過ぎない．１つのティックを経過することは，ただ全てのパフォーマンス・プロセスの時間変数が同期的にインクリメントされることである．ただし，ティックの同期化点はアトミックであるので，その点では時間変数のインクリメント以外に副作用がない．それで，以降は用語を乱用して「ティック」を操作として使うときは「ティックの同期化点」,「ティック」を単位として使うときは「ティックの経過時間」の意味とする．

そして，定義 4.5 で述べた通りにピリオド以外には時間経過しないので，ピリオド以外にはティックが発生できない．それで，あるシステムをモデル化する際，殆どの複雑な作業は分岐点で行うが，それらの作業は図 4.7(a) のように時間を消費しない意味を持つ．例えば，図 4.7(b) のように「GET b」と「PUT b」の間に実は１つの「wait COMP」が行

われたが，図4.7(a) のようにそのピリオドが時間的になし（空ピリオド）と見える．この空ピリオドはUPPAAL の緊急プレースに対応する．

時間ロックとゼノ効果　時間経過しない分岐点で任意の作業を行うことができるセマンティックスによって，１つの分岐点で無限の作業が入る，つまり「時間ロック」が起こりえる．この時間ロックは哲学でのゼノの時間矛盾（「ゼノ効果」と呼び）であるので，普通のプログラムのデッドロックのようなエラーとして見なす必要がある．

4.3.3　リ時間オートマトンの実現

リ時間オートマトンを実現するためには，基底を拡張した時間変数の概念とピリオドの概念を実現する必要がある．

【時間変数】

本研究のリ時間オートマトンはティックのセマンティックスを持つので，時間変数の型をPROMELA の `int` 型の別名 `time_t` とする．`time_t` 型で宣言された各変数が，ティックが発生するときに全部同期的にインクリメントされる．そして，時間変数のスコープは大域，つまりプロセス共通のみであり，プロセス局所の時間変数が宣言できない．`time_t` 型の変数と `int` 型の変数の違う点は，その同期的自動インクリメントと大域スコープの２つのみである．ただし，ユーザによって宣言された時間変数のほかに，`delay(min,max,I[,flag])` などの時間制御用命令で用いられる「隠し局所時間変数」がプロセス毎に自動的に宣言される．これらの隠し局所時間変数がユーザに見えない．

【時間制御用命令】

リ時間オートマトンの中心の概念，ピリオドを命令 `ensure(Pin,Pstay,Pout)` でモデル化する．定義4.5で述べたピリオドの組$(n_{in}, n_{out}, p_{in}, p_{stay}, p_{out})$と命令 `ensure(Pin,Pstay,Pout)` の対応は以下のようである．

- n_{in}, n_{out}　は命令 `ensure(Pin,Pstay,Pout)` が呼び出された位置の直前の基礎コードの部分と直後の基礎コードの部分に対応する．
- P_{in}　は命令 `ensure(Pin,Pstay,Pout)` の引数 `Pin` に渡された論理式に対応する．命令の入り口で `Pin` の値を検査し，`Pin` が成り立たなかったら全システムをバックトラックさせる．PROMELA では１つのプロセスをバックトラックさせるのは「end: false;」のように記述できるが，全てのプロセスを一緒にバックトラックさせるのは直接記述できなく，SPIN の編集もかかった命令 `ensure(P)` と命令 `backtrack` を用いる必要になる．(以下を参照)

- P_{stay}, P_{out} 　は命令 ensure(Pin,Pstay,Pout) の引数 Pstay と Pout に渡された２つの論理式[2]に対応する．この２つの論理式を繰り返して検査し，Pout が成り立ったらすぐ命令を実行完了し，Pstay が成り立ったら

その基礎：
trytick(G)
endtick()
これはもう１つの並行テスト・アンド・セットの同期化である.（4.2.2 節を参照）図 4.8 を参照

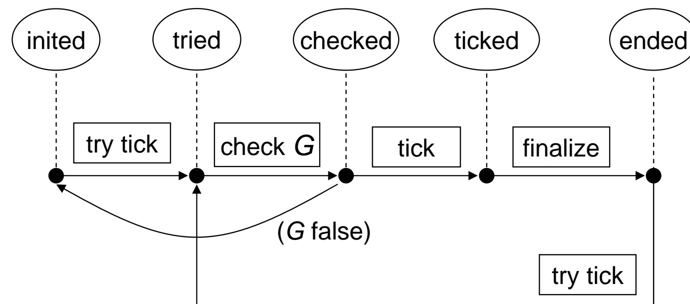

図 4.8: リ時間オートマトンの CTAS

tick()
ensure(P)
ensure(P) は ReTA のピリオドの入場ガード条件のセマンティックスを実現.

その導出：
delay(min,max,I[,flag])
waitfor(P,I[,flag])

各プロセスのティック状態を保存する制御変数 p_tick を自動的に宣言する.
状態爆発問題を防ぐために，大域時間変数と時間制御変数を hidden で宣言する.

【時間制御用プロセス __Clock()】

time_t で宣言した時間変数を自動的にインクリメントする.
各プロセスの並行テスト・アンド・セット同期化を補助する.
状態爆発問題を防ぐために，時間制御コードの全部を atomic 文に入れる.

[2]時間制御用命令はPROMELA のサブルーチンのように名前渡し (call-by-name) のセマンティックスを持ち，引数に渡されたものは式の値ではなく，式そのものである．その値が，サブルーチンの中で引数を参照するときに評価される.

4.3.4　パフォーマンスの解析・検証方法

各コンポーネントの利用率

繰り返しのある作業に対しては，繰り返し回数を抽象化してモデル化する
その繰り返しの中から外へは非決定的なジャンプでモデル化
最後にデッドロック（Promela の timeout）で実行を終了
開始からその時点までの総合時間と各コンポーネントのビジー時間と比べて，パフォーマンスの要求仕様を書き，`assert()` 文で記述
極値のパフォーマンスを表す定数（min, max）を試行錯誤で検索.

エラーのトレールによるパフォーマンスの解析

極値のパフォーマンスを表す定数を試行錯誤で検索すると無駄.
監視したい数量を埋め込み C コードによって検証時に出力し，全部のエラーを検出するオプションで検証し，それらのトレールを解析して各パフォーマンスの数量の関係を導く

第5章　実験結果と考察

5.1　実験の設定

5.1.1　Cell用プログラム：画像色反転

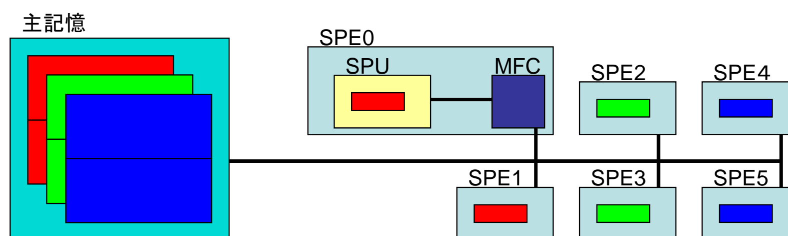

図 5.1: 画像色反転の作業分担

図5.1のように主記憶における画像のデータを BufSize のサイズである塊に分けて SPE の LS にロードし，SPU は繰り返して各データ塊を処理して主記憶に戻す．

5.1.2　2重バッファリングの最適化技術

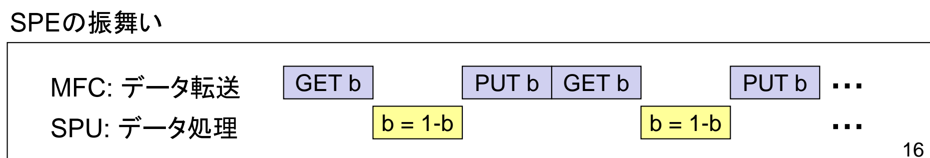

図 5.2: MFC と SPU の処理振舞い，１バッファの場合

図5.2のように１つのバッファのみを利用すると MFC と SPU は互いに待ち，パフォーマンスが悪い．

解決策は，図5.3のように２つのバッファを切り替えて利用し，MFC のデータ転送部分を SPU のデータ処理する部分が重なるようにする．

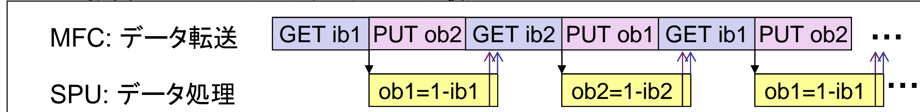

図 5.3: MFC と SPU の処理振舞い，２バッファの場合

5.1.3 パフォーマンスを影響するパラメーター

- **コマンドキューのモード**　FIFO キュー：完全決定的
  Cell チップで実行されたアルゴリズム：一部決定的
  Cell/B.E. 仕様で指定した非決定性の振舞い：完全非決定的

- **get と put の相対の実行時間**　SPU のデータ処理時間と比べ，MFC の get と put のデータ転送時間を相対係数で表現する．
  定数の時間（min=max）
  可変時間 (min から max まで)

5.2 待機オートマトンの枠組みによるパフォーマンス検証

5.2.1 アルゴリズムの誤りの検出

時間にも関わらず 2 重バッファリング技術を実現する最初のアルゴリズム，つまり get 完了のみを待つことの誤りがあるが，コマンドキューのモードを Cell/B.E. 仕様まで非決定性を付けないと問題が出現しない．

5.2.2 逐次実行の場合と平行実行の場合のパフォーマンス

図 5.4 のように，フェアネスの有無によって検証できるパフォーマンス変域が違う．

- **フェアネス無し**　2 重バッファリングはパフォーマンスの変域を拡張だけで，高いパフォーマンスまで届けるが，１バッファの方の最も低いパフォーマンスでも起こりえる．

- **フェアネス有り**　2 重バッファリングと 1 バッファの２つのパフォーマンス変域は交わりがなく，2 重バッファリングの方は絶対 1 バッファの方よりパフォーマンスが高い．即ち，2 重バッファリングの方は，どの実行列にも MFC と SPU の処理時間の重なり（BB 状態）がある．

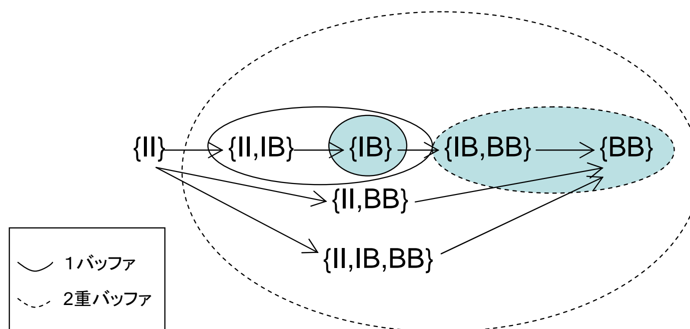

図 5.4: 待機オートマトン状態とパフォーマンス変域

5.3 時間オートマトンの枠組みによるパフォーマンス検証

1バッファと2重バッファリングの両方とパフォーマンスのパラメーターを組み合わせ，それぞれに対してパフォーマンスの解析を行う．
最初の結果は，待機オートマトンの枠組みで保証したことの通りに，2重バッファリングの方は絶対1バッファの方よりパフォーマンスが高い．(5.2節を参照)
以下は，総合処理時間（その最小・最大）をMFCとSPUの処理時間で表す具体結果である．ただし，MFCコマンド・キューのモードはFIFO(det)，Cellチップのアルゴリズム(chip)，Cell/B.E.アーキテクチャの仕様(spec)の3つがあり，$get, put, comp$とは固定サイズのデータ塊に当たってMFCの`get`コマンド，MFCの`put`コマンドとSPUのデータ計算のかかる時間であり，Nとはデータ塊の個数であり，MFC, SPUとはMFCとSPUの総合ビジー時間である．

表 5.1: ReTAの枠組みによって検証したパフォーマンス

buf#	Queue mode	*get*	*put*	*comp*	総合時間
1	det/chip/spec	定数	定数	定数	$T = MFC + SPU$
2	det/chip	5	10	15	$T = get + SPU + put$
2	det/chip	5	9	15	$T = get + SPU + put$
2	det/chip	6	7	5	$T = MFC$
2	det/chip	4	6	5	$T = (comp - get) + MFC$
2	det/chip	6	4	5	$T = MFC + (comp - put)$
2	det/chip	3	4	5	$T = (comp - get) + MFC + (comp - put)$
2	spec	6	7	5	$T_{min} = MFC$ $T_{max} = MFC + \lfloor \frac{N+1}{2} \rfloor comp$
2	spec	[5,6]	7	5	$T_{min} = MFC$ $T_{max} = MFC + \lfloor \frac{N+1}{2} \rfloor comp$

第6章　おわりに

6.1　本研究のまとめ

本研究は，高度並列化された複雑なCell/B.E.アーキテクチャ用のプログラムを対象として振舞いの正確さとパフォーマンスの両方が検証できるようにを，Spinモデル・チェッカと時間オートマトンに基づいて2つの枠組みを作成した．そして，画像色反転というCell用簡単なプログラムを作って検証してみた．プログラムのアルゴリズムの誤りの1つ，つまり同じバッファを同時にデータ転送とデータ処理することを検出した以外に，パフォーマンスに関する検証結果は以下のようである．

1つ目は待機オートマトンを実現した枠組みである．待機オートマトンの限界表現力，つまり7つのパターンしか区別できないという限界よりも，実現のときにフェアネスを注意しないと直感のパフォーマンス性質も検証できない．フェアネスをモデル化した枠組みを用いて検証すると,「2重バッファリングの最適化技術は必ずプログラムのパフォーマンスを常に上げる」という最低限のパフォーマンスの性質を検証できた.

2つ目はり時間オートマトンを実現した枠組みである．この枠組みによって人間が予想しやすい場合，つまりMFCとSPUの処理時間が固定でMFCコマンド・キューの振舞いはFIFOキューである場合に対して，MFCのgetとputコマンドとSPUの計算のかかる時間の比率の各場合に対応するパフォーマンスの性質を各々予想したとおりに検証できた.そして,「MFCコマンド・キューがFIFOキューではないCellチップに対してパフォーマンスが変わってしまう」という恐れを開放して，以上の設定をMFCコマンド・キューの振舞いのみをCellチップのアルゴリズムに変換して各場合を検証してみたら，パフォーマンスの性質が変わらないと分かった．それより，MFCコマンド・キューはCell/B.E.アーキテクチャの仕様で指定したランダムな選択を行うかつ，MFCのgetコマンドの処理時間が$tget_{min}$から$tget_{max}$まで変化するという非決定性が高い場合に対して，簡単に予想できないパフォーマンスの結果が検証できた．その1つは最悪の場合で，2重バッファリングの方のパフォーマンスが半分落下してしまう．もう1つは最良の場合ではgetコマンドの処理時間は$tget_{min}$ではない．これは「可変の処理のgetコマンドをできるだけ速くしたほうが良い」という直感に反して，その処理時間をSPUの処理時間の割合を含めて考えないといけないのである.

6.2 今後の課題

- **時間オートマトンの枠組みを自動化**　??節で説明したコードの自動生成を前処理機械（又はコンパイラ）によって本当に自動的に生成できるように実装を追加する．
- **Cell チップ全体のパフォーマンスの検証実験**　本研究はまだ 1 つのコアしか考察していないが，本当に実用の実験は各コア間の通信も含めたモデルに対してのパフォーマンスの検証である．そして，できるだけそれらのパフォーマンス性質を実際の Cell チップとプログラムで確認した方が良いと考えられる．
- **状態爆発問題を減少**　Partial order reduction のような仕組みで、長いピリオドに対して最初と最後のティックのみを調べ、その中間のティックを抽象化する．

参考文献

[1] Sony Computer Entertainment Incorporated. Cell Broadband Engine アーキテクチャ Version 1.01, 2006 年 10 月 3 日.

[2] 齊藤智隆. Cell と共に歩む. 東芝レビュー 61 巻 6 号，特集：Cell からの始まり, 2006 年 6 月.

付 録A　用語

マルチコア・プロセッサ (英：Multi-core Processor) 複数のコア

ヘテロジニアス・マルチコア・プロセッサ (英：Heterogeneous Multi-core Processor)(略：HMCP)　複数個のコア

モデル検査 (英：Model Checking) 複数のコア

多重バッファリング (英：Multiple Buffering/Multi-buffering) 複数の並列処理ユニットの平行性を向上するために複数のバッファを用いてデータを保管する.

2 重バッファリング (英：Double Buffering) 多重ぶっふぁリングの特別な場合で，2 つのバッファを用いる.

付 録B　API

本章ではCell用プログラムの応用モデル，いわゆる「ユーザ」，へのモデル・インタフェース[1]を記述する.「MFCパッケージ」はSPEの基盤モデル，つまりMFCとそのSPUへの結びの仕組みを提供する．この基盤モデルは待機オートマトンの枠組みと時間オートマトンの枠組みとどちらでも組み合わせて利用できる.「Waitパッケージ」は待機オートマトンの枠組みを提供する.「Clockパッケージ」は時間オートマトンの枠組みを提供する．APIの記述し方について，ユーザが利用できる要素は「•」目印が付いた箇条文で記述し，ユーザによって定義・指定する必要な要素は「◦」目印が付いた箇条文で記述する．

B.1　MFCパッケージ

◦ `SPU()`：SPUのプロセスの処理内容である．

◦ `MFCPutDelay()`：`put`コマンドの実行時に，ユーザが監視する作業の内容である．このサブルーチンは`put`コマンドが実行されるとき毎に呼び出される．

• `pMFC_current_cmd`：MFCは処理しているコマンド，又はMFCは最後に処理したコマンドである．

• MFCコマンドの実行順序の非決定性により，MFCコマンド・キューは必ずFIFOキューではないので，Promelaのメッセージ・バッファが使えなく，１つのコマンドの配列`SPUCmdQ[]`と１つのキュー管理のプロセス`MFCCmdReceiver()`によってモデル化する．

• SPUがMFCコマンドを発行する仕組みをコマンドのランデブー・チャンネル`MFCCmd_ch`を通して`MFCCmdReceiver()`へコマンドのメッセージを送ることでモデル化する．

• MFCのみの振舞いを検査するために，ランダムのコマンドを発行するSPUプロセスを用意する．このプロセスはPromelaの非決定選択によってモデル化する．

[1]Application Model Interfaceの略は「AMI」はずである．しかし，分かり易さのために本論文は「AMI」の代わりに，普及された用語 Application Program Interfaceの略「API」を用いて記述する．

B.2　Wait パッケージ

-
-

B.3　Clock パッケージ

○ MFCPutDelay()：

- pMFC_current_cmd：