データ駆動パラダイムによる実時間システムのプロトタイピング手法に関する研究

工学研究科
筑波大学

2002 年 3 月

我孫子 泰祐

内容梗概

本論文は，筆者が筑波大学大学院博士課程工学研究科電子・情報工学専攻在学中に西川研究室において遂行した，データ駆動パラダイムに基づく実時間処理システムのプロトタイピング手法に関する研究をまとめたものであり，次の 5 章をもって構成している．

第 1 章では，序論として，将来の通信インフラストラクチャを実現するための開発支援手法に関する従来研究の問題点を明らかにする．次に，本研究の目的と，本研究によって得られた成果とについて概説する．

第 2 章では，タグ操作を許すデータ駆動プログラムを開発支援するためのプロトタイピング手法を提案する．本手法は，タグ操作に起因する副作用を検出支援することを主たる目的としている．本手法では，プロトタイピングを介してユーザと仕様記述環境が対話的に，階層的な機能仕様から実行可能プログラムを直接生成する．より具体的には，ユーザが機能仕様として宣言的に記述した階層的なデータ駆動図式 (Data-Driven Schema; DDS) に対し，仕様記述環境が階層構造を展開してデータ駆動型解釈する．これにより動的データ駆動プロセッサの実行可能プログラムの生成に必要な情報を要求・獲得する．さらに，実行可能プログラムの部分的な実行結果を，必要に応じてデータ依存性を畳み込んだ上で，仕様記述上に提示してプロトタイピングする．この仕様記述上への提示を実現するために，本手法は，仕様記述と実行可能プログラムの間のデータ依存性に関する対応関係をつねに維持している．また，プロトタイピングには，従来からプログラム検証でのテストデータの効率的な生成や正当性の証明に使われてきた記号実行を採用している．さらに，データ依存性の畳み込み手法についても，副作用の原因となっているデータとその影響を受

けているデータとの依存関係を高々 6 ノード程度の DDS に畳み込み可能であることを示す.

第 3 章では，前章で述べたプロトタイピング手法を実装した仕様記述環境プロトタイプの実現法を示し，評価する．まず，記号実行の実現法として，汎用インタプリタを活用し，LISP 言語でいう万能関数 eval を用いて記号実行を実現する手法を提案する．次に，対話的なプロトタイピングを実現するためには，記号実行の計算コストの低減が重要となるので，過去の記号実行結果を活用することによって，記号実行のコストを，DDS の一階層の規模程度に抑制する手法を述べる．さらに，本手法を TCP/IP (Transmission Control Protocol / Internet Protocol) プロトコル処理プログラムの中の IP 層受信処理部の開発に適用して評価した結果を示す．その結果から，少なくともユーザが要求仕様を完全に正しく仕様記述に反映するか，または，記号実行結果をユーザが必ず活用する限り，タグ操作による副作用の検出支援のみならず，仕様の抜け・誤りの検出支援にも本手法が有効であることを示す.

第 4 章では，データ駆動型実時間システムの開発を支援するためのプロトタイピング手法を提案する．本手法では，通信処理分野における QoS (Quality of Service) の代表的構成要素として，データ流量ならびにターンアラウンドタイムをとりあげている．まず，ユーザの要求に関する仕様記述手法として，データ流量・ターンアラウンドタイムの仕様記述手法を提案する．また，工学的制約の仕様記述として，データ駆動プロセッサを構成するパイプライン構成の仕様記述手法を提案する．次に，これらの性能仕様・システム仕様に基づくプロトタイピングのための，プログラムの静的解析に基づく性能予測手法とその実現法を述べる．まず，パケットが通過するパイプラインステージと各ステージの平均転送時間とから，実行可能プログラム中のクリティカルパスを求め，ターンアラウンドタイムを予測する手法を述べる．次に，仕様上に記述されたデータ流量の情報と，プログラム構造とを解析して，プログラムの各アーク上を流れるデータ流量を予測する手法を述べる．また，生成されたプログラムの実行時にプロセッサのパイプラインが過負荷とならないことを確認するための，パイプライン占有率を予測する手法を示す.

さらに，音声圧縮・動画像圧縮アプリケーションを例としたメディア処理プログ

ラムへの本手法の適用結果を述べる．その結果，プログラムのボトルネック部分が，ターンアラウンドタイムの予測結果を用いて特定されたことを示す．また，負荷の予測結果を用いて，プログラム分散配置やプログラム構造の改善を支援できたことを示す．これらの改善により，最終的に，要求が満足可能であることが予測されたことを述べる．さらに，本手法にて改善したプログラムを CUE-v1 (Coordinating Users' requirements and Engineering constraints – version 1) の実機で実行した結果，ユーザの期待通りに動作することが確認されたことを述べる．また，性能予測に要する時間の実測結果からは，対話的に性能予測可能との結果が得られたことを示す．以上より，本プロトタイピング手法が，実時間システムの開発支援を対話的に行えるといえる．

第 5 章では，結論として，本研究で得られた成果と今後に残された課題についてまとめる．

関連発表論文

1. Hiroaki Nishikawa and Yasuhiro Wabiko, “A Reuse-Oriented Specification Environment Based on Novel Data-Driven Paradigm,” Proceedings of the Third World Conference on Integrated Design and Process Technology, Society for Design & Process Science, pp. 313-320 (July 1998).

2. Hiroaki Nishikawa and Yasuhiro Wabiko, “Prototype Data-Driven Specification Environment and Its Evaluations,” Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 446-453 (July 1998).

3. Yasuhiro Wabiko, Takuji Urata and Hiroaki Nishikawa, “A Specification and Prototyping Environment for Super-Integrated Data-Driven Processor Systems,” Proceedings of the Fifth World Conference on Integrated Design and Process Technology, Society for Design & Process Science(IDPT 1999), CD-ROM (June 2000).

4. Yasuhiro Wabiko and Hiroaki Nishikawa, “Verification and Emulation Facility for Data-Driven Realtime Processing Systems,” Proceedings of the Fifth World Conference on Integrated Design and Process Technology, Society for Design & Process Science(IDPT 2000), CD-ROM (June 2000).

5. Masashi Ohtsuki, Yasuhiro Wabiko and Hiroaki Nishikawa, “Real-time Execution System for CUE series Data-Driven Processors; RESCUE ,” Proceedings

of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1969-1975 (June 2000).

6. Ryuichi Kudo, Yasuhiro Wabiko and Hiroaki Nishikawa, "Performance Verification Scheme for Data-Driven Real-time Processing," Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1977-1983 (June 2000).

7. Yasuhiro Wabiko and Hiroaki Nishikawa, "A Data-Driven Paradigm to Develop and Tune Data-Driven Realtime System," Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 350-356 (June 2001).

8. 西川博昭, 我孫子泰祐, "タグ操作を許すデータ駆動プログラムの開発・再利用支援手法," 電子情報通信学会論文誌 D-I （2002 年 3 月掲載予定）.

9. Yasuhiro Wabiko and Hiroaki Nishikawa, "Performance Prediction and Verification Environment for Super-Integrated Data-Driven Processors; RESCUE," Transactions of the SDPS: Journal of Integrated Design and Process Science (to be published).

目 次

内容梗概

関連発表論文

1　序論　13

2　機能仕様からのデータ駆動プログラムの開発・保守支援手法　21

2.1　緒言 21

2.2　仕様記述・実行可能プログラムの階層構造と副作用検出支援 22

2.3　仕様記述と実行可能プログラムのデータ依存性に関する対応関係の維持 23

2.4　記号実行に基づくプロトタイピング手法 27

2.5　仕様記述の階層構造の再構成手法 29

2.6　結言 30

3　データ駆動型仕様記述環境の実現法とその評価　31

3.1　緒言 31

3.2　記号実行に基づくプロトタイピングの実現法 32

3.3　データ駆動型仕様記述環境 41

3.4　プロトコル処理プログラムへの適用を通じた評価 44

3.5　結言 50

4　データ駆動型実時間システムの性能評価支援手法　51

4.1　緒言 51

4.2 ユーザの要求・工学的制約の仕様記述 52
4.3 仕様記述環境による性能予測とエミュレータによる性能検証 56
4.4 音声・動画像圧縮プログラムへの適用を通じた評価 76
4.5 結言 77

5 結論 **79**

謝辞 **85**

参考文献 **87**

A RESCUE のシステム構成 **99**
A.1 RESCUE の外部システム構成 99
A.2 RESCUE の内部システム構成 101

B 動画圧縮アプリケーションの仕様記述例とプログラム **103**
B.1 RGB 形式から YCrCb 形式への変換処理の仕様記述例 103
B.2 生成されたプログラムのアセンブラソースコード 107
B.3 RGB 形式から YCrCb 形式への変換処理のターンアラウンドタイム・データ流量の性能予測例 110
B.4 RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 114

C プロトコル処理アプリケーションへの RESCUE の適用例 **119**
C.1 TCP/IP プロトコル処理のチェックサム計算の仕様記述例 119
C.2 ボトルネックとなっている PE の検出とボトルネックの解消例 119

図目次

2.1 データ駆動図式の構成要素 24
2.2 多面的な仕様記述と実行可能プログラム間のデータ依存性に関する対応関係の維持 25
2.3 あらゆる種類のデータ依存性を含む DDS 30

3.1 プリミティブ・部分プログラムとその記号実行規則 35
3.2 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (1) . . 38
3.3 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (2) . . 39
3.4 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (3) . . 40
3.5 データ駆動型仕様記述環境の内部構成 41
3.6 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (1) 45
3.7 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (2) 46
3.8 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (3) 47

4.1 ユーザの要求に関する仕様記述 54
4.2 工学的制約に関する仕様記述 55
4.3 RESCUE のシステム構成 57
4.4 RESCUE の内部構成 58
4.5 ターンアラウンドタイムの予測手法 60
4.6 ターンアラウンドタイム予測コストの低減法 65
4.7 データ流量の予測ルール 68
4.8 命令プリミティブ 1 個の実行時のパイプライン占有率の予測手法 . . . 70

4.9 パイプライン占有率の予測例 71
4.10 重ね合わせの計算コスト低減法 74

A.1 RESCUE の外部システム構成 100

B.1 RGB 形式から YCrCb 形式への変換処理 (1) 全体像 104
B.2 RGB 形式から YCrCb 形式への変換処理 (2) パケット複製処理 105
B.3 RGB 形式から YCrCb 形式への変換処理 (3)YCrCb 生成処理 106
B.5 RGB 形式から YCrCb 形式への変換処理の性能予測例 (1) 全体像 . . . 111
B.6 RGB 形式から YCrCb 形式への変換処理の性能予測例 (2) パケット複製処理 112
B.7 RGB 形式から YCrCb 形式への変換処理の性能予測例 (3)YCrCb 生成処理 113
B.8 RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (1) チップ OCP 32 115
B.9 RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (2) チップ OCP 33 の PE INT/MUL 116
B.10 RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (2) チップ OCP 33 の PE TBL 117

C.1 チェックサム計算処理の性能予測・検証の適用例：仕様記述 120
C.2 チェックサム計算処理の性能予測・検証の適用例：ボトルネックの検出 (1) 121
C.3 チェックサム計算処理の性能予測・検証の適用例：ボトルネックの検出 (2) 122
C.4 チェックサム計算処理の性能予測・検証の適用例：ボトルネックの解消 (1) 123
C.5 チェックサム計算処理の性能予測・検証の適用例：ボトルネックの解消 (2) 124

表目次

3.1 仕様記述環境におけるユーザの操作と環境からの応答との対応関係 . 43
3.2 プロトタイピング手法の評価 49

A.1 RESCUE の実現に利用したツール・ライブラリ 101

第 1 章

序論

将来の通信インフラストラクチャには，ユーザの要求に合わせたより多様なサービスを柔軟かつ迅速に提供することが期待されている [1]-[4]．このような通信インフラストラクチャにおいては，音声や動画像等の複数のメディアを，個々の時間制約を満足しつつ同時に処理するために，実時間多重処理が要求されると考えられる．このような通信インフラストラクチャを実現するために，並列・分散処理技術を活用したマルチメディアネットワーキング環境が広く研究されている [5]-[9]．

ネットワーキング環境を構成するノードは，ユーザとの接点となる端末ノードと，端末ノード間を接続するための基幹ノードの 2 種類に大別できる．ネットワーキング環境とユーザの接点となる端末ノードを実現する上では，消費電力やチップ面積の制約が強い．これらの制約下では，従来のノイマン型汎用プロセッサは不向きである [10]．事実，端末ノードの実現には，以前から ASIC (Application Specific Integrated Circuit) や FPGA (Field Programmable Gate Array) 等のハードウェアによる実現法が採用されてきた [11]-[13]．

これらのハードウェアによる実現法は，高い性能を得られる反面，柔軟性に乏しい．近年では，VLSI (Very Large Scale Integration) 実装技術の進歩による高性能化を背景にして，プロセッサによる実現法が採用されてきている．例えば，音声処理や動画像処理向けに設計されたプロセッサである DSP (Digital Signal Processor) さらにはメディアプロセッサやネットワークプロセッサ等の ASIP (Application Specific

Integrated Processor) が注目されている [14]-[21]．静的データ駆動原理に基づくデータ駆動プロセッサを用いてデジタル信号処理システムを開発しようとする研究もある [22]．経済専門家の間には，これらの DSP やネットワークプロセッサといった特定用途向けプロセッサと従来の汎用プロセッサとが 2002 年に同程度の市場規模になり，さらに 2003 年以降は特定用途向けプロセッサの市場規模が汎用プロセッサの市場規模を追い越すと予測している者もいる [18].

一般に端末ノードは基幹ノードと比較して台数が桁違いに多いので，システムに不具合が発生した際の修理が困難である [23]．特定用途向けプロセッサによる実時間多重処理システムの開発さらには保守を効率化する手法の確立が急務となっている所以である [2][24]．一般にシステム開発コストの大部分は，初期開発でなく保守に費やされるといわれている [25][26]．特に，既存のシステム部品の改変を伴う再利用時には，その副作用の検出が重要である [27][28].

これまで副作用検出が困難とされてきたのは，プログラム実行時に生じた副作用の発生箇所と内容を把握することが困難であったことに起因する [29]．複数の副作用の相互作用で副作用が隠蔽され一見正しく動いているかのように見える場合がある．また，仕様記述がユーザの要求を正確に反映していない場合がある．これらを考慮すると，副作用を検出するためには，プログラムが部分的に改変された時はつねにプログラムを実行して，その結果をユーザに提示しなくてはならない．従って，プロトタイピングは必須である.

市川らは [30]，自然言語風仕様記述言語 TELL/NSL を用いて記述された仕様に基づいて Prolog プログラムを生成し，これを実行してテストするラピッドプロトタイピング手法を提案した．しかしこの手法ではプログラムの実行結果はプログラムの出力データとして生成されるのみであるため，プログラムの正当性を検証するには出力データの内容を吟味する他ない．このように，プログラムの実行結果をプログラム水準で提示されてもユーザには理解が困難である．従って，プログラムの実行結果は仕様記述上に提示されるのが望ましい.

プログラムの実行結果を仕様記述水準に提示するためには，仕様記述とプログラムの間で，データ依存性に関する対応関係が維持されていることが前提となる．よく

知られているように，プログラムの実行機械として従来のノイマン型プロセッサを利用する限り，宣言的に書かれた仕様記述と逐次的に実行されるプログラムの間には隔たりが存在する．例えば SADT (Structured Analysis and Design Technique)[31] は，計算機のいかなる支援を受けようとも，要求定義からプログラム生成へシームレスに移行し得ないことが提案者自身によって指摘されている [32].

仕様は要求を表現したものであるから人間が記述する他ない．しかしプログラムは仕様を基に生成されればよく，プログラムを人間が記述する必要性は本来ない．仕様記述過程を通じてその仕様を実現するプログラムが生成されていき，なおかつプロトタイピングを用いてプログラム検証がつねに行われるような支援環境が望ましい．このような環境を用いれば，仕様の記述が終了した段階でユーザの要求を満足するプログラムが既に生成されているという枠組が構築できる．

ノイマン型プロセッサの利用を前提とした従来の開発手法の多くは，仕様記述とプログラムの双方を人間が記述するものであった．そのため，例えば従来の逐次型プログラミング言語で記述されたプログラムを人間が改変した際の副作用波及範囲を明示するための，プログラムスライシングやプログラム依存グラフが提案されている [33][34].

仕様とプログラムを人間が記述する立場をとると，仕様と実行可能プログラムとの一貫性の維持が問題となる．従来手法の多くは，仕様記述と実行可能プログラムとの一貫性を主として人間の手で維持するものである．しかし現実には，納期が迫ってくるとプログラムのソースコードのみを修正し，仕様の修正が後回しにされる結果，仕様とプログラムの一貫性の維持が困難となっていることが少なくない [35]．例えば通信分野では，国際標準化団体 ISO (International Organization for Standardization) においてテスト仕様記述言語 TTCN (Tree and Tabular Combined Notation)[36] が標準化され広く利用されている．例えば，構内交換機のプログラムを改変した際のテストを TTCN を用いて自動化する手法が提案され，テストのコストが低減されたという報告がある [37]．しかし同時に，TTCN 自身の文法が複雑で理解が困難であるために，通信プロトコルの仕様が変更された場合に TTCN によるテスト仕様を書き直すコストが問題点として指摘されている [38].

また，藤井らは [35] HCP (Hierarchical and ComPact description chart) チャート [39] で記述された仕様と C 言語で記述されたプログラムソースコードとの間での相互変換を実現した．また，Slomka らは [40]，SDL (Specification Description Language) と MSC (Message Sequence Chart) で書かれた仕様を基にして，C 言語によるプログラムソースコードと VHDL (Very high speed integrated circuit Hardware Description Language) によるハードウェア記述を生成する手法を提案している．Slomka らは，このシステムをマルチメディア情報端末向けの音声処理・UDP (User Datagram Protocol)[41]/IP (Internet Protocol) [42] プロトコル処理システムに適用した結果，性能のボトルネックの 1/3 を解消できたと報告している．Malik らの MESCAL (Modern Embedded Systems, Compilers, Architectures and Languages) プロジェクト [43] では，状態遷移図やデータ駆動図式 (Data-Driven Schema; DDS)[44] に基づく仕様から C 言語によるプログラムソースコードと VHDL によるハードウェア記述を生成する手法を提案している．しかし，藤井らの研究は，プログラムの実行結果を仕様上に提示することを想定していない．Slomka らの研究は，性能評価結果を MSC 上に提示する．しかし，SDL による機能仕様に関しては，SDL シミュレータによるデッドロック・ライブロックの検出に止まっている．即ち，プログラムの実行結果を SDL 上に明示することを想定していない．Malik らの MESCAL プロジェクトも Slomka らのプロジェクトと同様であり，性能評価結果を仕様上に提示するものの，機能仕様に基づく副作用検出支援のためにプログラム実行結果を仕様上に提示することは考慮されていない．

実行機械としてのノイマン型プロセッサとその上の高級言語の利用を前提とする限り，仕様記述に基づいてノイマン型プロセッサ上の実行可能プログラムを生成する過程には，コンパイラの介在が必要不可欠である．前述の藤井ら，Slomka ら，MESCAL プロジェクトのいずれにおいても，プログラムの生成にはコンパイラが必要である．このコンパイルの逆変換は一般に定義できない [45][46]．従って，プログラム実行結果を仕様上に正確に明示するのが不可能となる．

以上より，システム開発における副作用検出が従来困難とされてきた原因の本質は，実行機械として暗黙のうちにノイマン型プロセッサを前提とするが故に，宣言

的に書かれた仕様とノイマン型プロセッサ上で逐次的に実行されるプログラムとの乖離が不可避な点にあると捉えるべきである．

再利用性を向上させシステム開発コストを低減するには，並列処理の仕様は，逐次処理向けに設計された表記法を並列処理にも拡張したものでなく，並列性を自然に表現できる図的な表記法を用いて記述されるべきである [47]．HCP のように制御依存性を明示する記法は，設計対象システムの機能が，実行順序に意味がある処理の組み合わせで表現できるときには適している．しかしメディア処理やプロトコル処理の多くのように，システムの機能が入力データから出力データへの段階的変換として表現できる場合もある．このような場合には，DDS のように，データ依存性を明示する記法が向いている [48]．また，要求分析からシステム設計，プログラム設計のどのフェーズにおいてもデータ依存性は本質的な情報である．本研究は，本質的な情報は明示されるべきであるから，仕様記述にはデータ依存性が明示されるべきであるとの立場をとっている．

従って本研究は，並列処理を自然に表現できる図的な記法のひとつである DDS に注目している．DDS は，従来より並列処理システムの仕様記述の手段として広く利用されてきた [25]．例えば，SADT や 構造化分析 (Structured Analysis; SA)/構造化設計 (Structured Design; SD) 等がよく知られている [49]．また DDS は，従来から並列プログラミング言語としても広く利用されてきた [48][50] のみならず，データ駆動プロセッサ上の実行可能プログラムの表現法でもある．例えば，プログラミング言語として書かれた DDS を基にして，データ駆動プロセッサを想定した実行可能プログラムを生成し，これをノイマン型汎用プロセッサ上のシミュレータ上で実行するという手法が提案されている [51]．この手法を，交換機の開発に 10 年にわたって利用した結果，開発コストを従来の半分に低減したと報告されている．しかしながら，プログラムをシミュレータで実行することから実行効率が従来の 50% 程度に過ぎないことも同時に指摘されている．

ここで，DDS が，仕様記述と実行可能プログラムの双方を表現できることに着目する．仕様として書かれた DDS から，データ駆動プロセッサ上の実行可能プログラムを生成することを前提とする．すると，仕様として記述された階層的な DDS

の階層構造を展開したのと同一のデータ依存性を持つプログラムを生成することによって，仕様からプログラムを，コンパイラを介さず直接的に生成することが可能となる．これにより，仕様記述と実行可能プログラムとの間のデータ依存性に関する対応関係は維持される．さらに DDS は，プロセッサのパイプライン構成の表記法としても利用できる．命令セットあるいはプロセッサ割当を介して，パイプライン構成と実行可能プログラムにも対応関係が維持できる．これによって，仕様記述・実行可能プログラム・パイプライン構成のすべてを，それらの間のデータ依存性に関する対応関係を維持しつつ統合的に取り扱うパラダイムが構築できる．

我々の研究室では，将来の通信インフラストラクチャのありかたが少なくともユーザの要求と工学的制約に依存して決定されるという視点から，CUE (Coordinating Users' requirements and Engineering constraints) プロジェクト [52] を遂行してきた．これまでに，TCP(Transmission Control Protocol)[53]/IP や，CORBA (Common Object Request Broker Architecture)[54] に基づく分散オブジェクト環境における IIOP/GIOP (Internet Inter-ORB Protocol / General Inter-ORB Protocol) 等のプロトコル処理のデータ駆動型実現を通じて，チップマルチプロセッサ型動的データ駆動プロセッサ CUE が多重処理を実行時のオーバヘッドなく効率的に実行できることを実証してきた [55][56]．

本研究は，DDS が仕様記述・実行可能プログラム・パイプライン構成のすべてを表現できる事実に着目して，仕様記述から実行可能プログラム，さらにはプロセッサのパイプライン構成に至るまでの統一的なデータ駆動パラダイムに基き，CUE プロセッサによる実時間システムの開発・保守支援手法を確立しようとするものである．本論文では，開発・保守支援に必須のプロトタイピング手法を明らかにし，本手法を開発・保守支援環境の機能として実現して，その有効性を実証することを目的とする．

本研究は，仕様記述として，機能仕様のみならず，マルチメディアを取り扱うために本質的である性能・実時間性に関する要求を表現する性能仕様まで含めてその対象としている．機能仕様には，データ構造，データ依存性，およびプリミティブ割当が含まれる．性能仕様には，データ流量，ターンアラウンドタイム，パイプラ

イン構成，およびプロセッサ割当が含まれる．これらの仕様に基づいて生成する応用プログラムとしては，TCP/IP や 分散オブジェクト環境のプロトコル処理プログラムならびに音声や動画像のメディア処理プログラムを生成することを想定している．これらのうち本稿では，性能仕様を満足するためには機能仕様が満足されていることが前提となるため，まず機能仕様のみを対象にして，副作用検出支援手法を検討した．応用プログラムとしては，TCP/IP プロトコル処理部の中の IP 層での受信処理部をとりあげた．次に，機能仕様に加えて性能仕様を仕様記述の対象とした．応用プログラムとしては音声・動画像処理をとりあげた．

本研究が採用している動的データ駆動型実行方式は，履歴依存性を許すデータ駆動図式 D^3L (Diagrammatical Data-Driven Language)[57] に基いている．本方式では，ストリームをトークン列として実現し，同一アーク上の個々のトークンを識別するために，従来の色 [58] でなく順序の概念を持つ世代 [57] と呼ばれるタグを付与する．これにより，TCP/IP プロトコル処理のデータ駆動型実現法 [55][56] では，IP データグラムや TCP セグメントを 1 バイト単位で並列処理している．後に 3 章で述べるように，本方式では実行時にタグを書き換える操作を許す．この操作を以下ではタグ操作または世代操作と呼ぶ．

SISAL (Streams and Iteration in a Single Assignment Language)[59], Id (Irvine Dataflow)[60] 等の従来のデータフロー言語では，言語仕様の関数性からもたらされる決定性が重視されていた [61]．そのためこれらの言語では単一代入則が堅持されていたので，履歴依存処理はあくまでも関数的言語仕様を基礎とした上で検討された．例えば，Gaudiot は，ヒープで表現された配列の各要素を指し示すポインタをトークンにタグとして付与し，このタグを実行時に排他的に書き換えることで非決定性を回避するマクロアクターを提案した [62]．Arvind らは，単一代入則の堅持をメモリ側の機能として実現する I-構造を提案した [60]．また，データフロー言語を含めた従来の並列プログラミング言語の多くは，並列性を暗黙的に記述する立場をとっていた [58][63]．従って，これらの言語で書かれた並列処理プログラムを効率的に実行するにはコンパイラによる並列性抽出が必要不可欠であった．本研究では，データ駆動原理の本質は，処理の関数性でなく，命令実行がデータ依存性によって決定

される点であると捉えている．従って筆者は，副作用の有無は本来的にあくまでも仕様記述にてらして議論すべきものであり [64]，履歴の改変やそれにより生じる非決定性の有無に対して定義されるべきものではないとの立場をとっている．

本論文の構成は以下のようである．第 2 章では，記号実行 [65] に基づくプロトタイピングを介して対話的に，階層的仕様記述から実行可能プログラムを直接生成する手法を提案する．第 3 章ではまず，記号実行に基づくプロトタイピング手法を実装したデータ駆動型仕様記述環境 [66]-[68] の実現法を述べる．次に，本仕様記述環境を，TCP/IP プロトコル処理の中の IP 層受信処理部の開発に適用して，評価する．その結果，動的データ駆動原理においてトークンに付与されたタグ [58] である世代 [57] の操作に起因する副作用が検出されたことを示す．さらに，従来検出が困難とされてきた仕様の抜け・誤りが検出されたことから，本手法が Verification[69][70] にも有効なことを示す．第 4 章ではまず，データ流量・時間制約といった性能・実時間性に関する要求を含めた性能仕様に基づいて実時間システムを生成するためのプロトタイピング手法を提案する．次に，その手法を実装した実時間実行システム RESCUE (Realtime Execution System for CUE-series data-driven processors)[71]-[75] の実現法を述べる．最後に，メディア処理の具体例として音声・動画像の圧縮を取りあげ，RESCUE の効果を評価する．その結果，RESCUE の対話的な性能評価支援機能が，プロセッサアーキテクチャのチューニングにも有効なことを示す．第 5 章では，本研究で得られた成果と今後に残された課題をまとめる．

第 2 章

機能仕様からのデータ駆動プログラムの開発・保守支援手法

2.1 緒言

本章では，タグ操作を許すデータ駆動プログラムの開発を支援するためのプロトタイピング手法を述べる．本手法は，プロトタイピングを介してユーザとシステムが対話的に実行可能なデータ駆動プログラムを直接生成することで，副作用を検出支援するものである．このプロトタイピングは，従来からテストデータの生成に活用されてきた記号実行に基いている．まず，本研究では仕様記述の階層構造を必要に応じて再構成することを前提にすることを述べる．次に，副作用検出を支援するためには，仕様記述とプログラムのデータ依存性に関する対応関係を維持することが重要であることを述べる．最後に，副作用を仕様記述上に提示する際にユーザの了解性を維持するために必須である，データ依存性の畳み込み手法を示す．

2.2 仕様記述・実行可能プログラムの階層構造と副作用検出支援

構造化分析 [31] やオブジェクト指向分析 [25] を例にして，仕様記述には，従来より階層構造が導入されてきた．事実，心理学者の間では，同時に対象とする個数がマジカル数 7 ± 2 程度を越えると，人間の情報処理能力が急激に低下することが広く知られている [76][77]．例えば，浜田らは，詳細化作業中の機能モジュールの個数がマジカル数を越えると仕様の抜けが発生しやすくなることを指摘している [78].

仕様記述の階層構造は，問題や実行可能プログラムに対して一意に定まるものではない．あくまでもある見方に従った仕様記述過程を反映して生成されたものであって，本質的に恣意的である．従来の研究には，階層構造が恣意的であるが故に，階層構造を詳細化する基準を代数的手法によって定式化しようとする試みがある [79][80]. しかし本研究では，仕様記述の階層構造を必要に応じて再構成し，多面的な仕様記述を許すことが重要であると捉えている．

これに対して，実行可能プログラムは，実行時のオーバヘッドを極力排除するためにインライン展開することもあれば，プログラム記憶メモリの容量の制約等から階層化することもある [81]．例えば，MIT の TTDA (Tagged-Token Dataflow Architecture)[58] での call, apply コンストラクタがこのような階層化にあたる．既に述べたように，本研究では，実行可能プログラムは仕様から直接的に生成する，即ち，ユーザが直接プログラムを記述しない立場をとっている．従って，実行可能プログラムの階層構造は了解性の問題とは直接関係ないので，仕様記述から生成されるプログラムはインライン展開されていることを前提とする．

ユーザの思考過程は，トップダウン・ボトムアップが混在する場合が普通である．例えば，従来のプログラミングにおいては，トップダウンとボトムアップのどちらか一方のみに基づいて開発するのではなく，トップダウンとボトムアップを併用すると効率がよいことが知られている [82]．従って，階層的な仕様の記述過程をユーザの思考に馴染むものとするためには，トップダウン・ボトムアップのいずれかのみに限定したアプローチは制約が強過ぎる．同時に，仕様記述過程がプログラムの

実行過程に束縛されないことが重要である．

既に述べたように，プログラム実行時に生じた副作用の検出を支援するためには，プロトタイピングが必須である．即ち，改変された部分プログラムの実行結果を，階層的な仕様上に明示しなくてはならない．

2.3 仕様記述と実行可能プログラムのデータ依存性に関する対応関係の維持

プログラムの実行結果を仕様記述上に提示するには，仕様記述と実行可能プログラムの間での，データ依存性の対応関係の維持が前提となる．ノイマン型プロセッサとその上の高級言語の利用を前提とした，構造化プログラミングとその流れを汲む SADT[31]，OMT[83] 等の従来手法では，実行可能プログラムを得る過程でコンパイルが不可欠であった．宣言的に書かれた仕様記述と手続き型に実行されるプログラムとの乖離が避けらないからである．実行可能プログラムから仕様記述への逆変換は一般に定義できないので，それらの従来手法では，仕様記述と実行可能プログラムの間のデータ依存性に関する対応関係は主として人手で維持する他なかった．従って，この対応関係の維持は困難であったばかりか，プログラム実行時に生じた副作用を仕様記述水準で明示するのは一般に不可能であった．

本研究は，並列処理の表記法の一つである DDS で書かれた並列処理プログラムを最も効率的に実行するデータ駆動プロセッサアーキテクチャを検討する設計思想に基づいている．即ち，仕様記述と実行可能プログラムの双方には，本質的な情報としてデータ依存性が含まれていること，および，DDS がデータ駆動プロセッサの実行可能プログラムの表現法のみならず，SADT や OMT において仕様記述手段としても利用されてきた事実に着目した．

DDS の構成要素を 図 2.1に示す．DDS は，仕様記述あるいは部分仕様を記述するためのブロック（図 2.1(a)），機能要素を示すノード（図 2.1(b1)），ノードの入出力を示すポート，外部との入出力を示すソース（図 2.1(b2)）・シンク（図 2.1(b3)），データ依存性を示すアーク（図 2.1(c)），およびそれらの名前からなる．

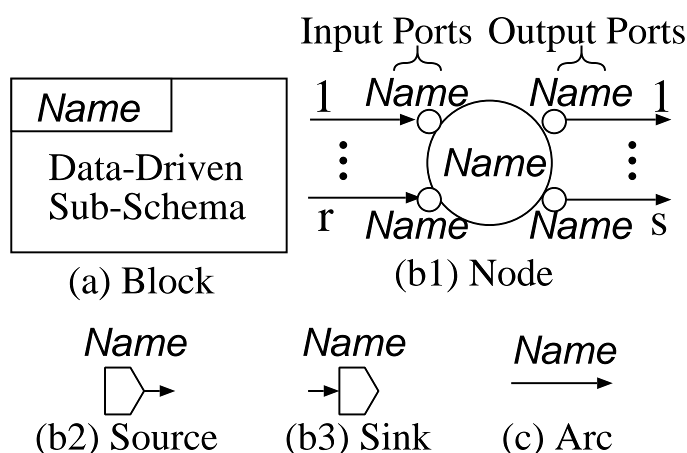


図 2.1: データ駆動図式の構成要素

図 2.2に示すように，本研究では，仕様記述と実行可能プログラムの双方の表現法に DDS を採用し，仕様記述に対して実行可能プログラムの生成に必要な情報を付与することによって，実行可能プログラムを得る方式を採用した．即ち，仕様として記述された階層的な DDS (a) の階層構造を展開したのと同一のデータ依存性を持つ実行可能プログラム (b) を生成する．この方式により，データ依存性については，仕様記述 (a) と実行可能プログラム (b) の間でつねに対応関係を維持できる．従って，プログラム実行時に生じた副作用を仕様記述水準に明示できる．さらに，(a) とは別の階層構造を持つ仕様 (a') と (b) の間でも同様に対応関係を維持できる．このことから，仕様記述の階層構造を (a) から (a') へ，あるいはその逆のように再構成することも可能となる．

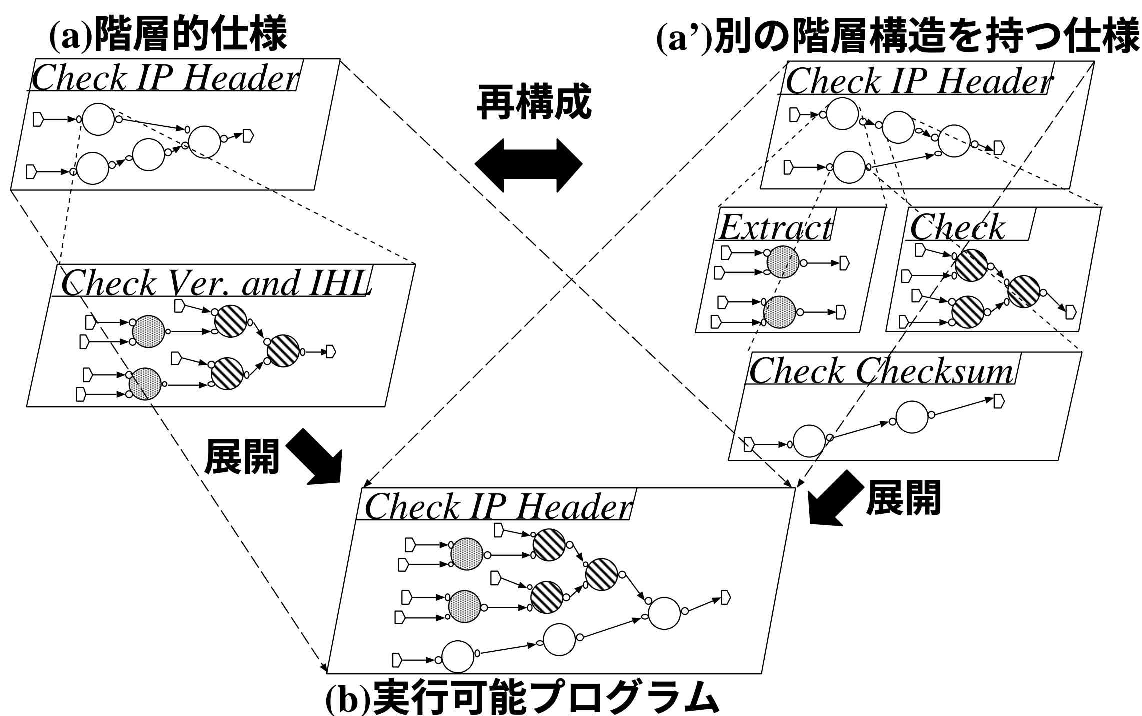


図 2.2: 多面的な仕様記述と実行可能プログラム間のデータ依存性に関する対応関係の維持

DDS は，ペトリネットグラフのサブクラスであり，問題に内在する並列処理性をデータ依存性によって表現する．この DDS に対し，トークン流に関するデータ駆動原理を適用することにより， DDS で表現されたプログラムの実行時における，同時併行・多重・パイプラインのすべての並列処理性が自然に表現される．この特徴に注目し，Jack B. Dennis は，DDS に基づく並列プログラム表記法とともに，副作用なしとなるデータ駆動実行 (Well-Behaved Execution) とこれを保証する良構造 (Well-Formed Structure) を提案した [44]．DDS の，副作用のない実行については，次のことが既に分かっている [57].

定義 1. 内部にトークンを含まない状態 M_0 にある DDS に, 入力トークン $t_{i0}, \cdots t_{im}$ を与えた時, $t_{i0}, \cdots t_{im}$ をすべて消費し, 有限の実行系列により, $t_{i0}, \cdots t_{im}$ に対して一意的な出力トークン $t_{o0}, \cdots t_{on}$ を生成した後, 再び M_0 に戻る時, この DDS は, $t_{i0}, \cdots t_{im}$ に関して副作用なしである.

定義 2. 関数的オペレータを非巡回的に結合して記述された DDS は良構造である.

性質 1. 良構造 DDS は，それを構成する関数的オペレータ群の定義域内の入力に関して副作用なしである.

定義 1 より，副作用のない DDS は，それ自身入力トークン $t_{i0}, \cdots t_{im}$ から出力トークン $t_{o0}, \cdots t_{om}$ を生成する関数的オペレータとみなすことができる．従って，明らかに次の系が成立する.

系 1. 良構造 DDS 中の関数的オペレータを副作用のない DDS で置換して生成される DDS は良構造である.

D^3L[57] を含め従来のデータ駆動言語の多くは，系 1 に基づき， 階層的記述を許す良構造な DDS と関数的オペレータを核として，副作用なしを保証していたが，良構造はあくまでも副作用回避のための十分条件に過ぎず，制約が強すぎるきらいがある．例えば，良構造に厳密に従えば，関数的オペレータとするため，分岐した

出力はすべてマージしなければならないが，実際にはマージせずともトークンの競合やいわゆるファントムトークンによる非決定性が回避されれば十分である．

加えて，プログラムの正しさはそれ自体で議論できるものではなく，あくまでもある定められた仕様にてらして判断されるものである．事実，従来の検証方法では，プログラムの仕様を入力表明と出力表明の組として定義し，入力表明を満たす正当な入力に対してプログラムを実行した時，プログラムが確かに停止し且つ出力表明を満たす正当な出力が得られる場合にそのプログラムは全正当であると定義している．この全正当を証明する手法として，部分正当性および停止性を示す方法など，さまざまな研究がある [64]．

従って本研究では，定義 1 を基にして，仕様記述にてらして正当性を検証する立場をとり入れ，次のように拡張した定義 3 を副作用のないデータ駆動実行として採用する．

定義 3 S を，m 個の入力アークと n 個の出力アークを持つ DDS とする．M_0 を，仕様記述を満足する初期トークンと履歴を内部に持った S の状態であるとする．S に入出力アークを付加したものを S' とする．以下の 1)2) のいずれかの条件を満足するとき，データ駆動図式 S の実行系列は M_0 と入力トークンに関して副作用なしである．即ち，1)S' が，状態 M_0 から開始して有限の実行系列の後に，仕様記述を満足する状態 M_1 になる．2) 無限の実行系列がそれ自身で仕様記述を満足する．

2.4 記号実行に基づくプロトタイピング手法

副作用の有無の検証を目的としてプログラムを実行する際に，すべての入力データを網羅するのは一般に不可能である [29][23]．本研究では，プログラム検証に記号実行 [65] を採用した．記号実行は，具体的な入力データ値の代わりにデータを代表する記号を与えてプログラムを実行することで，プログラムの性質を形式的に検証するものである．例えば Alur らは，温度センサーの入力に基づいて核反応炉を制御する問題を数学モデルで記述して，その振舞いを記号実行を用いて検証する手法

を提案している [84].

また，抽象解釈 (Abstract Interpretation)[85] という技法を用いた研究もある [86]. 抽象解釈では，プログラムの詳細化を入出力データ型の詳細化と捉える．即ち，詳細化過程の各段階における（抽象的な）プログラムの機能を，その段階での入出力データ型の関係を用いて定義する．しかし本研究が実行システムとして対象としている CUE プロセッサは，信号処理を目的に開発されたものであるので，データ型という概念を持たない．従って本研究ではデータ型という概念を明示的に取り扱わないので，抽象解釈は採用せず，記号実行を採用した．

記号実行の結果としては，出力データを入力データの記号で表現した式が得られる．異なる記号実行結果がユーザにとっては同一の値を意味する場合もあるので，記号実行結果から副作用の有無を判定できるのはユーザのみである．既に述べたように，複数の副作用の相互作用により副作用が一時的に隠蔽される危険性を考慮すると，ユーザが仕様を改変した時はつねに副作用検出支援を試みる必要があるので，対話的なプロトタイピングは必須となる．

従って，副作用のない実行可能プログラムを得るためには，記号実行に基づくプロトタイピングを介して実行可能プログラムを対話的に直接生成していく他ない．即ち，ユーザが宣言的に仕様として記述した階層的な DDS に対し，仕様記述環境は階層構造を展開した上でデータ駆動型解釈を試みる．このとき，仕様記述と実行可能プログラムの間のデータ依存性に関する対応関係は維持される．仕様記述環境は，実行可能プログラムの生成に必要な命令プリミティブ割り当て・ポート間の対応づけの情報をユーザに要求し，獲得する．プログラムが部分的に実行可能となった場合はこれを記号実行し，その結果を対応する仕様記述上に提示する．

2.5 仕様記述の階層構造の再構成手法

記号実行の結果，ある出力データの記号式がそれまでのものから変化した場合，これが原因となって後続のプログラムの実行時に副作用が発生する可能性がある．後続のプログラムも記号実行することで，副作用を受ける部分が特定される．この部分の DDS の規模は，つねにユーザの了解可能な規模に収まるとは限らない．従って，必要に応じてデータ依存性を畳み込み，仕様記述の階層構造を再構成する必要がある．

あらゆる DDS について，任意の 2 データ間の依存関係は，図 2.3に含まれるアーク間の依存関係のいずれかに対応する [67]．これは以下のように示される．即ち，2 つのデータに対応するアークに挟まれた副図式 Main がまず決定される（2 つのデータ間にデータ依存性が無い場合は空である．以下同様.）．DDS においてデータ依存性を有する 2 ノードは必ず一方が上流，他方が下流となるので，Main 内のノードとデータ依存性を持つノードは Main に対して上流か下流のいずれかに分類される．これにより，Main へ向かうデータ依存性を持つ副図式 Pre，Main からのデータ依存性を持つ副図式 Suc が定まる．同様に，Pre と Suc との間のデータ依存性は，Pre から Suc へ向かうデータ依存性と，逆に Suc から Pre へ戻るデータ依存性に分類される．そのようなデータ依存性に挟まれた副図式を，前者を Con，後者を Loop とする．ここで明らかに，Main, Pre, Suc, Con, Loop 以外の副図式としては Main, Pre, Suc, Con, Loop のいずれともデータ依存性を持たない副図式 Indep しか存在しない．Main, Pre, Suc, Con, Loop, Indep は外部との入出力を持ち得ることから，最終的に得られる DDS は図 2.3 のようである．

この DDS 上に記号実行結果の情報を提示すれば，すべての DDS において，ある入出力データ間の依存関係をこの程度の複雑さの DDS を用いて明示できる．

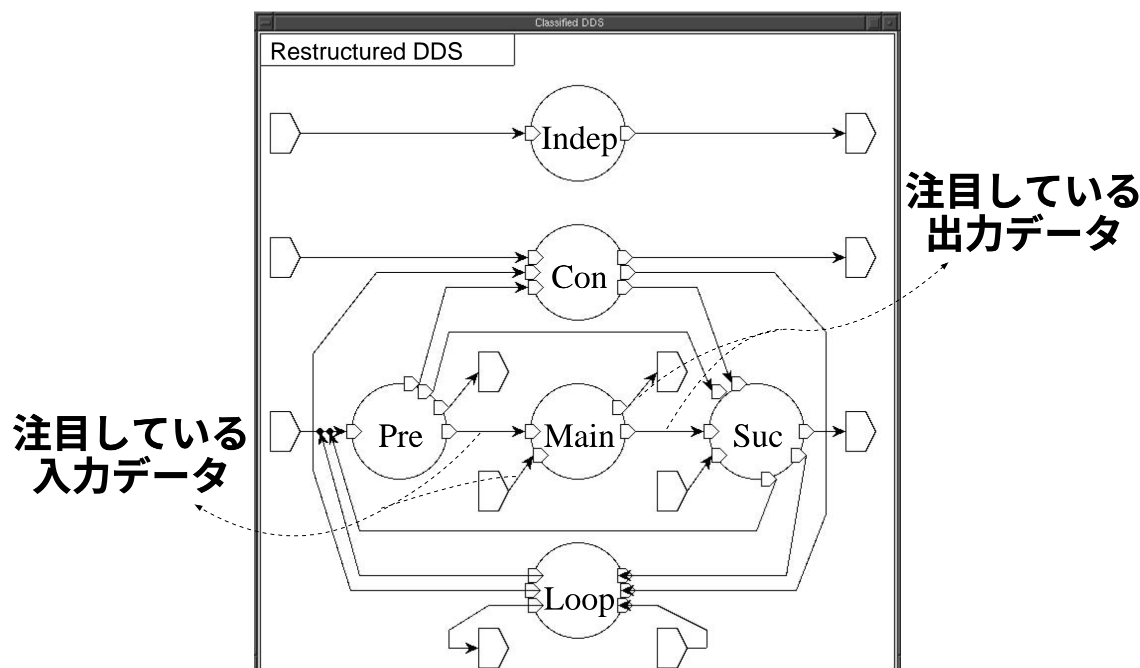


図 2.3: あらゆる種類のデータ依存性を含む DDS

2.6 結言

本章では，記号実行に基づくプロトタイピングを介してユーザとシステムが対話的に実行可能なデータ駆動プログラムを直接生成することによって，副作用を検出支援する手法を提案した．まず，本研究では仕様記述の階層構造を必要に応じて再構成することを前提にすることを述べた．次に，副作用検出を支援するためには，仕様記述とプログラムのデータ依存性に関する対応関係を維持することが重要であることを述べた．最後に，副作用を仕様記述上に提示する際にユーザの了解性を維持するために必須である，データ依存性の畳み込み手法を示した．

次章では，データ駆動型仕様記述環境におけるこれらの手法の実現法を示す．さらに，これらの手法を TCP/IP プロトコル処理プログラムに適用して評価する．

第 3 章

データ駆動型仕様記述環境の実現法とその評価

3.1 緒言

本章では，記号実行に基づくプロトタイピング手法を実装したデータ駆動型仕様記述環境 [66]-[68] の実現法とその評価を述べる．まず，記号実行の実現法として，LISP 言語でいう万能関数 eval[87] を用いて記号実行を実現する手法を述べる．次に，対話的なプロトタイピングを実現するためには，記号実行の計算コストの低減が重要となるので，過去の記号実行結果を活用することによって，記号実行のコストを，DDS の一階層の規模程度に抑制する手法を述べる．さらに，本仕様記述環境を，TCP/IP (Transmission Control Protocol / Internet Protocol)[53][42] プロトコル処理の中の IP 層受信処理部の開発に適用して，評価する．その結果，動的データ駆動原理においてトークンに付与されたタグ [58] である世代 [57] の操作に起因する副作用が検出されたことを示す．さらに，従来検出が困難とされてきた仕様の抜け・誤りが検出されたことから，本手法が Verification[69] にも有効なことを示す．

3.2 記号実行に基づくプロトタイピングの実現法

仕様記述過程においてユーザに課せられる制約は極力排除されるべきであるから，本研究での仕様記述過程は実行方式としてのデータ駆動原理に非依存とした．即ち，ノードやアークをどの順序で記述するかはユーザの自由である．また，仕様記述の階層構造の記述時と同様，命令プリミティブのポートと仕様記述上のノードのポートとの間の対応づけを，仕様記述環境が自動的に行うようにすると，実行機械の命令プリミティブの実装に依存して仕様記述上のノードの入出力ポートを記述する必要が生じるため仕様記述の自由度を制約する．従って本研究ではこの対応関係もまたユーザが付与するものとした．

記号実行を実現するには，命令プリミティブ毎の記号実行規則とそのインタプリタが必要である．本実現法では，汎用記号処理系 Perl をインタプリタに採用した．即ち，記号実行規則は Perl プログラムとして表現され，この記号実行規則を変数に文字列として格納した上で，この変数の内容をプログラムとして実行する機構（LISP 言語でいう eval 機構 [87]）を用いることで，記号実行を実現した [67].

例えば，図 3.1(a) に示すように，左側の入力トークンの世代から右側の入力トークンのデータを減算する命令プリミティブ (gsub) の記号実行規則は，Perl 風の文法を用いて

$$\text{“}\{\underline{\mathrm{O}_{t0}},\ \underline{\mathrm{O}_{d0}}\} \leftarrow \{\underline{(\mathrm{I}_{t0}\text{-}\mathrm{I}_{d1})},\ \underline{\mathrm{I}_{d0}}\}\text{”}$$

のように表現される．ここで，下線つきの部分は変数を意味し，ダブルクォートで囲まれた部分（“···”）は文字列を意味する．←は代入演算子の形をした文字である．O は出力記号，I は入力記号を意味し，$_{t0}$ はタグ，$_{d0,d1}$ はデータを意味する．これらの記号実行規則に対して，入力アーク $(a0, a1)$ に到着したトークンの持つデータを表す記号式として

$$\{\underline{\mathrm{I}_{t0}},\ \underline{\mathrm{I}_{t0}},\ \underline{\mathrm{I}_{d0}},\ \underline{\mathrm{I}_{d1}}\} \leftarrow \{\text{“TAG0”, “TAG1”,“DATA0”,“DATA1”}\}$$

なる条件を与えた上で eval 機構で評価すると，

$$\{\underline{\mathrm{O}_{t0}},\underline{\mathrm{O}_{d0}}\} \leftarrow \{\text{“(TAG0-DATA1)”,“DATA0”}\}$$

なる記号実行結果が得られる．この記号実行結果を得ることを，以下本稿では,「アーク $a2$ の記号実行をする」と呼ぶ．副作用を検出支援するためには，仕様の改変に伴って実行可能プログラムが変化した時にはつねに記号実行を実施する必要がある．さらに，記号実行結果の式は，プログラムの規模に依存した規模の式となるため，これを仕様記述上に提示する際には，ユーザが了解可能な規模で表現する必要がある．本研究では，ある記号実行結果に対しアーク名を別名として付与し，そのアークよりも下流の記号実行結果はそのアーク名を用いて表現することで，記号実行結果の複雑さを抑えられるようにした．

記号実行の計算コストが実行可能プログラム全体の規模に依存すると，対話的な仕様記述環境の実現が原理的に困難となる．本研究では，過去の記号実行結果を利用して計算コストを低減する．即ち，ブロック毎に，そのブロックを命令プリミティブと見なした記号実行規則をシンク毎に生成した後，ブロック単位でまとめる．さらに，このブロックがノードに割り当てられていればそのノード毎に記号実行規則を保存する．具体的には，図 3.1(b) に示すように，$\underline{\mathrm{I}_{t0}}$ などの形をした記号を，ブロック内のソースに入力として与えて，そのブロックに対応する部分プログラムを記号実行する．例えば，前出のタグ操作命令 2 つが直列に繋がったプログラムの場合，

$$\{\underline{\mathrm{I}_{t0}}, \underline{\mathrm{I}_{t1}}, \underline{\mathrm{I}_{t2}}, \underline{\mathrm{I}_{d0}}, \underline{\mathrm{I}_{d1}}, \underline{\mathrm{I}_{d2}}\} \leftarrow \{\text{``}\underline{\mathrm{I}_{t0}}\text{''}, \text{``}\underline{\mathrm{I}_{t1}}\text{''}, \text{``}\underline{\mathrm{I}_{t2}}\text{''}, \text{``}\underline{\mathrm{I}_{d0}}\text{''}, \text{``}\underline{\mathrm{I}_{d1}}\text{''}, \text{``}\underline{\mathrm{I}_{d2}}\text{''}\}$$

なる条件を与えて eval 機構で評価すると，

$$\{\underline{\mathrm{O}_{t0}},\underline{\mathrm{O}_{d0}}\} \leftarrow \{\text{``}((\underline{\mathrm{I}_{t0}}\text{-}\underline{\mathrm{I}_{d1}})\text{-}\underline{\mathrm{I}_{d2}})\text{''},\text{``}\underline{\mathrm{I}_{d0}}\text{''}\}$$

のような結果が得られ，これは，図 3.1(c) の"gsub*2" のように，このブロックを 1 命令プリミティブと見なした時の記号実行規則に他ならない．これにより，仕様記述上の任意のノードは，それを実現する部分プログラムの規模に関わらず，eval 機構による評価 1 回のみでプロトタイピングできる．

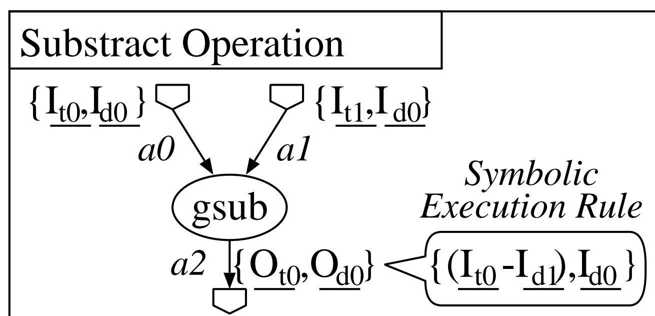
以上より，記号実行の実施条件を以下のように定義した．即ち，

1° プログラムへの入力を示すソース（以下，単にソースと記述した場合はこの意味とする）が記述された場合，そのブロック内で一意の識別子を付与する．

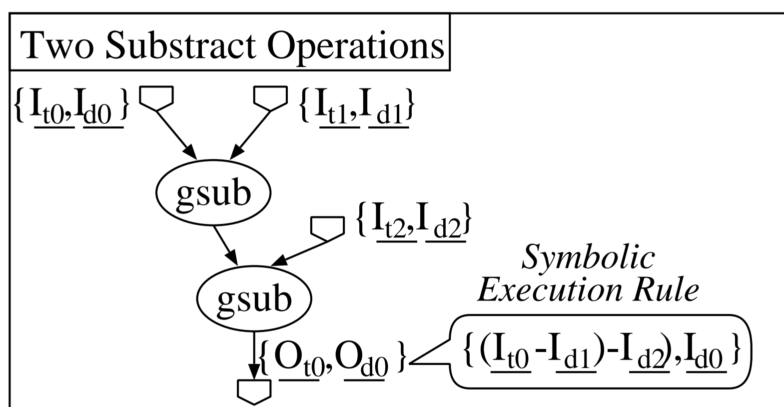
2° 入力ポートが記述された場合，この入力ポートが定義されているノード上で一意の識別子をこの入力ポートに対して付与する．

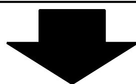
3° ノードの仕様が改変された場合，出力ポートに対し，そのノードの入力を用いて記号実行規則を生成する．ここで，ノードの仕様の改変とは，入出力ポートの定義・削除，割り当てる命令プリミティブ・ブロックの変更あるいは記号実行規則の更新，入力アークの記号実行規則の更新を指す．また，ノードの入力とは，入力ポートにアークがあればそのアークが持つ記号実行規則またはユーザが付与したアーク名，アークがなければ入力ポートが持つ識別子を指す．出力ポートのうち，対応する記号実行結果が変化し且つアークが接続されているものがあるならば，その各々に対して，記号実行規則を更新し仕様記述上に記号実行結果を提示して **4**° へ．記号実行結果が変化しないか，または，出力ポートにアークが接続されていなければ何もしない．

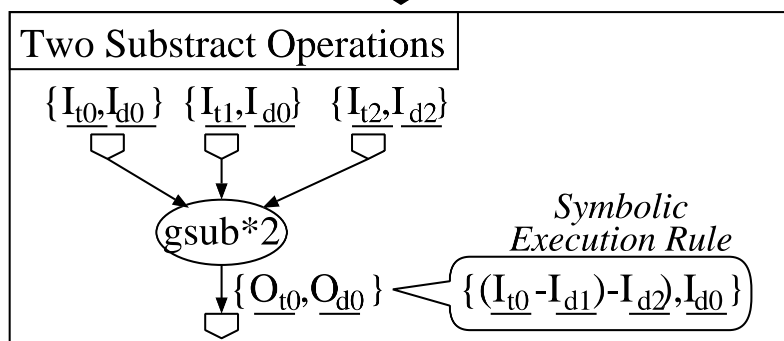
4° アーク上の仕様記述に対して改変が加えられた場合，アークの記号実行を行う．ここで，アーク上の仕様記述に対する改変とは，アークの定義，アーク名の変更，始点ポートの記号実行規則の更新を指す．記号実行結果が変化し且つ終点が入力ポートならば，アークの記号実行規則を更新し，接続先のノードに対して **3**° へ．記号実行結果が変化し且つ終点がシンクならば，シンクの記号実行規則を更新し，このブロックの記号実行規則についても更新する．その後，このブロックにノードが割り当てられていればそのノードの記号実行規則も更新した後，そのノードに対して **3**° へ．記号実行結果が変化しない場合は何もしない．



(a) Symbolic Execution Rule and Primitive







(b) Regarding Sub-Program As A Primitive
By Defining A New Symbolic Execution Rule

図 3.1: プリミティブ・部分プログラムとその記号実行規則

これらのような記号実行を実現するために前提となる，仕様とプログラムとのデータ依存性に関する対応関係を，次のような方法によって維持した．

まず，仕様が階層構造をとらない場合における，仕様とプログラムのデータ依存性に関する一貫性維持の実現法を図 3.2に示す．図 3.2(a) は，出力データを生成しない場合を除くと最も単純な DDS である．図 3.2(b) は，入力データもしくは出力データを複数持つ場合を示した図である．図 3.2(c) は，入力データがマージされている場合もしくは出力データをコピーする場合を示した図である．仕様が階層構造をとっておらず ブロック 1 個のみで記述されている場合には，プログラムとしての DDS は仕様として書かれた DDS とまったく同じデータ依存性となる．従って，(a)〜(c) のいずれの場合においても，図中 B_s と書かれた破線に示すように仕様上のソースとプログラム上のソースは 1 対 1 に対応する．これはシンク，アーク，入力ポート，出力ポートについても同様である．仕様上のシンクとプログラム上のシンクの対応関係は図中 B_k と書かれた破線で示した．仕様上の入力ポートとプログラム上の入力ポートの対応関係，および，仕様上の出力ポートとプログラム上の出力ポートの対応関係は図中 B_p と書かれた破線で示した．仕様上のアークとプログラム上のアークの対応関係は図中 B_a と書かれた破線で示した．

仕様が階層構造をとる場合における，仕様とプログラムのデータ依存性に関する一貫性維持の実現法を図 3.3の (d) ならびに図 3.4の (e) に示す．図 3.3の (d) における仕様としての DDS には，1 ノードを詳細化した結果 1 ノードとなるような特殊な場合を除いて最も単純な階層的 DDS のひとつを示した．仕様の最下位層については，階層構造をとらない場合と同様に，仕様上のソースとプログラム上のソースは 1 対 1 に対応する．シンク，アーク，入力ポート，出力ポートについても同様である．仕様の最上位層については，ソースはプログラム上のソースに対応する．シンクについても同様である．従って，仕様上でソースを始点とするアークは，プログラム上でもソースを始点とするアークと対応する．そのようなアークの終点としての入力ポートについても同様であり，仕様上の入力ポートはプログラム上の入力ポートと対応する．シンクを終点とするアークについても同様である．仕様上でシンクを終点とするアークは，プログラム上でもシンクを終点とするアークと対応す

る．そのようなアークの始点としての出力ポートについても同様であり，仕様上の出力ポートはプログラム上の出力ポートと対応する．

次に，図 3.4の (e) における仕様としての DDS には，仕様の最上位層が 2 ノード以上を含む場合に最も単純な階層的 DDS のひとつを示した．この図 3.4の (e) においては，図の了解性を維持するため，図 3.3の (d) とまったく同様の対応関係を示す破線は省略した．図 3.4の B_a に示すように，最上位層でノード間を接続するアーク a は，プログラム上でもアーク α に対応する．従って，図 3.4の B_p に示すように，a の始点となる出力ポートには α の始点となる出力ポートが対応する．同様に，a の終点となる入力ポートには α の終点となる入力ポートが対応する．図 3.4の「最下位層の仕様 (1)」でシンクと接続されているアーク a_k もまた，プログラム上のアーク α に対応する．従って，このアークの終点となるシンク k は，図 3.4の B_k に示すように，α の終点となる入力ポートと対応する．同様に，図 3.4の「最下位層の仕様 (2)」でソースと接続されているアーク a_s もまた，プログラム上のアーク α に対応する．従って，このアークの始点となるソース s は，図 3.4の B_s に示すように，α の終点となる入力ポートと対応する．

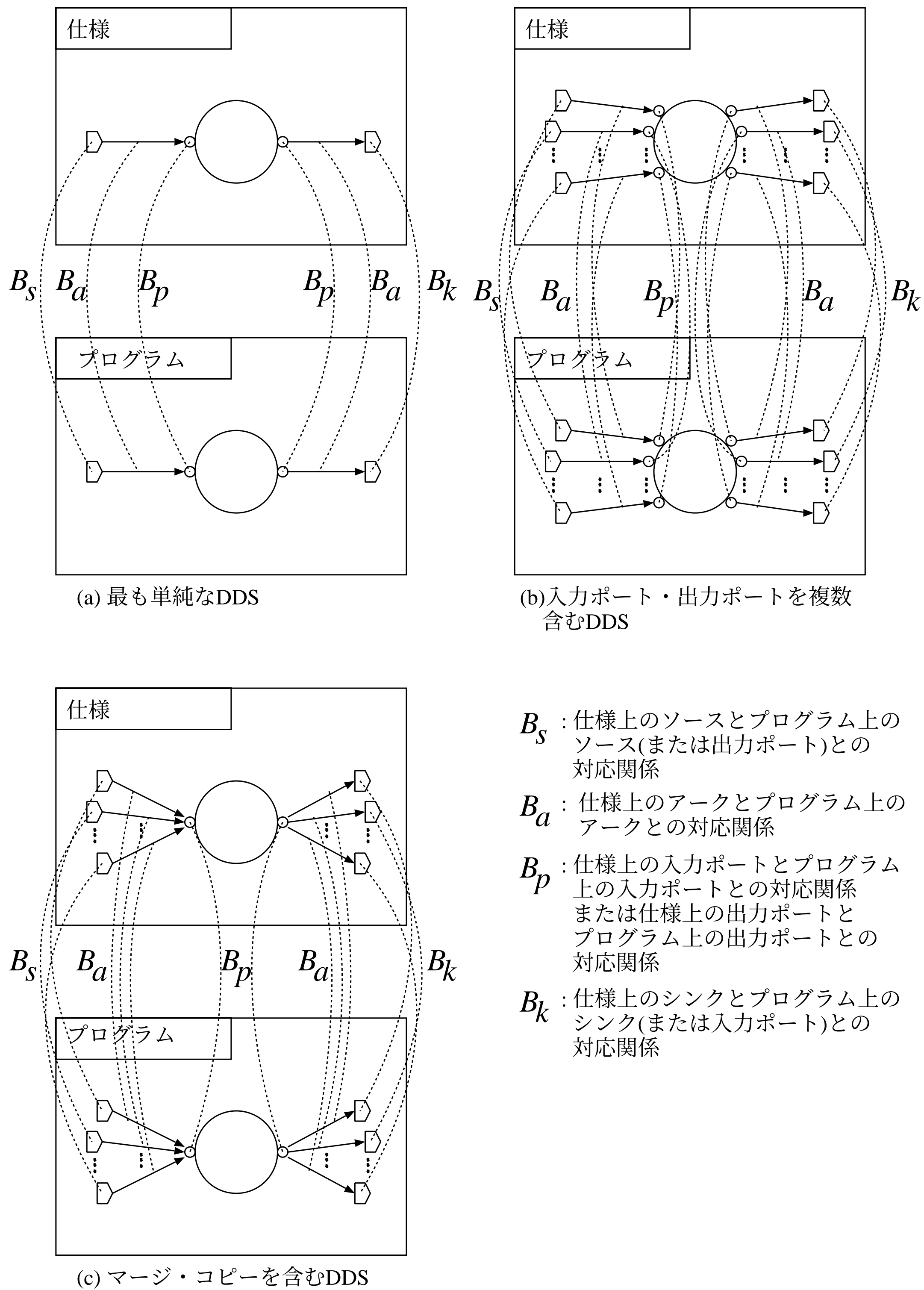
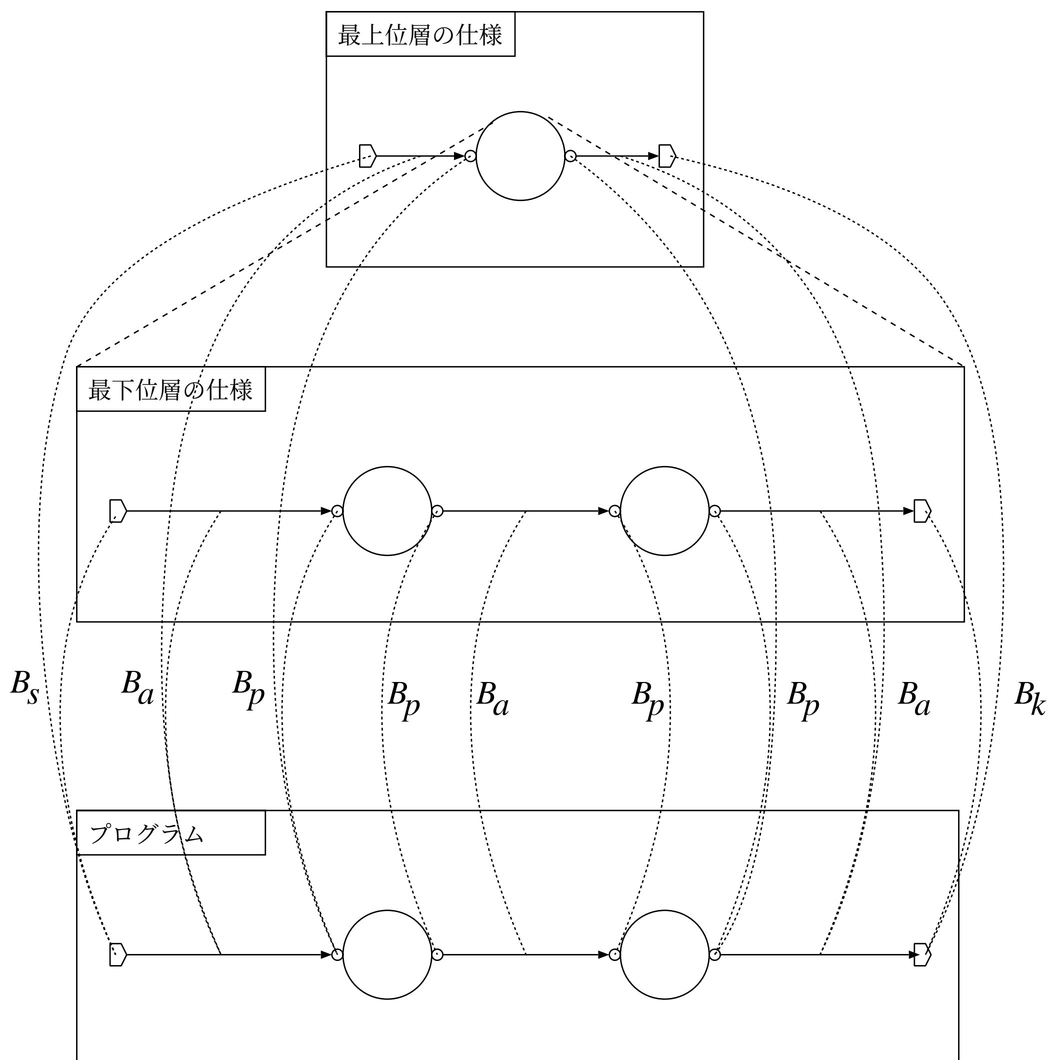


図 3.2: 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (1)



B_s : 仕様上のソースとプログラム上のソース(または出力ポート)との対応関係

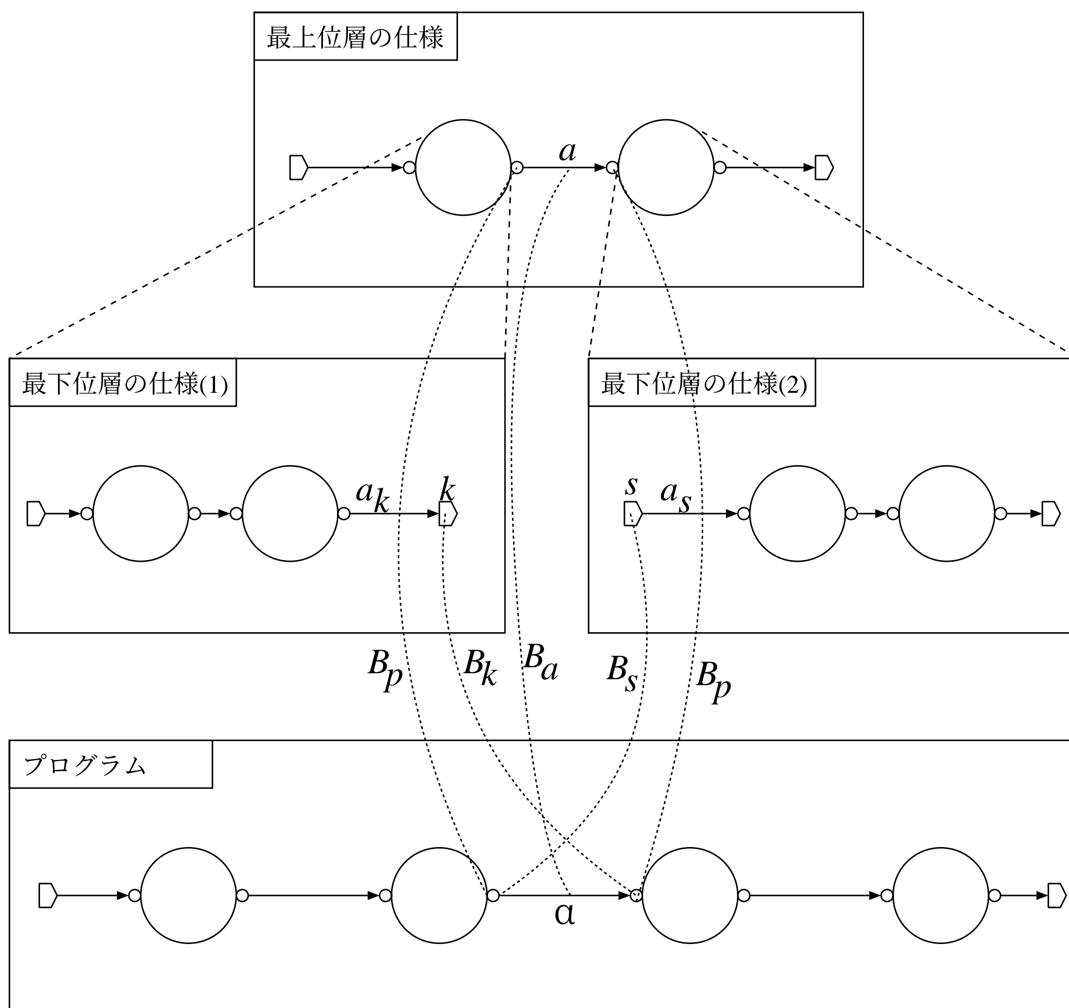
B_a : 仕様上のアークとプログラム上のアークとの対応関係

B_p : 仕様上の入力ポートとプログラム上の入力ポートとの対応関係
または仕様上の出力ポートとプログラム上の出力ポートとの対応関係

B_k : 仕様上のシンクとプログラム上のシンク(または入力ポート)との対応関係

(d) 最も単純な階層的DDS

図 3.3: 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (2)



B_s : 仕様上のソースとプログラム上のソース(または出力ポート)との対応関係

B_a : 仕様上のアークとプログラム上のアークとの対応関係

B_p : 仕様上の入力ポートとプログラム上の入力ポートとの対応関係
または仕様上の出力ポートとプログラム上の出力ポートとの対応関係

B_k : 仕様上のシンクとプログラム上のシンク(または入力ポート)との対応関係

(e) 最上位層が2ノード以上含む場合の最も単純な階層的DDS

図 3.4: 仕様とプログラムのデータ依存性に関する一貫性維持の実現法 (3)

3.3 データ駆動型仕様記述環境

本プロトタイピング手法を組み込んだデータ駆動型仕様記述環境を実現した[67][68]．その内部構成を図 3.5に示す．まず，ユーザは機能仕様を記述する．仕様記述環境は，機能仕様を内部表現に変換するとともに，実行可能プログラムおよび入力ストリームを生成する．仕様記述環境は同時に，それらを用いた記号実行結果を仕様記述上に明示する．記号実行結果を見ることで，ユーザはプログラムを検証する．

本仕様記述環境の出力として最終的に生成されるプログラムは，現在の処，取り扱いの容易さからアセンブラソースコードの形式をとっているが，これは実行可能プログラムを生成しているのと実質的に等価である．

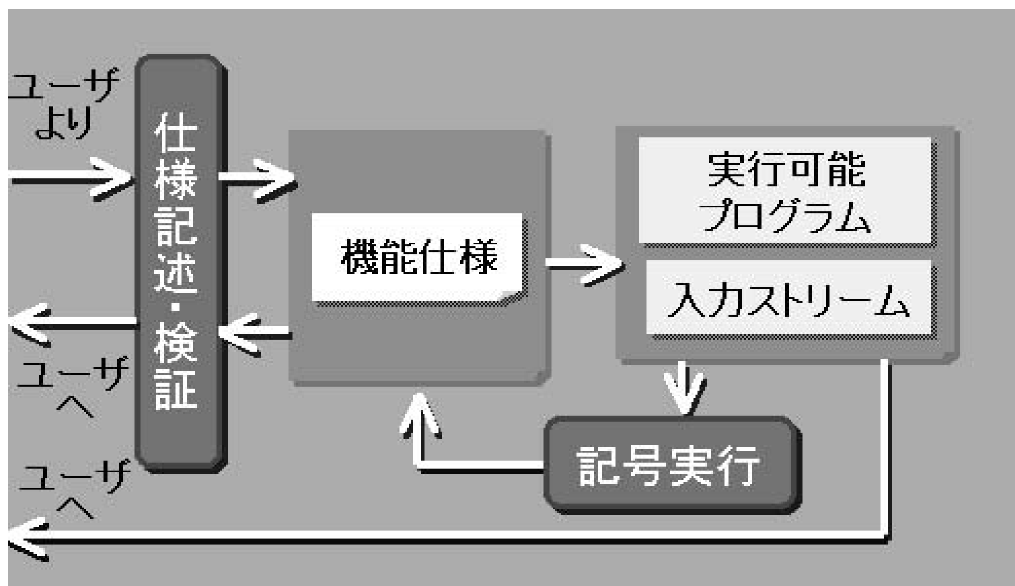


図 3.5: データ駆動型仕様記述環境の内部構成

表 3.1に，ユーザの仕様記述操作と仕様記述環境からの応答との対応関係を示す．表 3.1の (a)〜(h) は，DDS の構成要素に直接対応する．仕様記述時の制約を極小化するため，存在しないブロック内にはノードを記述できないなど，無意味な仕様記述を回避するために必要なものを除き，ユーザの操作の順序に制約を設けていない．

(i)〜(l) は階層構造の記述操作で，(i) はノードの機能が命令プリミティブで実現可能な場合に，(j) はボトムアップな記述順序に，(k) はトップダウンな記述順序にそれぞれ必要である．(l) は階層構造を表現するのに必要である．1 ノードからなる仕様記述とそのプログラム生成には (a)〜(i) で十分であって，また，階層的な仕様を記述するためには (j)〜(l) で十分である．以上より，ユーザの操作 (a)〜(l) は，仕様の記述および仕様からの実行可能プログラムの直接生成に必要かつ十分である．

また，表 3.1に示した，仕様記述環境からの応答の十分性に関しては以下のようである．即ち，プログラム生成，記号実行の実施，および記号実行結果の仕様記述上への提示の処理すべてを，ユーザからのあらゆる操作 (a)〜(l) に対し行うようにすれば，十分であることは明らかである．しかしながら，ノード名・ソース名・シンク名は生成されるプログラムに影響しないので，これらの操作（表 3.1(d)(e)(h)）の際にはプログラム生成は不要である．空のブロックを定義する操作（表 3.1(a)(k)）についても同様である．一方，これら以外の操作，例えば，ノードの記述やソースの記述など（表 3.1(b)(c)(f)(g)(i)(j)(l)）は，生成されるプログラムに影響を与えるので，階層構造の展開とプログラム生成が必要である．これらの操作のうち，ノード・ソース・シンク・ポートの記述操作は，それら単独ではプログラムが実行可能となり得ない．従って，プログラムが生成される操作のうち表 3.1(b)(c)(f) を除いた，表 3.1(g)(i)(j)(l) の場合に，実行可能性の検査と記号実行の実施，および，記号実行結果の仕様記述上への提示が必要である．以上より，表 3.1の仕様記述環境からの応答は必要かつ十分である．

仕様記述環境がプログラムの実行可能性をチェックした際には，実行可能となるために不足している情報が明らかとなる．既に述べたように，この不足している情報を積極的に獲得するため，仕様記述環境は，不足している情報に対応する「?」を表示するようにした．

表 3.1: 仕様記述環境におけるユーザの操作と環境からの応答との対応関係

	ユーザの操作	仕様記述環境からの応答
(a)	空ブロックを新たに生成する.	ブロック名の記述を促す.
(b)	ソース・シンクを記述する.	プログラム上にソース・シンクを生成する．ソース・シンク名，アークの記述を促す.
(c)	ノードを記述する.	プログラム上にソース・シンクを生成する．ノード名，入出力ポートの定義を促す.
(d)	ブロック名を記述する.	何もしない.
(e)	ノード名を記述する.	何もしない.
(f)	ポートを記述する.	プログラム上にポートを生成する．ポート名，アークの記述を促す.
(g)	アークを記述する.	対応するアークをプログラム上に生成する．以下 (h) と同じ.
(h)	ソース名・シンク名・ポート名・アーク名を記述する.	生成されたプログラムの部分的な実行可能性をチェックし，実行可能ならば記号実行を行って，結果を仕様記述上に提示する.
(i)	ノードに命令プリミティブを割り当てる.	対応するプログラム上に命令プリミティブを埋め込む．ノードの入出力ポート・データと命令プリミティブの入出力ポート・データとの対応づけを促す．以下 (h) と同じ.
(j)	ノードに既存のブロックを割り当てる.	ノードの入出力ポートと，ブロックのソース・シンクとの対応づけを促す．以下 (h) と同じ.
(k)	ノードに空のブロックを割り当てる.	ブロック名の記述を促す.
(l)	ノードの入出力ポート・データとブロックのソース・シンク・データとの対応づけを与える.	階層構造を展開したのと同一の接続構造を持つプログラムを生成する．以下 (h) と同じ.

3.4 プロトコル処理プログラムへの適用を通じた評価

本研究では，具体的な応用プログラムとして，TCP/IP プロトコル処理プログラム [55][56] を取りあげ，その中の IP 層受信処理部の一部の開発に本仕様記述環境を適用した．開発過程の典型例を図 3.6〜3.8に示す．これは，IP 層受信処理部の中の，下位 4 ビットのマスク付き比較 (Compare with Masking(Lower)) ブロックである．このブロックは，受け取った IP データグラム内の IP バージョン (IP Ver.) が 4 であるか否かを判定しその結果を出力するためのものであり，一方の数 X (A Number X) に対してマスク値 0xf (Mask Value 0xf) でマスキングし (Masking)，その結果 (Masked X) が他方の数 Y (A Number Y) と等しいか否かを調べて (Compare) その結果 (Whether Masked X Equals to Y or Not) を出力する．

図 3.6は，これらの入出力データのみが記述された段階である．次に，図 3.7に示すように，ユーザは，マスキング (Masking) および比較 (Compare) ノードを記述し，比較 (Compare) ノードに対して誤って加算 (add) 命令プリミティブを割り当てた．図 3.7に破線で示すように，ユーザが，"A Number Y"を d1 に, "Masked X"を d2 に, "Whether Masked X Equals to Y or Not"を d3 に，というように，仕様上のノードの入出力データを命令プリミティブの入出力データに対応づけた．すると，仕様記述環境は "(d1+d2)" という記号実行結果を，仕様記述水準で "(A Number Y+Masked X)" と提示した．これを見てユーザは命令プリミティブ割り当ての誤りに気付いた．そして，図 3.8に示すように，改めて比較命令プリミティブ (eq) を割り当てた．この結果提示された "(A Number Y==Masked X?)" という記号実行結果から，ユーザは，期待通りの実行可能プログラムが得られたことを確認した．

図 3.6: 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (1)

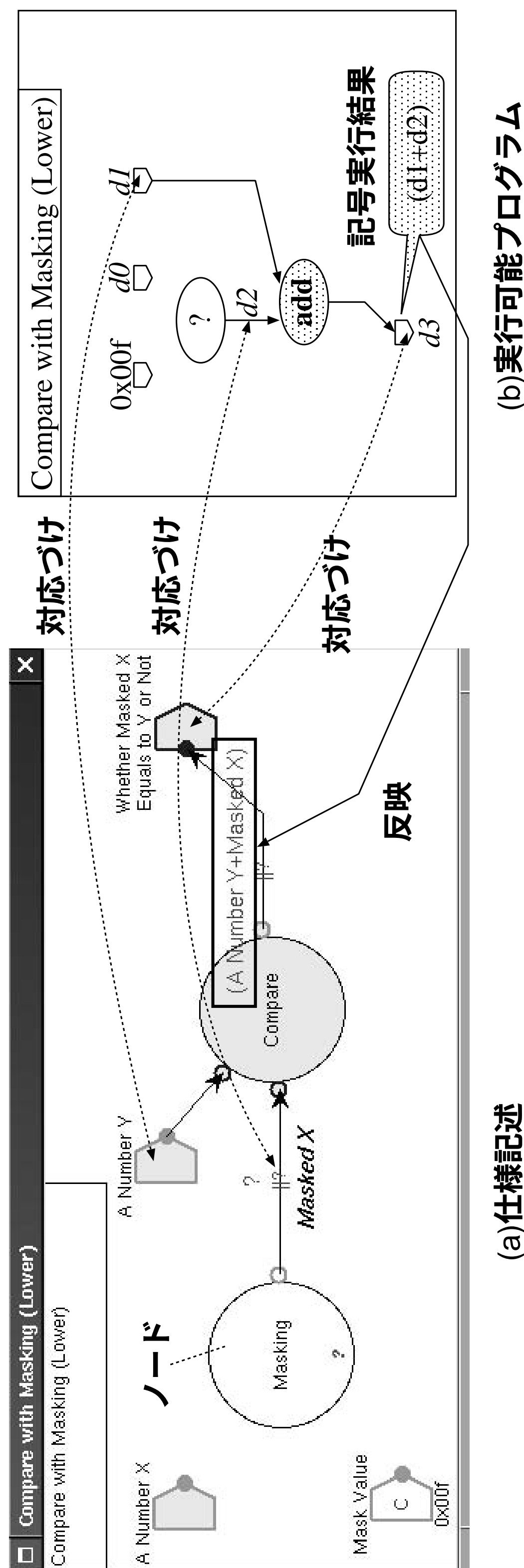


図 3.7: 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (2)

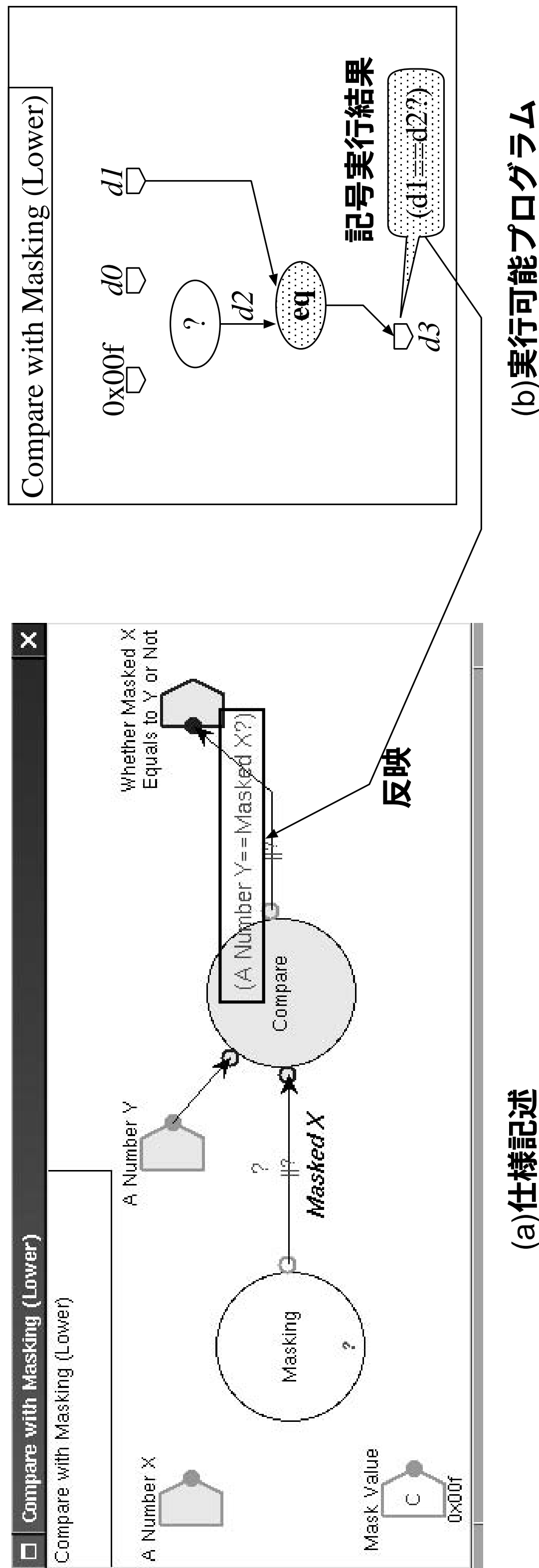


図 3.8: 仕様記述からの実行可能プログラムの直接生成とプロトタイピング (3)

本評価の対象プログラムを構成する命令プリミティブは世代操作命令プリミティブを除けば関数的であり，主に副作用の原因になるのは世代操作命令プリミティブである．副作用以外に発生し得る誤りのうち，従来検出が困難とされてきたものに，仕様記述とプログラム間の対応づけに関する誤りがある．これは，命令プリミティブや階層間の入出力の対応付け等，プログラム生成に必要な情報を，ユーザから獲得する限り避けられない．タグ操作による副作用は，原因となるタグ操作自身はユーザの要求を正確に反映して指定されているのに対して，対応づけに関する誤りは，その対応づけの仕様記述がユーザの要求をそもそも正確に反映していない点が異なる．

従って，本評価で発生し得る副作用・誤りの原因は，以下のいずれかである．即ち，

a) **世代操作の副作用による世代の不一致：** 先行するノードでの世代操作の副作用により，後続のプログラムで入力の世代が一致せず実行不可能となる場合．

b) **仕様記述とプログラムの間の対応づけの誤り：**入出力ポートとソース・シンク間の対応づけの誤り，アーク接続の誤り，命令プリミティブ割り当ての誤り，および，定数インスタンスの誤り．

本手法を用いて IP 層受信処理部を生成し，本手法を評価した結果を表 3.2に示す．表中の数値は，プログラム生成過程において得られた，副作用や誤りを発見した記録を，まとめたものである．今回は，今後の指針を得るために誤り発生の傾向を見ることに主眼を置いたので，誤りの個数でなく，割合を採用した．即ち，まず，誤りの個数を誤りの種類毎に集計した後，それらの個数の全体に占める割当を表 1 に示した.（四捨五入の関係上，合計は必ずしも 100%にならない.）

まず全体の 37% は世代操作の副作用であり，その検出支援に本手法が有効なことが示された．当該プログラム中で世代操作命令プリミティブが使用された場合の概ね 4 回に 1 回は誤りが発生しており，誤りの発生頻度は比較的高い．

次に，アークの記述の誤りが 43%，階層間のポート対応づけの誤りが 8% ，命令プリミティブの選択の誤りが 11%，および，定数インスタンスの指定の誤りが 2% であった．当該プログラムの仕様上で記述されたアークの本数，ソース・シンク数，生成されたプログラム上の命令プリミティブ数，および，定数インスタンス数からは，これらの誤りは発生頻度が比較的低いと言える．

表 3.2: プロトタイピング手法の評価

誤りの原因	割合
世代操作による世代の不一致の副作用	37%
アーク接続の誤り	43%
命令プリミティブ割り当ての誤り	11%
ポートとソース・シンクの対応づけの誤り	8%
定数インスタンスの誤り	2%

世代操作の副作用は，その原因となっている命令プリミティブの近傍でプログラムが実行不可能とならずに，後続のプログラムが実行された結果としてその存在が発見される場合が少なくなく，その場合には，原因の特定が困難である．本手法では，原因となる命令プリミティブが実行可能となった時点で，記号実行によるプロトタイピング結果が提示されるため，副作用は発生した時点で検出される．

一方，副作用以外の，ユーザの指定の誤りはいずれも，生成されたプログラムは実行可能である．その中でも特に，アークの記述の誤りと階層間のポート対応づけの誤りに関しては，正しい仕様記述と誤っている仕様記述の図的な差がわずかであるため，仕様を見るのみでは検出は困難である．

従って，これらの副作用ならびにユーザの指定の誤りの検出が支援されたことは，本手法の有効性を示したと考えている．

第 2 章で述べたように，データ依存性は仕様・プログラムの双方にとって本質的な情報である．実用的な規模の仕様を，ユーザの了解性を維持して取り扱うためには，仕様に階層的構造を採用せざるを得ない．階層間の入出力の対応づけ，命令プリミティブ，定数インスタンスの指定等，ユーザのみが知り得る情報でかつプログラム生成に必要な情報は，ユーザから獲得するのが妥当である．

本評価の結果からは，少なくとも機能仕様に基づいて IP 層受信処理プログラムを生成することを対象とした場合には，タグ操作の副作用さらにはユーザの指定の誤りの検出支援に，本手法が妥当かつ有効なことが示された．この点が本手法の最

大の成果の一つと考えている．

3.5 結言

本章では，記号実行に基づくプロトタイピング手法の実現法とその評価を示した．まず，記号実行の実現法として，LISP 言語でいう万能関数 eval を用いて記号実行を実現する手法を提案した．次に，この記号実行の実現上前提となる，仕様記述としての DDS とプログラムとしての DDS の間のデータ依存性に関する対応関係の維持の実現法を述べた．次に，対話的なプロトタイピングを実現するためには，記号実行の計算コストの低減が重要となるので，過去の記号実行結果を活用することによって，記号実行のコストを，DDS の一階層の規模程度に抑制する手法を述べた．これによって，仕様記述上の任意のノードは，それを実現する部分プログラムの規模に関わらず，eval 機構による評価 1 回のみでプロトタイピングできるようにした．次に，ユーザのオペレーションに対する記号実行の起動条件を示した．この起動条件によって，ユーザが仕様記述を変更した時にはつねに記号実行結果が提示されるようにした．

さらに，本手法を実装した仕様記述環境を，TCP/IP プロトコル処理プログラムの中の IP 層受信処理部の開発に適用して評価した結果を述べた．まず，本評価で発生し得る副作用・誤りの原因が，世代操作の副作用による世代の不一致か，または仕様記述とプログラムの間の対応づけの誤りのいずれかであることを示した．次に，本手法を用いて IP 層受信処理部のプログラムを生成した過程において発見された副作用や誤りを示した．その結果，少なくともユーザが要求仕様を完全に正しく仕様記述に反映するか，または，記号実行結果をユーザが必ず活用する限り，タグ操作による副作用の検出支援のみならず，仕様の抜け・誤りの検出支援にも本手法が有効であることを示した．

第 4 章

データ駆動型実時間システムの性能評価支援手法

4.1　緒言

本章では初めに，データ駆動型実時間システムの開発を支援するためのプロトタイピング手法を述べる．まず，ユーザの要求に関する仕様記述手法として，データ流量・ターンアラウンドタイムの仕様記述手法を提案する．次に，工学的制約の仕様記述として，パイプライン構成の仕様記述手法を提案する．続いて，これらの性能仕様・システム仕様に基づくプロトタイピングのための，性能予測・検証手法とその実現法を述べる．まず，パケットが通過するパイプラインステージと各ステージの平均転送時間とから，実行可能プログラム中のクリティカルパスを求める手法，ならびに，このクリティカルパスの情報を用いてターンアラウンドタイムを予測する手法を述べる．次に，仕様上に記述されたデータ流量の情報と，プログラム構造とを解析して，プログラムの各アーク上を流れるデータ流量を予測する手法を述べる．さらに，生成されたプログラムの実行時にプロセッサのパイプラインが過負荷とならないことを確認するための，パイプライン占有率を予測する手法を述べる．

次に，本プロトタイピング手法を実装した RESCUE (Realtime Execution System for CUE series data-driven processors) を，音声圧縮・動画像圧縮アプリケーション

に適用して評価する．その結果，RESCUE によって，性能評価を対話的に行えること，ボトルネックを検出支援できたこと，ならびに，性能予測結果が，プログラム分散割当の変更による性能改善の指針となったことを報告する．さらに，性能予測に要する時間の実測結果からは，対話的に性能予測可能との結果が得られたことを示す．以上より，本プロトタイピング手法が，実時間システムの開発支援に対して有効であることを示す．

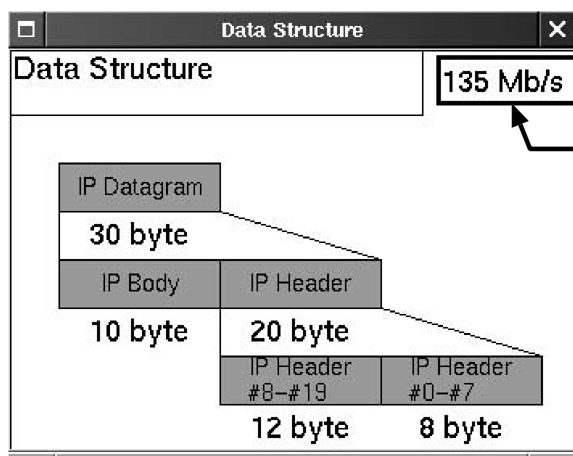
4.2 ユーザの要求・工学的制約の仕様記述

実時間システムは，機能仕様を満足するのみでは不十分であり，性能や時間制約に関する要求を満足する必要がある．このような，性能に関する要求は一般に QoS (Quality of Service) とよばれている [88]．広義には，QoS の構成要素には定量的なものに限らず定性的なものまで含まれていて，応用分野によって様々である [89][90]．情報通信システムにおける性能・時間制約に関する定量的な要求に注目すると，ターンアラウンドタイム，データ流量，さらには，ジッタ，誤り率などが QoS の構成要素である [91]．

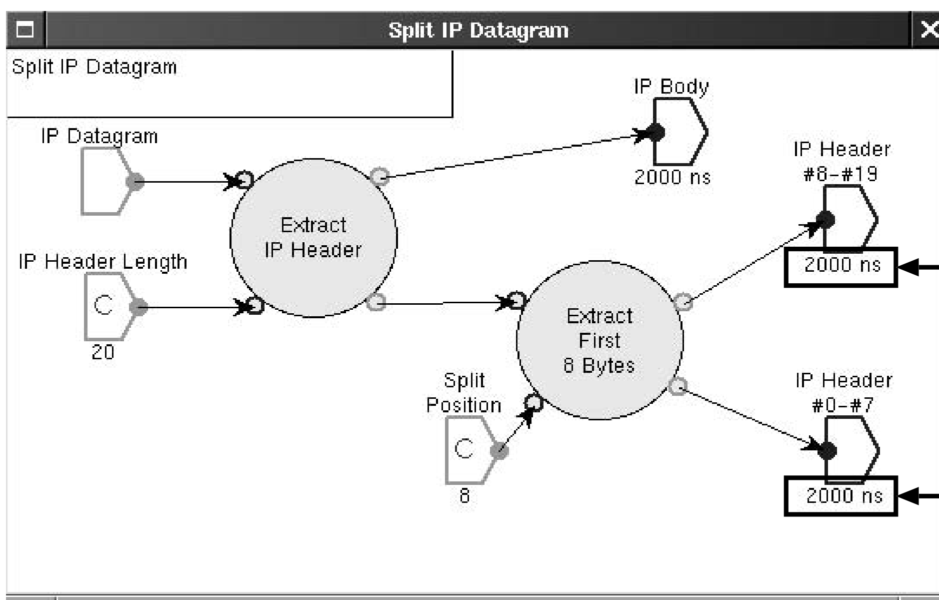
本研究では，これらのうち特に重要であるターンアラウンドタイム，データ流量をとりあげ，それらの仕様記述を導入した．まず，データ流量はストリームに対して定義されるべきものであるので，これを直接的に記述できるようにした．図 4.1 に例を示す．これは，TCP/IP プロトコルにおける IP データグラム (IP Datagram) のデータ構造の仕様記述である．図 4.1 (a) において，IP データグラムを OC-3 (Optical Carrier 3) ATM (Asynchronous Transfer Mode) の最大実効転送レートである 135Mb/s で入力したいというユーザの要求が記述されている．次に，ターンアラウンドタイムに関しては，入力ストリームの先頭のパケットがソースに入力されてから，出力ストリームの末尾のパケットがシンクから出力されるまでの時間として定義し，これをシンクに直接記述できるようにした．図 4.1(b) に例を示す．これは，TCP/IP プロトコル処理の IP 層受信処理部の一部である．この仕様では，入力されてきた IP データグラム (IP Datagram) をまずヘッダ部 (IP Header) とボディ

部 (IP Body) に分割 (Extract IP Header) する．さらに，後の処理の都合のため，ヘッダ部を前半 8 バイト (IP Header #0-#7) と後半 12 バイト (IP Header #8-#19) に分割 (Extract First 8 Bytes) する．この例では，ターンアラウンドタイムは 2000 [nsec] と仕様記述されている．

一方，工学的制約の仕様記述として，CUE プロセッサのパイプラインを構成するステージ水準に至るまでのパイプライン構成の仕様記述を導入した．例を図 4.2 に示す．この図では，CUE プロセッサ CUE-v1 (CUE-version1) が記述されている．まず，ステージを接続して構成される機能ブロック（この例では合流部）が図 4.2(c) に記述されている．この機能ブロックを接続して構成されるプロセッサ (PE)（この例では INT と呼ばれる PE）が図 4.2(b) である．この PE を接続して構成されるチップ（この例では CUE-v1）に関する記述が図 4.2(a) である．

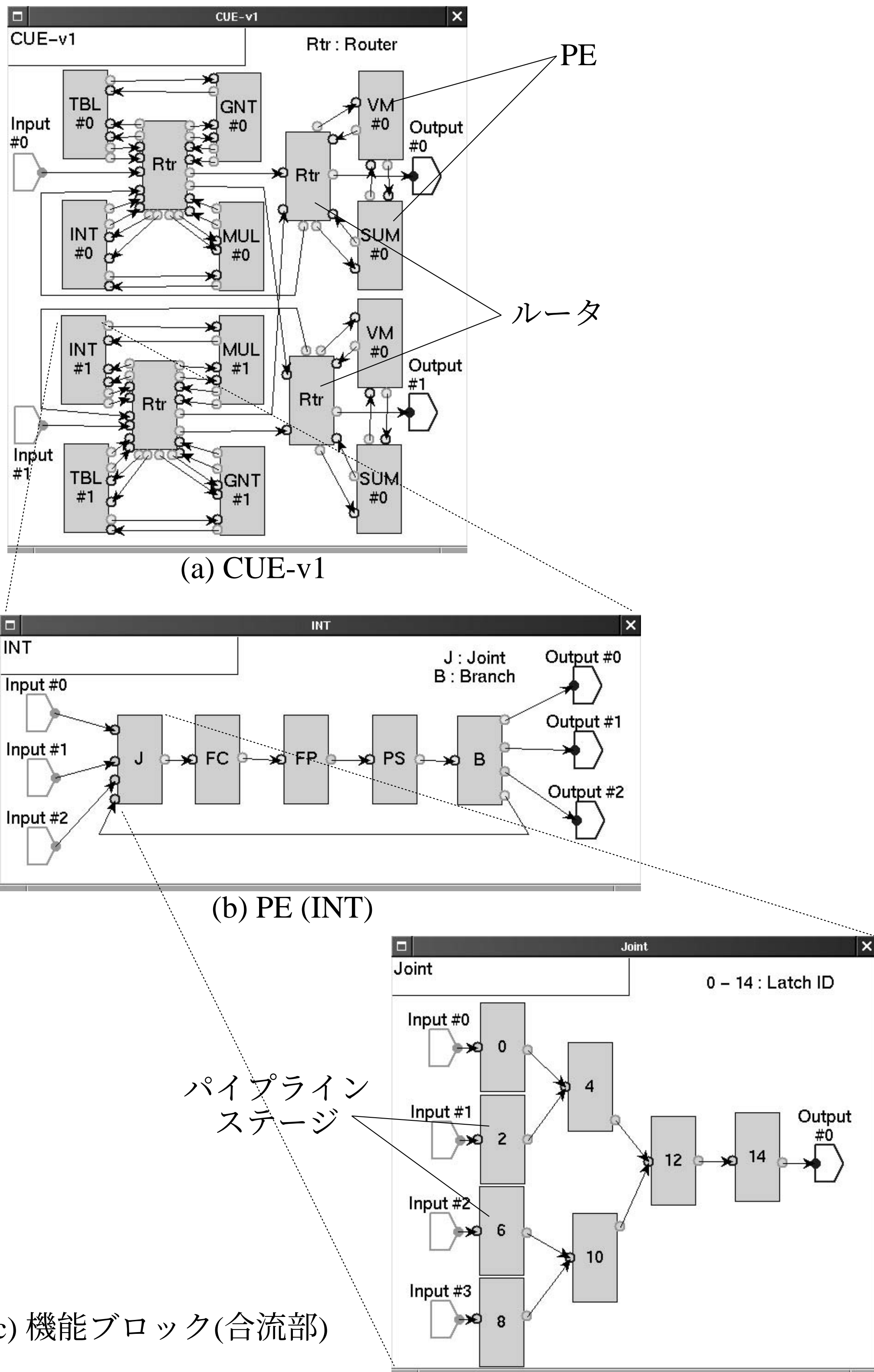


(a) データ構造の仕様記述



(b) アルゴリズムの仕様記述

図 4.1: ユーザの要求に関する仕様記述



(a) CUE-v1

(b) PE (INT)

(c) 機能ブロック(合流部)

図 4.2: 工学的制約に関する仕様記述

4.3 仕様記述環境による性能予測とエミュレータによる性能検証

CUE プロセッサは，同時並行的に実行される複数の独立したプロセス間の相互干渉が発生しない理想的な多重処理能力を有する [52]．そのため，あるプログラムが CUE プロセッサ上で実行される際に要する時間は，プロセッサが過負荷領域に陥らない限り，その実行時にパケットが通過するパイプラインのパイプラインタクト（パケットがパイライン上を 1 ステージ進むのにかかる平均時間）の総和として予測できる．データ流量についても同様に，あるストリームがプロセッサに入ってから出るまでの流れは，命令プリミティブによってストリームが分岐・複製・消去されない限り，且つプロセッサが過負荷領域に陥らない限り，保存される．

従って，本研究では，ターンアラウンドタイムとデータ流量，ならびにプロセッサのパイプライン占有率を，実行可能プログラムの静的解析により予測する．同時に，CUE プロセッサを構成するエラスティックパイプラインのステージ間のハンドシェイク動作をパイプラインタクトの 10 倍程度の精度で模擬するエミュレータ [92] を用いて，プログラム実行時に CUE プロセッサが過負荷領域に陥らないことを保証する．

RESCUE のシステム構成を図 4.3に示す．図 4.3に示すように，仕様記述環境が動作する PC ワークステーションと CUE プロセッサエミュレータが動作する CUE-v1 およびデータ駆動ネットワーキングインタフェースボードとを，IP ルータや ATM スイッチで接続した．RESCUE とエミュレータは，既存の OS (Operating System) 上の TCP/IP プロトコルスタックと，TCP/IP プロトコル処理のデータ駆動型実現法 [55] とを介して通信する．また，RESCUE の内部構成を図 4.4に示す．ユーザから付与された機能仕様・性能仕様を基に，実行可能プログラム，パイプライン構成，入力ストリームの各情報が生成される．これらを用いて性能予測した結果，エミュレーションに必要なデータ流制御情報が生成されるとともに，性能予測結果がユーザに示される．また，プログラム，パイプライン構成，入力ストリーム，データ流制御情報が仕様記述環境からエミュレータに送信されると，エミュレーションが実

行され，その結果として，トレース情報が仕様記述環境に送信される．仕様記述環境はこのトレース情報をもとにして，エミュレーション結果をユーザに提示する．

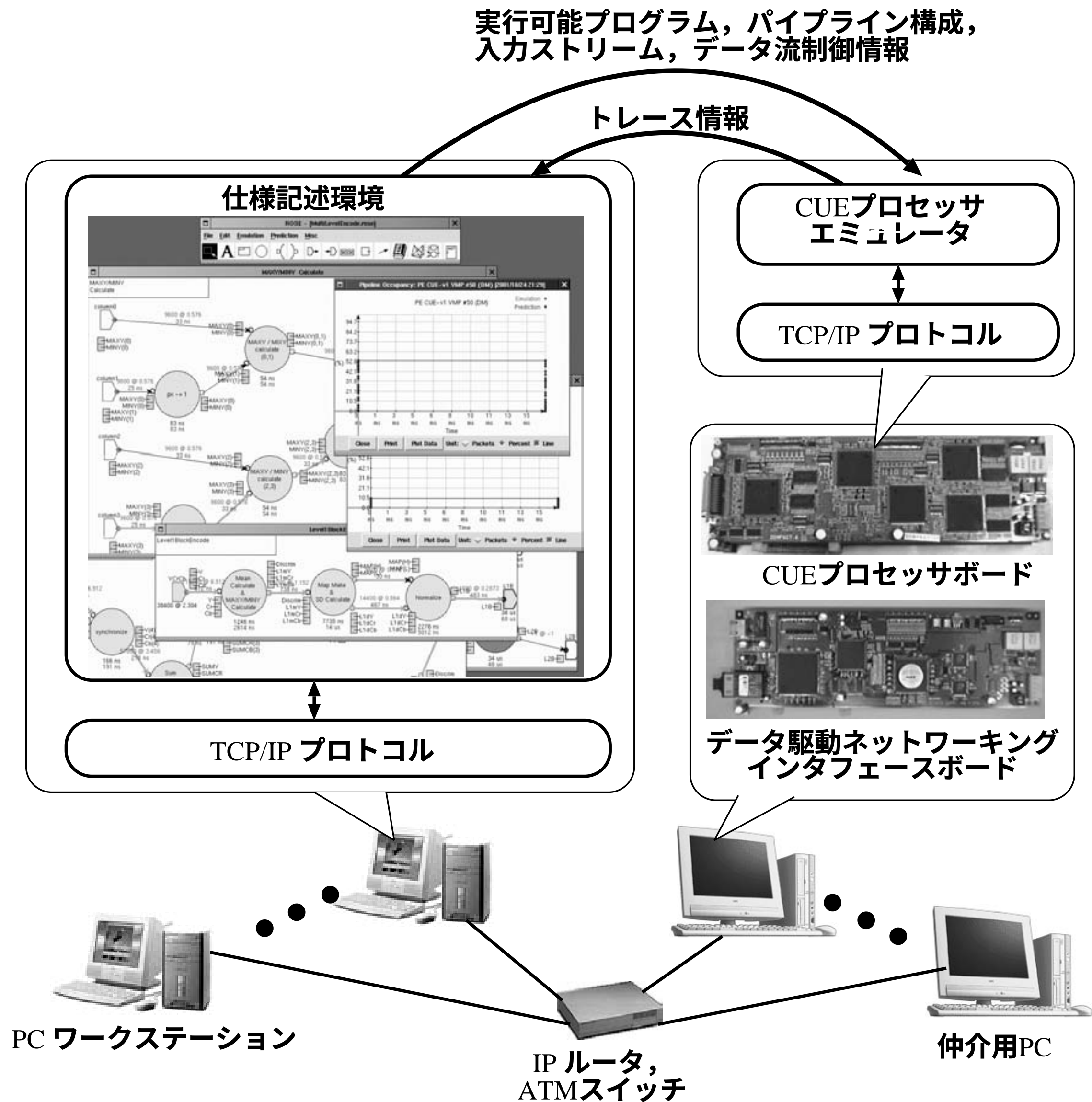


図 4.3: RESCUE のシステム構成

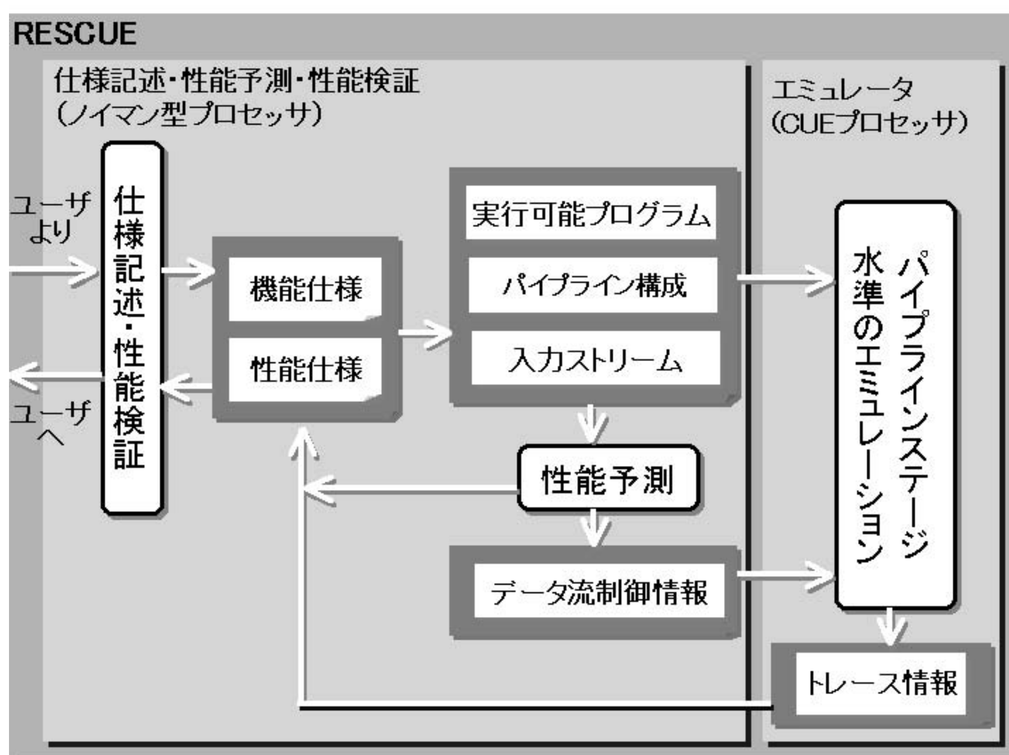


図 4.4: RESCUE の内部構成

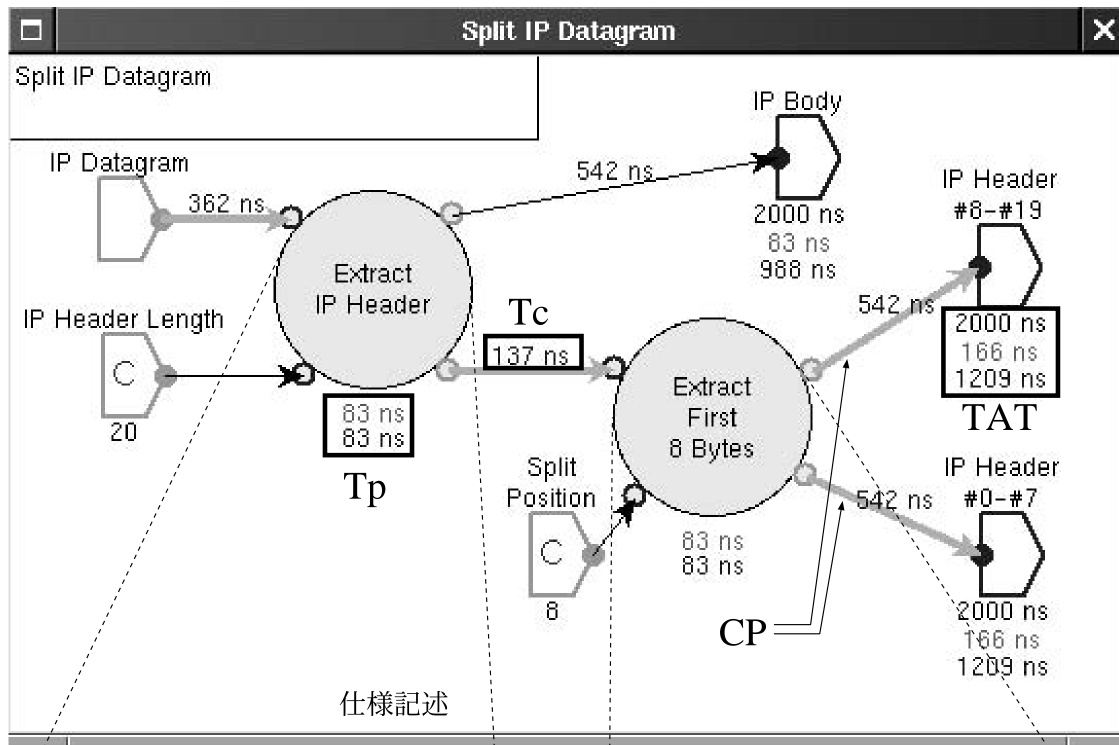
ターンアラウンドタイムの予測機能の実現法

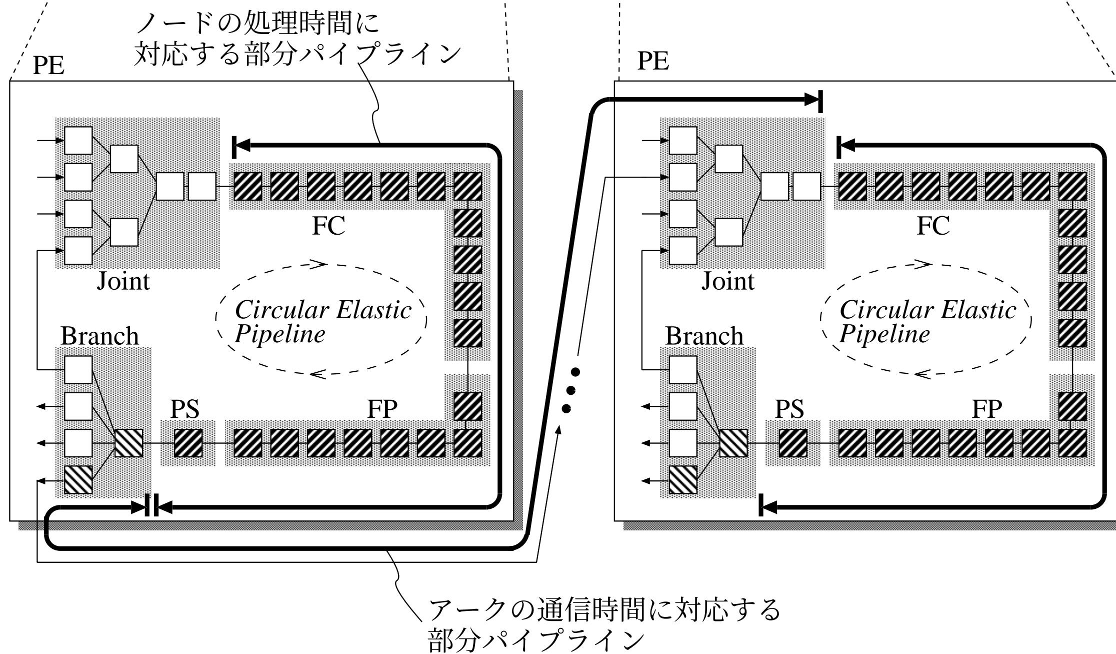
図 4.5に示すように，仕様上のあるノードの機能がプリミティブで実現される場合，そのノードの最短ターンアラウンドタイムはデータ駆動プロセッサ上の FP のパイプライン段を通過するのにかかる時間（=パイプライン段数×パイプラインタクト）である．同様に，仕様上のあるノードの機能が複数のプリミティブからなる部分プログラムで実現される場合，そのノードの最短ターンアラウンドタイムはそれらの FP とそれらの間の通信のために通過するパイプライン段を通過するのにかかる時間である．

生成されたプログラムのターンアラウンドタイムの短縮を支援するためには，ボトルネックとなっている部分がユーザに明示されなければならない．特に，PE 間の通信によるオーバヘッドを明示する必要がある．本研究では，PE 間の通信によ

るオーバヘッドをアークの通信時間として明示するため，ノードの処理時間・アークの通信時間とパイプラインとの対応関係を以下のように定義した．即ち，図 4.5 に示すように，ノードでの処理時間を，あるパケットが FC (Firing Control) に入ってから PS (Program Storage) を出るまでの時間として定義し，また同様にアークでの通信時間を，PS を出てから FC に入るまでの時間として定義した．

さらに本研究では，クリティカルパスおよびその最短ターンアラウンドタイムの情報を含め，図 4.5のように視覚化する．図 4.5に示されている例において，“Extract IP Header” および “Extract First 8bytes” なる 2 つのノードはプリミティブによって実現されており，同一の PE に配置されている．FC, FP(Functional Processor), PS はそれぞれ 11 段，8 段，1 段のパイプラインから構成されているので，FC から PS までは合計 20 段である．また，PS から FC までは，図では省略されているが，この場合はプロセッサ間通信を含めて 合計 33 段である．CUE-v1 のパイプラインタクトは 4.17[nsec] である．以上より，“Extract IP Header” ノードの処理時間は図 4.5(a) に示すように $20 \times 4.17 \approx 83$[nsec] と予測され，“IP Header” アークの通信時間は図 4.5(b) に示すように $33 \times 4.17 \approx 137$[nsec] と予測される．





Tp : ノードの処理時間
Tc : アークの処理時間
CP : クリティカルパス

TAT : ターンアラウンドタイム
上段：ユーザの要求
中段：理想的なターンアラウンドタイム
下段：実際のターンアラウンドタイム

図 4.5: ターンアラウンドタイムの予測手法

クリティカルパス検出アルゴリズムは以下のようである．まず，本アルゴリズムは，あるブロックのクリティカルパスを検出するものである．これには以下の仮定が必要である：

- このブロック内のすべてのノードの処理時間とすべてのアークの通信時間が既に予測されている．

RESCUE では，プリミティブもブロックも割り当てられていないノードには，デフォルトのプリミティブ（現在は 2 入力加算命令）が割り当てられる．また，PE が割り当てられていないノードは，デフォルトの PE に配置される．RESCUE は，これらのデフォルト値を用いてつねに積極的にノードとアークのターンアラウンドタイムを予測するので，この仮定はつねに成立する．

RESCUE において DDS でのすべての要素（ソース，シンク，ノード，入出力ポート，アーク）は，それぞれ一意の識別子を内部的に持っている：ソース ID，シンク ID，ノード ID，入力ポート ID，出力ポート ID，アーク ID．**route** は，あるシンクからあるソースまでの経路である．クリティカルパスは，ブロックでの 1 つ以上の **route** である．それぞれの **route** は以下の情報を保持している：

routeID この route の一意の識別子

nextArc 次回訪問すべきアーク ID

tat 現在のターンアラウンドタイム

idealTat 現在の理想的なターンアラウンドタイム

path この **route** を表現するための線形リスト．1 つのシンク ID，0 個以上のアーク ID，1 つのソース ID からなる．

0° まず，以下のようにいくつかの変数を初期化する：

maxTat 現在の最大ターンアラウンドタイム．初期値 0.

maxIdealTat 現在の最大理想ターンアラウンドタイム．初期値 0.

crPath 現在の処クリティカルパスである経路の経路 ID.

toDo path が未確定の経路の経路 ID の FIFO.

次に，ブロック内のすべてのシンクに接続されている個々のアークについて以下を行う：

1. 新規に経路 ID を生成.
2. この経路に関する以下の情報を初期化する.
   - **nextArc** にこのアークのアーク ID を代入.
   - **tat** に 0 を代入.
   - **idealTat** に 0 を代入.
   - **path** に，このシンクのシンク ID と，このアークのアーク ID からなる線形リストを代入.
3. 経路 ID を **toDo** に追加.

1° 以下を実行する.

1. もし **toDo** が空でないならば，**toDo** の先頭から 1 個取り出し，これを **curRoute** とする．もし **toDo** が空ならば，**2**° に行く.
2. 以下の場合には，**curRoute** の **nextArc** のターンアラウンドタイムを **tat** に加算する：
   - **nextArc** にソースもシンクも接続されていない．または,
   - **nextArc** にソースかシンクが接続されていて，かつ，このブロックは階層的仕様における最上位ブロックである.
3. もし **nextArc** がソースに接続されていて **tat** > **maxTat** ならば,
   以下を実行した後に **1**° へ行く.
   - **maxTat** に **tat** を代入.
   - **maxIdealTat** に **idealTat** を代入.

- **crPath** に **curRoute** を代入.

4. **nextArc** の始点であるノードの予測済ターンアラウンドタイムを，**tat** に加算する.
5. そのノードの入力ポートに接続されているすべてのアークのリストを **arcList** とする．もし当該アークが 2 本以上あるならば，1 個を残して **arcList** から取り出し，それらのアーク ID について以下を実行する：
   - 新規に経路 ID を生成する.
   - **curRoute** が持つ **tat**, **idealTat**, **path** の情報を，その経路 ID の情報にコピーする.
   - **nextArc** をそのアーク ID とする.
   - **toDo** に アーク ID を追加する.
6. **nextArc** に，**arcList** 内のアーク ID を代入する．**1°** へ.

2° 終了．このときの **crPath** の内容はクリティカルパスを示している．現在の **maxTat** と **maxIdealTat** の内容は，クリティカルパスの実際のターンアラウンドタイムと理想的なターンアラウンドタイムである.

ターンアラウンドタイムの予測コストの低減法

対話的な性能評価を実現するためには，予測コストの低減が重要である．本研究では，ターンアラウンドタイムの予測を以下のようにインクリメンタルに実行する．即ち，あるブロック β があるノード ν に割当てられているとする．ある β のターンアラウンドタイム τ_β が新規に予測されたかもしくは変更された場合，τ_β を，ν の実行時間 τ_ν として記憶する．そして，ν が記述されているブロックのターンアラウンドタイムの更新時などを例として，ν の実行時間が必要とされたときは，τ_β を再計算せず，τ_ν を利用する.

より正確には，既に述べたように，本研究では仕様上のすべてのアークは実行可能プログラム上のアークと対応する．従って，図 4.6に示すように，β_1 のソースと

繋がっているアークと，ν_0 の入力アークとは，プログラム上で同一のアーク a_0 に対応する．同様に，β_1 のシンクと繋がっているアークと，ν_0 の出力アークとは，プログラム上で同一のアーク a_6 に対応する．この性質から，$\tau_\beta = \tau_\nu$ とはならない．τ_ν と τ_β の関係は正確には以下のようである．

$$\tau_\nu = \tau_\beta - \tau_{\beta i} - \tau_{\beta o}$$

ただし，$\tau_{\beta i}$ は τ_β のうちブロック β 内のソースから接続されているアークに対応する通信時間である．同様に，$\tau_{\beta o}$ は，τ_β のうちブロック β 内のシンクへ接続されているアークに対応する通信時間である．図 4.6の例では,

$$\begin{cases} \tau_{\beta i} & = & \tau_{a0} \\ \tau_{\beta o} & = & \tau_{a6} \end{cases}$$

である．

この方式によれば，あるブロック β のターンアラウンドタイムの更新コストは，そのブロックが実際にどの程度の規模のプログラムで実現されているかには無関係である．即ち，β の仕様として記述された DDS の規模に依存する．2 章で述べたように，1 ブロックあたりに記述できる DDS の規模には自ずから限界があるのであって，本低減手法によればその程度に予測コストを抑制できる．

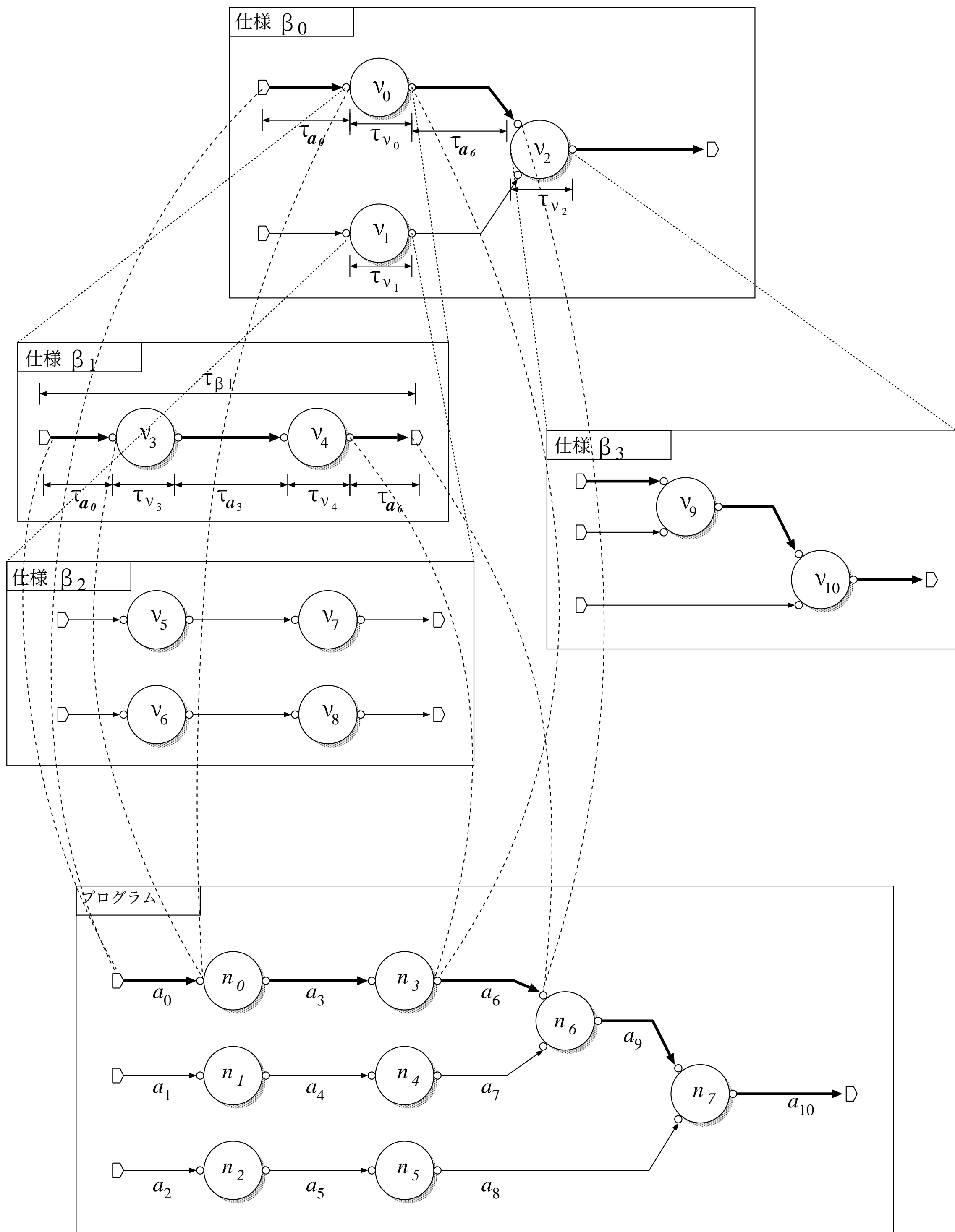


図 4.6: ターンアラウンドタイム予測コストの低減法

ターンアラウンドタイムの検証支援機能の実現法

エミュレーション結果からのターンアラウンドタイムの検証を実時間で行えるようにするためには，ノードの処理時間およびアークの通信時間を，可能な限り早い時点で更新する必要がある．前項で述べたように，本研究では，ノードでの処理時間を，あるパケットが FC に入ってから PS を出るまでの時間として定義しており，また同様にアークでの通信時間を，PS を出てから FC に入るまでの時間として定義している．従って，これらの更新処理を以下のように実現した．即ち，エミュレータから仕様記述環境に送られてきたあるパケット情報が，1)FC（または PS）へ入力されたパケットに関するものだった場合，仕様記述環境はそれを記録する．さもなくば，2)PS（または FC）から出力されたパケットに関するものだった場合，仕様記述環境は，そのタイムスタンプと，そのパケットが FC（または PS）に入力されたときのパケット情報のタイムスタンプとの差を計算し，これをノードでの処理時間（またはアークでの通信時間）として表示する．

パイプライン占有率の予測機能の実現法

本研究では，図 4.7に示すような規則に従って，実行可能プログラム上の個々のアークについて，定常的なデータ流量を予測する．即ち，データ流量の予測結果は，実行可能プログラム上の個々のアークに対して定義された 1 つの数値として表現される．

まず，システムの外部との定常的な入力データ流量は，ユーザの要求の一部として仕様に記述されているものとする．また，仕様上のあるノードが 2 つ以上の出力ポートを持ち，それらの出力ポートから定常的には異なるデータ流量が発生する場合，入力データ流量に対する出力データ流量の比率（分岐レート）が，0 以上 1 以下の値として，出力ポートに対して仕様に記述されているものとする．データの複製を実行するノードの場合，すべてのポートの分岐レートが 1 である．分岐レート 0 はデータ消失 (absorb) を意味する．

あるプリミティブノードのまわりの DDS は，典型的には，図 4.7に示す 3 種類のいずれか，あるいはそれらの組み合わせである．図 4.7(a) は，入力アークがマー

ジ構造をとっておらず，なおかつ，出力アークがコピー構造をとっていないものである．図 4.7(b) は，入力アークがマージ構造をとっているものである．図 4.7(c) は，出力アークがコピー構造をとっているものである．

図 4.7(a) の場合，出力アークのデータ流量は，入力アークのデータ流量のうち最も小さなものに制約される．従って，ある出力アークのデータ流量は，すべての入力アークの持つデータ流量のうち最小の値に，その出力アークに繋がっている出力ポートが持つ分岐レートを乗じたものとして予測される．例えば，出力アーク a_2 のデータ流量 F_2 は，

$$F_2 = \min(F_0, F_1) \times B_0$$

となる．min は，引数のうちの最小値を与える関数，F_i は入力アーク i のデータ流量，B_0 は分岐レート $(0 \leq B_0 \leq 1)$ である．

図 4.7(b) のように，2 本以上のアークが 1 つの入力ポートへマージしている場合，この入力ポートに対するデータ流量は，それらのアークのデータ流量の合計であると見なせる．従ってこの場合，ある出力アークのデータ流量は，すべての入力アークの持つデータ流量の合計に，その出力アークに繋がっている出力ポートが持つ分岐レートを乗じたものとして予測される．例えば，アーク a_2 のデータ流量 F_2 は，

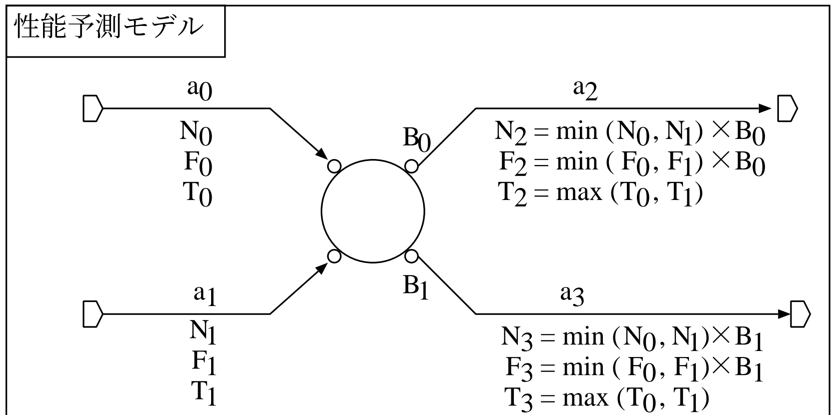
$$F_2 = (F_0 + F_1) \times B_0$$

となる．

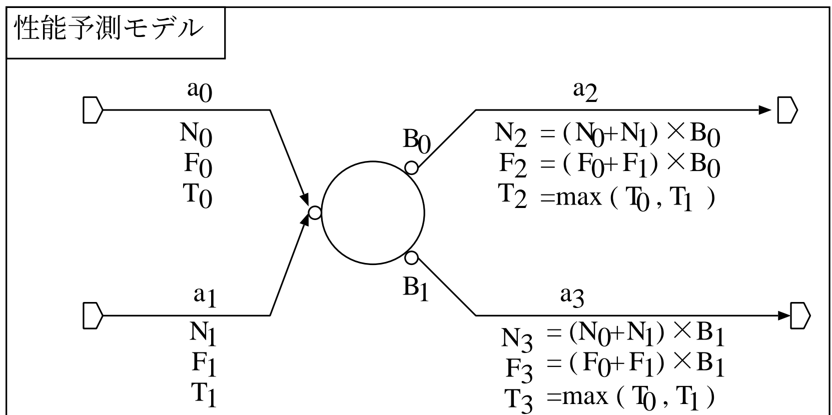
図 4.7(c) のようなコピー構造は，前述のように，分岐レートが 1 であることと等価である．例えば，出力アーク a_2 のデータ流量 F_2 は，

$$F_2 = \min(F_0, F_1)$$

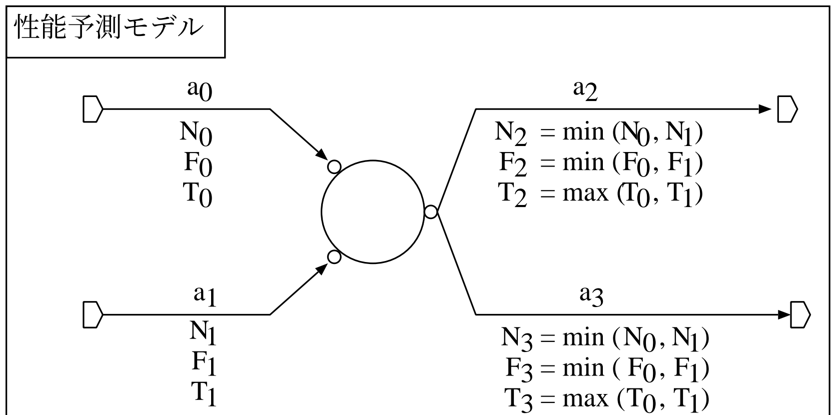
となる．



(a) 入力にマージ構造がなく，出力にコピー構造がない場合



(b) 入力にマージ構造があり，出力にコピー構造がない場合



(c) 入力にマージ構造がなく，出力にコピー構造がある場合

a_i : アーク識別子
N_i : パケット数
F_i : データ流量
T_i : ターンアラウンドタイム
B_i : 分岐レート ($0.0 \leqq B_i \leqq 1.0$)
min (m,n) : m と n の小さい方を返す.
max (m,n) : m と n の大きい方を返す.

図 4.7: データ流量の予測ルール

次に，プリミティブに対応付けられているあるノードが，あるプロセッサに割り当てられていることによって実行時にプロセッサに発生するパイプライン占有率は，ノードへの入力データの仕様から，図 4.8のように算出できる．まず，ある一定のデータ流量を持つ入力データがプロセッサのパイプラインに入力されるので，パイプライン占有率 r は，区間 $t = [t_0, t_1]$ において 0 から開始して一定の傾き ρ/I_1 で増加する．ここで，$I_0 (= t_0)$ は，入力データがこのノードに入力されるまでの時間 [nsec] である．また，t_1 は，入力データの先頭パケットがプロセッサのパイプラインから出力され始める時刻である．プロセッサのパイプライン段数 s[段] とパイプラインタクト δ[nsec] とすると $I_1 = s \times \delta$ である．区間 $t = [t_1, t_2]$ では，入力されるデータ流量と出力されるデータ流量が等しくなる．このためパイプライン占有率 r は区間 $t = [t_1, t_2]$ において一定の値 ρ をとる．ρ は，区間 $t = [t_1, t_2]$ においてパイプラインステージ上に存在するパケット数をステージ段数で割った値である．$I_2 (= t_2 - t_0)$ は，入力データの先頭のパケットが入力されてから最後尾のパケットが入力されるまでの時間 [nsec] である．パケット数を n[個]，データ流量を R[MPacket/sec] とすると $I_2 = n/(R \times 10^3)$ である．また，t_2 は，入力データの最後尾のパケットがパイプラインに入力された時刻である．区間 $t = [t_2, t_3]$ では，区間 $t = [t_0, t_1]$ の逆に，パイプライン占有率 r は，ρ から開始して一定の傾き ρ/I_1 で減少し，従って時刻 $t_3 (= t_2 + I_1)$ に 0 となる．

IP 層受信処理プログラムでの例を図 4.8に示す．この例では，30 バイト長の “IP Datagram” なるデータは，先頭 20 バイトのヘッダ部 (“IP Header”) とそれに続く 10 バイトの本体部 (“IP Body”) からなる構造を持つことが仕様上に記述されている．“Extract IP Header” ノードには，“IP Datagram” を “IP Header” と “IP Body” とに分割する機能を担っていることが仕様として記述されている．この例では，“IP Datagram” なるデータが 135[Mb/s] で入力される．現在の TCP/IP プロトコル処理プログラムは 1 パケットにデータを 8[bit] ずつ格納した状態で扱うよう設計されているので，135[Mb/s] は約 17[MPacket/s] に相当する．CUE-v1 の場合，パイプラインタクトは平均約 4.17[nsec] と見なすことができる．従って，約 17[MPacket/s] のデータ流量は，約 14[tact] 毎に 1 パケット入力されることを意味し，これは約 7%

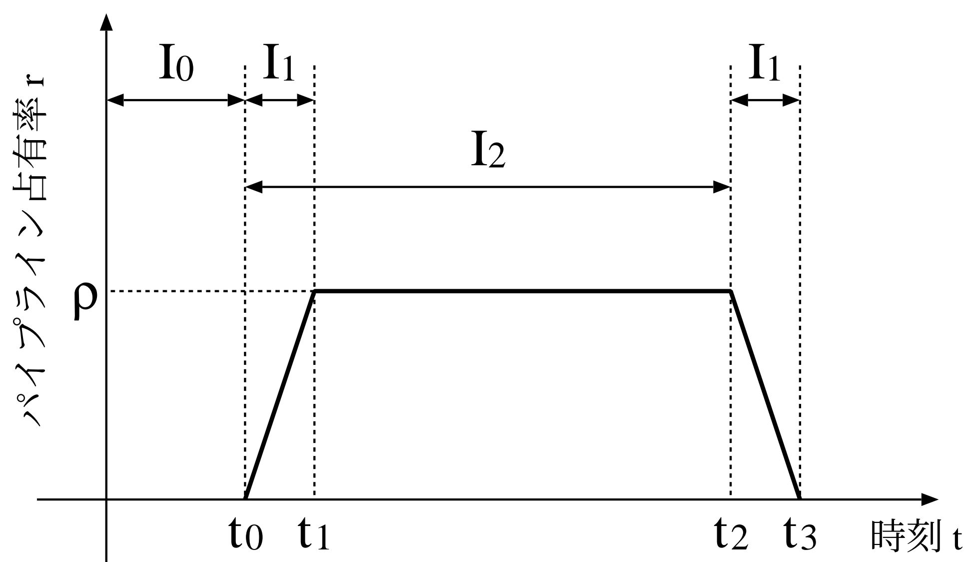


図 4.8: 命令プリミティブ 1 個の実行時のパイプライン占有率の予測手法

のパイプライン占有率に相当する．この例で使用している PE の FC から PS までのパイプライン段数は 20 段 である．図 4.9(b1) に示すように，“Extract IP Header” に関しては，パイプライン占有率のグラフが最初にある一定の傾きで増加する区間の幅は，20[段] × 4.17[nsec]=83[nsec] となる．このノードにデータ “IP Datagram” の先頭のパケットが到着する時刻が 362.7[nsec] であるから，この区間が終了する時刻は $362.7 + 83 \approx 446$[nsec] となる．次に，一定のパイプライン占有率を維持するフェーズは，最後となる 30 個目のパケットが入力されるまでに 1769.5[nsec] かかることから，$362.7+1769.5 \approx 2132$[nsec] までとなる．この時刻からパイプライン占有率が減少し 0 に至るまでには再び 83[nsec] かかるので，パイプライン占有率が 0 となる時刻は $2132+83 = 2215$[nsec] となる．図 4.9(b2) に示すように，“Extract First 8 Bytes” ノードに関しても基本的に同様である．このノードは “Extract IP Header” の出力ストリームが到着し始める時刻 583[nsec] からグラフが開始する点と，このノードを通過するデータは “IP Datagram” のうちの “IP Header” のみであるので，データ長が 30 から 20 に減少しており，従って，パイプライン占有率が一定となっている時間が約 1096[nsec] と短い点が異なる．

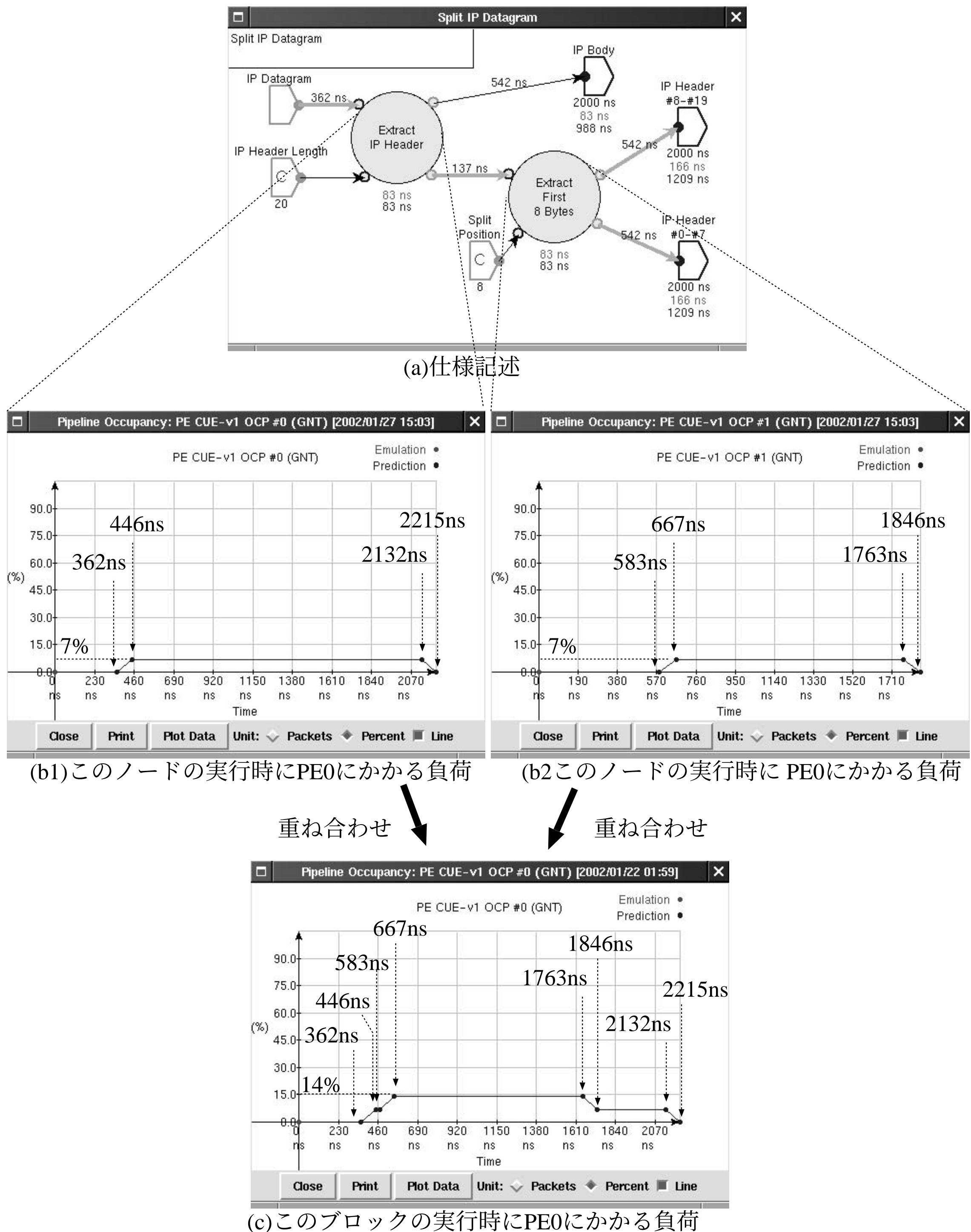


図 4.9: パイプライン占有率の予測例

従って，パイプライン占有率の予測には，既に述べたデータ流量の予測結果の他に，

- プリミティブノードにおける処理時間
- アークにおける通信時間
- アーク上を流れるパケット数

の情報が必要である．これらのうち，プリミティブノードにおける処理時間ならびにアークにおける通信時間については，前節に述べたように予測される．アーク上を流れるパケット数については，データ流量とまったく同様に，図 4.7(a)〜(c) に示すように予測される．即ち，図 4.7(a) については

$$N_2 = \min(N_0, N_1) \times B_0$$

となり，図 4.7(b) については

$$N_2 = (N_0 + N_1) \times B_0$$

となる．また，図 4.7(c) については，

$$N_2 = \min(N_0, N_1)$$

となる．

以上により，あるプリミティブノードの実行時にそのプリミティブノードが割当てられているプロセッサに発生するパイプライン占有率が予測される．この予測結果と，プログラム分散配置の情報とに基づいて，プログラム実行時のプロセッサのパイプライン占有率を以下のようにして算出する．即ち，あるプロセッサに実行時にかかるパイプライン占有率は，そのプロセッサに配置されているノードへの入力により生じるパイプライン占有率の和である．従って，あるプロセッサに配置されているノードについて予測されたパイプライン占有率のグラフをすべて重ね合わせることで，図 4.9 のように，実行時のプロセッサのパイプライン占有率を算出できる．図 4.9に示した例では，“Extract IP Header” ノードと “Extract First 8bytes”

ノードが同一の PE に配置されているとすると，プログラム実行時にこの PE に要求されるスループットは区間 $t = [667, 1763]$[nsec] において最大 14 %となっている．

ここで，グラフをタクトごとのパイプライン占有率を用いて内部的に表現していると，グラフを重ね合わせる際に，タクトごとにパイプライン占有率を計算しなければならない．その場合，重ね合わせの計算コストは，プログラム実行にかかるタクト数に依存するために，RESCUE の対話性を維持するのが原理的に困難となる．従って，グラフを重ね合わせる際の計算コストを極小化するために，グラフを線分の集合として表現することを考えた．具体的な表現法としては，以下の 3 つの場合が挙げられる．

(1) 座標 (x_i, y_i) による表現 一方のグラフ A の点 P_i の x 座標 x_i を他方のグラフ B において定義域 D に含む線分の傾き k_j を算出し，線分 $P_{i-1} - P_i$ の傾き k_i と k_j の和 k_l を求める．重ね合わせ後の終点の y 座標 y_{i+1} を，始点の座標と k_l と x_i から求める．計算コストは，2 整数の加減乗除および比較演算に要するコストを 1 とすると，D の算出に平均 $\log_2 5$，k_j の算出に 3，k_i の算出に 3，y_{i+1} の算出に 3 となるので，合計 $\log_2 5 + 3 + 3 + 3 \approx 11.3$ である．

(2) ある点の x 座標とその点を始点とする線分の傾きの組 (x_i, k_i) による表現 一方のグラフ A の点 P_i の x 座標 x_i を他方のグラフ B において定義域 D に含む線分の傾き k_j を用いて，線分 $P_{i-1} - P_i$ の傾き k_i と k_j の和 k_l を求める．計算コストは，同様に，D の算出に平均 $\log_2 5$，k_l の算出に 1 となるので，合計 $\log_2 5 + 1 \approx 3.3$ である．

(3) ある点の x 座標とその点での傾きの変化量の組 $(x_i, \Delta k_i)$ による表現 図4.10に示すように，グラフ A $(= ((p_0, \Delta k_0), \cdots, (p_i, \Delta k_i), \cdots))$ とグラフ B $(= ((q_0, \Delta l_0), \cdots, (q_j, \Delta l_j), \cdots))$ とする．ある i, j に関して $(p_i = q_j)$ 即ち x 座標が同一の節点については傾きの変化量の和 $(\Delta k_i + \Delta l_j)$ が，$(p_i \neq q_j)$ 即ち x 座標が同一でない節点については Δk_i が，重ね合わせ後のグラフのその x 座標の傾きの変化量となる．従って，この場合の計算コストは，Δk_i の算出に 1 または 0 となるのみである．

以上より，方法 (3) の計算コストが最小であるので，本研究ではこれを採用した．

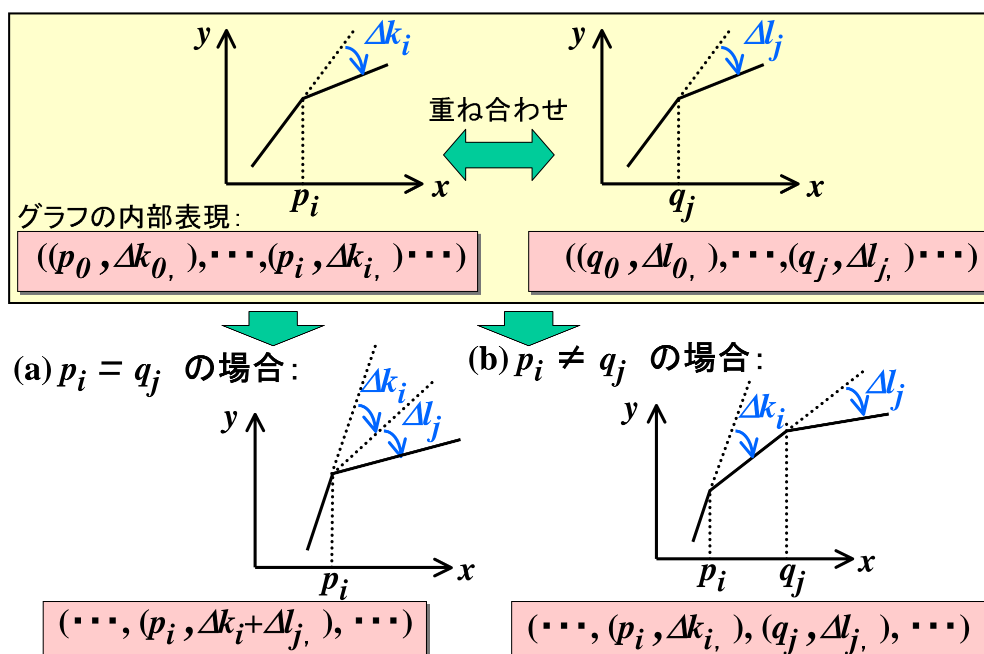


図 4.10: 重ね合わせの計算コスト低減法

パイプライン占有率の検証支援の実現法

パイプライン占有率に関する検証を実時間で実現するための方法としては，検証結果の更新頻度の最も高い方法として，エミュレータからパケット情報が到着する毎にその時点までの検証結果を表示する方法が考えられる．本研究ではこれを以下のように実現した．即ち，あるプロセッサに入力されるパケットに関するパケット情報がエミュレータから送られてきた際に，仕様記述環境はそのパケット情報を単純に記憶する．また，あるプロセッサから出力されるパケットに関するパケット情報がエミュレータから送られてきた際に，そのパケットがそのプロセッサに入力された時のパケット情報から入力時刻を求める．次に，それらの時刻の間の時間帯に

対応するパイプライン占有率をパケット 1 個に相当する分だけ増加させる．そして，パイプライン占有率のグラフを再度描画する．

4.4 音声・動画像圧縮プログラムへの適用を通じた評価

RESCUE を用いて，音声圧縮プログラム，ならびに動画像圧縮プログラムを開発した．音声圧縮としては，携帯電話・PHS 等で用いられている 16bit ADPCM (Adaptive Differential Pulse Code Modulation)[93] をとりあげた．また，動画像圧縮としては，画素をブロック化した後に標準偏差や平均を用いてハフマン符号化を施すアルゴリズムをとりあげた．

まず音声圧縮については，携帯電話相当の 8KHz の品質では実時間処理可能であるが，CD(Compact Disc) 相当の 44.1KHz の品質では実時間処理できないという性能予測結果が得られた．そのため，CUE-v1 のパケットのデータ幅を現行の 12bit から 16bit に拡張したアーキテクチャを前提とする実行可能プログラムを生成したところ，実時間処理可能との性能予測結果が得られた．次に，動画像圧縮については，24bit フルカラー（RGB 各 8 ビット）・VGA （640×480 ピクセル）サイズ・15 フレーム/秒の入力動画ストリームを処理することが要求された．その結果，圧縮結果の出力処理で逐次的アルゴリズムが不可避なためにボトルネックとなることが，ターンアラウンドタイムの予測結果を用いて特定された．また，負荷の予測結果を用いて，プログラム分散配置やプログラム構造の改善を支援できた．これらの改善により，最終的に，要求が満足可能であることが RESCUE で予測され，なおかつ，CUE-v1 の実機で実行した結果，期待通りに動作することが確認された．また，クロック周波数 1GHz 程度の PC (Personal Computer) 上で動作する UNIX での実測結果からは，性能予測は約 3 秒以内に完了するとの結果が得られた．

以上より，RESCUE の対話的な性能評価支援の有効性が示された．加えて，RESCUE が，CUE プロセッサアーキテクチャのチューニング支援にも有効であることが示された．

4.5　結言

本章では，データ駆動型実時間システムの開発を支援するためのプロトタイピング手法を提案した．まず，ユーザの要求に関する仕様記述手法として，データ流量・ターンアラウンドタイムの仕様記述手法を提案した．データ流量は，データストリームに対して定義されるべきものであるので，これを直接的にデータストリームに対して記述できるようにした．ターンアラウンドタイムは，ストリームがある処理部に入力されてから出力されるまでの時間であるので，ブロックの出力を示すソースに対してこれを直接的に記述できるようにした．また，工学的制約の仕様記述として，パイプライン構成の仕様記述手法を提案した．この手法では，最も低い水準としてパイプラインステージ間の接続構成を記述できる．それより上の水準では順に，機能ブロック，プロセッサ，チップマルチプロセッサ，プロセッサボード，システムといった階層的システム構成を記述できる．

次に，これらの性能仕様・システム仕様に基づくプロトタイピング機能を実装した RESCUE における，性能予測・検証手法とその実現法を示した．まず，パケットが通過するパイプラインステージと各ステージの平均転送時間とから，実行可能プログラム中のクリティカルパスを求める手法を述べた．次に，このクリティカルパスの情報を用いてターンアラウンドタイムを予測する手法を述べた．クリティカルパスのターンアラウンドタイムは，クリティカルパスに対応するパイプラインステージの総転送時間として予測可能である．次に，仕様上に記述されたデータ流量の情報と，プログラムのデータ依存性とを解析して，プログラムの各アーク上を流れる定常的なデータ流量を予測する手法を述べた．さらに，生成されたプログラムが実行可能であることを確認するために，ターンアラウンドタイムとデータ流量の予測結果を用いて，プロセッサのパイプライン占有率を予測する手法を述べた．

本プロトタイピング手法を，音声圧縮・動画像圧縮アプリケーションに適用して評価した．その結果，RESCUE によって，性能評価を対話的に行えること，ボトルネックを検出支援できたこと，ならびに，性能予測結果が，プログラム分散割当の変更による性能改善の指針となったことを示した．さらに，性能予測に要する時間

の実測結果からは，対話的に性能予測可能との結果が得られたことを示した．以上より，本プロトタイピング手法が，実時間システムの開発支援に対して有効であることが示された．

第 5 章

結論

本研究では，将来の通信インフラストラクチャを実現するためのデータ駆動型実時間システムの開発を支援するために，仕様記述から実行可能プログラム，実行システムに至るまでの統一的なデータ駆動パラダイムに基づいたプロトタイピング手法を明らかにすることを目的とした．

第 2 章では，記号実行に基づくプロトタイピングを介してユーザとシステムが対話的に実行可能なデータ駆動プログラムを直接生成することで，副作用を検出支援する手法を提案した．まず，本研究では仕様記述の階層構造を必要に応じて再構成することを前提にすることを述べた．次に，副作用検出を支援するためには，仕様記述とプログラムのデータ依存性に関する対応関係を維持することが重要であることを述べた．最後に，副作用を仕様記述上に提示する際にユーザの了解性を維持するために必須である，データ依存性の畳み込み手法を示した．

第 3 章では，記号実行に基づくプロトタイピング手法の実現法を提案し，その評価を通じて本プロトタイピング手法の有効性を示した．まず，記号実行の実現法として，LISP 言語でいう万能関数 eval を用いて記号実行を実現する手法を述べた．次に，対話的なプロトタイピングを実現するためには，記号実行の計算コストの低減が重要となるので，過去の記号実行結果を活用することによって，記号実行のコストを，DDS の一階層の規模程度に抑制する手法を述べた．これによって，仕様記述上の任意のノードは，それを実現する部分プログラムの規模に関わらず，eval 機

構による評価 1 回のみでプロトタイピングできるようにした．次に，ユーザのオペレーションに対する記号実行の起動条件を示した．この起動条件によって，ユーザが仕様記述を変更した時にはつねに記号実行結果が提示されるようになっている．また，記号実行に基づくプロトタイピング機能を実装した仕様記述環境について，その構成を示した．

次に，本手法を TCP/IP プロトコル処理プログラムの中の IP 層受信処理部の開発に適用して評価した結果を述べた．まず，本評価で発生し得る副作用・誤りの原因が，世代操作の副作用による世代の不一致か，または仕様記述とプログラムの間の対応づけの誤りのいずれかであることを示した．次に，本手法を用いて IP 層受信処理部のプログラムを生成した過程において発見された副作用や誤りを示した．その結果，少なくともユーザが要求仕様を完全に正しく仕様記述に反映するか，または，記号実行結果をユーザが必ず活用する限り，タグ操作による副作用の検出支援のみならず，仕様の抜け・誤りの検出支援にも本手法が有効であることを示した．

第 4 章では，データ駆動型実時間システムの開発を支援するためのプロトタイピング手法を提案した．まず，ユーザの要求に関する仕様記述手法として，データ流量・ターンアラウンドタイムの仕様記述手法を提案した．データ流量は，データストリームに対して定義されるべきものであるので，これを直接的に記述できるようにした．また，ターンアラウンドタイムは，ストリームがある処理部に入力されてから出力されるまでの時間であるので，ブロックの出力を示すシンクに対し直接的に記述できるようにした．また，工学的制約の仕様記述として，パイプライン構成の仕様記述手法を提案した．この手法では，最も低い水準としてパイプラインステージ間の接続構成を記述できる．それより上の水準では順に，機能ブロック，プロセッサ，チップマルチプロセッサ，プロセッサボード，システムといった階層的システム構成を記述できる．

次に，これらの性能仕様・システム仕様に基づくプロトタイピングのための，性能予測・検証手法とその実現法を述べた．まず，パケットが通過するパイプラインステージと各ステージの平均転送時間とから，実行可能プログラム中のクリティカルパスを求める手法を述べた．次に，このクリティカルパスの情報を用いてターンア

ラウンドタイムを予測する手法を述べた．次に，仕様上に記述されたデータ流量の情報と，プログラム構造とを解析して，プログラムの各アーク上を流れるデータ流量を予測する手法を述べた．さらに，生成されたプログラムが実行可能であることを確認するために，ターンアラウンドタイムとデータ流量の予測結果を用いて，プロセッサのパイプライン占有率を予測する手法を述べた．

さらに，本プロトタイピング手法を実装した RESCUE を，音声圧縮・動画像圧縮アプリケーションに適用して評価した結果を述べた．その結果，RESCUE によって，性能評価を対話的に行えること，ボトルネックを検出支援できたこと，ならびに，性能予測結果が，プログラム分散割当の変更による性能改善の指針となったことを示した．さらに，性能予測に要する時間の実測結果からは，対話的に性能予測可能との結果が得られたことを示した．以上より，本プロトタイピング手法が，実時間システムの開発支援に対して有効であることが示された．

今後に残された課題としては，以下に述べるようなものが挙げられる．

本論文では，プログラムの初期開発時について検討したが，ソフトウェアのライフサイクルを考えると，初期開発に要した期間よりも，改変を加えながら使い続けていく期間の方が長いと考えられる．今回の評価に使用した TCP/IP プロトコル処理プログラムを部分的に再利用しながらのアプリケーション開発が可能となったので，今後，再利用を主体としたプログラム開発における支援手法についても検討する必要がある．また，本論文では，システム開発工程のうち主としてシステム設計工程とその下流工程を対象とし，要求分析工程は対象としていないかった．しかしながら，要求分析工程においても，要求を入出力データ間の関係として表現する限り，本論文で述べた手法を用いて，要求の洗練化が支援できると考えられる．また，本論文では，仕様に記述される内容としては設計目的 (What) に着目し，設計理由 (Why) については対象としなかった．しかしながら，浜田らは，仕様記述過程において設計理由を逐一データベース化することにより，後に設計変更が必要となった場合に修正すべき箇所を検索・明示する手法を提案している [78]．本研究においても，仕様記述に設計理由 (Why) を導入し，これを副作用検出支援に活用することは意義があると考えられる．

本論文で提案したプロトタイピング手法は，データ駆動プロセッサのプログラム生成のみならず，従来のノイマン型プロセッサのプログラム開発にも応用できる．データ依存性をレジスタやメモリを介した受け渡しで実現し，これが正しく実行されるよう制御依存性を付加することによって，ノイマン型プロセッサ用のプログラムも生成できると考えられる．

本論文で取りあげたような TCP/IP プロトコル処理の一部の開発では問題とならなかったが，より一般的なプログラムの開発・保守には，メモリ読み書きに関する副作用の検出支援が必須である．メモリ読み書きに関する副作用は，ある書き込みノードがあるメモリ領域に書き込んだ内容を，対応する読み込みノードが読む前に別の書き込みノードが上書きすることによって起こる．従って，その検出は，読み書き対象となっているメモリ領域が重複している可能性を記号実行結果を用いて警告すること，および，あるメモリ領域に対する読み書きシーケンスをシーケンス図によって明示することで，少なくともある程度支援可能と考えられる．

さらには，本プロトタイピング手法を礎にして，プログラム割当・命令セットアーキテクチャ・パイプライン構成を応用に応じて最適設計する手法を定式化することが挙げられる．例えば，チップ面積の制約を無視すれば，インライン展開された実行可能プログラムのデータ依存性を，そのまま回路として実装するのが最も高い性能を達成できる．しかし一般にはチップ面積の制約から不可能なので，データ依存性をプロセッサさらにはチップとして畳み込んでいく手法の確立が必要である．そのためには，命令プリミティブ水準に止まらず，命令プリミティブを実現する加算器やシフタ等，レジスタトランスファーレベルまで扱えるように，仕様記述の範囲を拡張する必要があると考えられる．このような最適設計手法を確立すれば，RESCUE を，仕様記述に基づいてソフトウェア・ハードウェア両面の最適設計をガイドする環境へと発展できる可能性がある．

また，CUE プロジェクトでは現在，ノイマン型プロセッサとデータ駆動プロセッサの協調動作によって，全体として実時間多重処理を効率よく実現するプロセッサアーキテクチャが検討されている．これは，逐次型のアルゴリズムはノイマン型プロセッサで実行し，並列型のアルゴリズムはデータ駆動プロセッサで実行するもの

である．先に述べたように，DDS からノイマン型プログラムを生成するためには，データ依存性をレジスタないしメモリを介したデータ受け渡し処理に変換し，制御依存性を付加すればよい．この協調動作時の性能評価・検証手法を定式化し実装することによって，RESCUE は将来の CUE プロセッサの開発にも有効となると考えられる．

従って，本論文に述べた，仕様記述から実行可能プログラム，パイプライン構成に至るまでの一貫したデータ駆動パラダイムに基づく開発支援手法は，次世代の情報インフラストラクチャに必須と考えられる実時間多重処理システムを実現する上で，今後なお一層の発展が期待される．

謝辞

本研究の全過程を通じて, 終始適確かつ熱心な御指導・御鞭撻を賜った筑波大学電子・情報工学系教授 西川博昭先生には, 心から深く感謝の意を表するとともに，厚く御礼を申し上げる．

本論文の審査にあたり，懇篤なる御指導を頂くとともに種々の御高配を賜った，筑波大学 機能工学系 教授 白川友紀先生，電子・情報工学系教授 田中二郎先生，同助教授 安永守利先生，ならびに，同助教授 朴泰祐先生に深謝の意を表する．

データ駆動型プロセッサのアーキテクチャやシミュレータ/エミュレータ環境に関して多大な御支援を頂いたシャープ株式会社 宮田宗一博士，木原誠一郎氏，紫竹リカルド毅史氏ならびに芳田眞一氏に心より御礼申し上げる．

ネットワーキングアーキテクチャに関して多大な御教示と御助言を頂いた，NTT 情報流通プラットフォーム研究所 石井啓之博士に心から感謝する．

本論文の作成にあたり適切な御助言と御示唆を頂いた，筑波大学電子・情報工学系講師 冨安洋史先生に心から感謝する．

本研究を進めるにあたり多大な御助言と御協力を頂いた工学研究科 5 年次青木一浩氏に深く感謝する．

西川研究室の卒業生である大胐将史氏ならびに工藤龍一氏には，RESCUE の実現にあたってご協力いただいたことを心から感謝する．

工学研究科 4 年次 榑林亮介氏，システム情報工学研究科 2 年次 伊藤伸也氏，同 1 年次 高橋徹氏，安村康平氏には，RESCUE の評価に際してご協力頂いたとともに，常日頃から有益な議論を頂いた．ここに記して感謝する次第である．

参考文献

[1] 村井 純,“次世代インターネット技術,” 電子情報通信学会誌, Vol.84, No.1, pp.2-9, Jan. 2001.

[2] 中島 毅, 別所 雄三, 山中 弘, 広田 和洋, “状態遷移モデルで記述された要求仕様に基づく組込みソフトウェアの自動試験法,” 電子情報通信学会論文誌, Vol. J84-D-I, No.6, pp.682-692, June 2001.

[3] 山尾 泰, 梅田 成視, 大津 徹, 中嶋 信生, “第 4 世代移動通信の展望 — 無線システムを中心とした課題について —,” 電子情報通信学会誌, Vol.J83-B, No.10, pp.1364-1373, Oct. 2000.

[4] 安家 武, 西村 正寿, Andreas Sucahyono, 中西 正洋, 戸出 英樹, 池田 博昌, “スクリーニングサービスを含めた通信サービス仕様エミュレータの構築及び評価,” 電子情報通信学会論文誌, Vol. J82-B, No.5, pp.858-867, May 1999.

[5] Marcel Mampaey, “TINA for Services and Advanced Signaling and Control in Next-Generation Networks,” IEEE Communication Magazine, pp.104-110, Oct. 2000.

[6] 白鳥 則郎, 木下 哲男, 菅沼 拓夫, 菅原 研次, 藤田 茂, “次世代ネットワークソフトウェアの構築に向けて,” 電子情報通信学会論文誌, Vol.82-B, No.5, pp.694-701, May 1999.

[7] 淺谷 耕一, 小関 伸也,“GII (Global Information Infrastructure) に関する標準とその動向,” 電子情報通信学会誌, Vol.82, No.3, pp.222-231, Mar. 1999.

[8] 萩島功一, 小柳 恵一, 北見 憲一, 山口 治男, “インフラ系ネットワークの変革,” 電子情報通信学会誌, Vol.81, No.4, pp.371-379, Apr. 1998.

[9] 岸本 了造, “次世代のマルチメディア通信トランスポートネットワークについて,” 電子情報通信学会論文誌, Vol. J79-B-I, No.5, pp.195-206, May 1996.

[10] 宮崎孝, 黒田一郎, “汎用プロセッサを用いたビデオコーデック,” 電子情報通信学会論文誌, Vol.J83-A, No.12, pp.1339-1348, Dec. 2000.

[11] 鮫島 康則, 宮崎 敏明, 深沢 友雄, 寺元 光生, 松広 一良, “リアルタイムパケットフィルタ,” 電子情報通信学会論文誌, Vol.J83-B, No.10, pp.1419-1427, Oct. 2000.

[12] ウィチャイ ブンクムクラオ, 三木 信弘, “帯域消去フィルタの FPGA による実現,” 電子情報通信学会論文誌, Vol.J83-D2, No.11, pp.2171-2179, Nov. 2000.

[13] 丸山 充, 中野 治, 西村 一敏, “多重度の向上を目指したプロトコル処理技術の提案と評価,” 電子情報通信学会技術報告, IN93-113, pp.45-52, Jan. 1994.

[14] 鈴木 康夫, 荒木 純道, “ソフトウェア無線機とその国内における開発の現状,” 電子情報通信学会論文誌, Vol.J84-B, No.7, pp.1120-1131, July 2001.

[15] Hoon Coi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang, Chong-Min Kyung, “Synthesis of Application Specific Instructions for Embedded DSP Software,” IEEE Transactions on Computers, Vol.48, No.6, pp.603-614, June 1999.

[16] 高橋 真史, “携帯電話用マルチメディア SoC の開発,” 電子情報通信学会誌, Vol.84, No.11, pp.790-795, Nov. 2001.

[17] Margarida F. Jacome, Gustavo de Vaciana, “Design Challenges for New Application-Specific Processors,” IEEE Design & Test of Computers, pp.40-50, April-June 2000.

[18] Margarida F. Jacome, Helvio P. Peixoto, “A Survey of Digital Design Reuse,” IEEE Design & Test of Computers, Vol.18, No.3, pp.98-107, 2001.

[19] Sujit Dey, Clark N. Taylor, Debashis Panigrahi, Krishna Sekar, Li Chen, Pablo Sanchez, “Using a Soft Core in a SoC Design: Experiences with picoJava,” IEEE Design & Test of Computers, Vol.17, No.3, pp.60-71, 2000.

[20] Chris Basoglu, Karl Zhao, Keiji Kojima, Atsuo Kawaguchi, “The MAP-CA VLIW-based Media Processor From Equator Technologies Inc and Hitachi Ltd.,” Equator Technologies, Inc.

[21] 中村 浄重, 酒居 敬一, 阿江 忠, “VLIW ハードウェアスタックプロセッサを用いたマルチメディアデータ処理,” 電子情報通信学会論文誌, Vol.J81-D-I, No.1, pp.21-27, Jan. 1998.

[22] Kwok Wah Yeung, “A Data-driven Multiprocessor Architecture for High Throughput Digital Siginal Processing,” Ph.D Thesis, University of California at Berkeley, May 1995.

[23] Jorgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jorn Lind-Nielsen, Kim G. Larsen, Gerd Behrmann, Kare Kristoffersen, Arne Skou, Henrik Leerberg, Niels Bo Theilgaard, “Practical Verification of Embedded Software,” IEEE Computer, Vol.33, No.5, pp.68-75, May 2000.

[24] 木下 真吾, 山下 博之, 村主 俊彦, 永見 康一, “大規模 ASIC の機能検証に重点をおいたハードウェア / ソフトウェア協調シミュレーション,” 電子情報通信学会論文誌, Vol.J81-D-I, No.3, pp. 303-317, Mar 1998.

[25] 河村 一樹, “ソフトウェア工学入門,” 近代科学社, 1995.

[26] Glenford J. Myers, 有澤 誠 訳, “ソフトウェアの信頼性 — ソフトウェア・エンジニアリング概説,” 近代科学社, 1977.

[27] Elaine J. Weyuker, "Testing Component-Based Software : A Cautionary Tale," IEEE Software, Vol.15, No.5, pp.54-59, 1998.

[28] 直井 邦彰, 高橋 直久, "経路依存フローグラフを用いた意味構成管理モデル," 電子情報通信学会論文誌, Vol.J80-D-I, No.11, pp.898-906, Nov. 1997.

[29] Jeffrey Voas, "Can Chaotic Methods Impove Software Quality Predictions?," IEEE Software, Vol.17, No.5, pp.21-22, 2000.

[30] 市川 至, 蓬莱 尚幸, 佐伯 元司, 米崎 直樹, 榎本 肇, "自然言語に基づく静的システムの仕様のプロトタイププログラムへの変換手法," 情報処理学会論文誌, Vol.27, No.11, pp.1112-1128, Nov.1986.

[31] Douglas T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol.SE-3, No.1, pp.16-34, Mar. 1977.

[32] 国井 利泰 監修, 大野 徇郎, 斉藤 信夫, 原田 賢一, 山崎 利治 編, "ソフトウェア工学 — 要求仕様技術 —," bit 臨時増刊, Vol 10, No.10, pp. 216-230, Aug. 1978.

[33] 西松 顯, 西江 圭介, 楠本 真二, 井上 克郎, "フォールト位置特定におけるプログラムスライスの実験的評価," 電子情報通信学会論文誌, Vol.J82-D1, No.11, pp.1336-1344, Nov. 1999.

[34] 高田 智規, 佐藤 慎一, 井上 克郎, "プログラム依存グラフの効率的な更新手法," 電子情報通信学会論文誌, Vol.J81-D-I, No.3, pp.253-260, Mar 1998.

[35] 藤井 論, 千葉 雅弘, 高柳 雄一, 中村 智法, "ソフトウェア開発支援システム SDSS における CASE 統合化," 情報処理学会論文誌, Vol.36, No.1, pp.84-92, Jan. 1995.

[36] 勝山 光太郎, 佐藤 文明, "設計仕様記述用言語 TTCN の特質と処理系の現状と動向," 情報処理, Vol.31, No.1, pp.65-74, Jan. 1990.

[37] Bertram Weber, "Automating PBX System Testing," IEEE Design & Test of Computers, Vol.16, No.3, pp.44-52, 1999.

[38] 辻 宏郷, 佐藤 文明, 勝山 光太郎, 水野 忠則, 曽我 正和, "形式手法による通信ソフトウェア試験データの生成とその試験法," 情報処理学会論文誌, Vol.34, No.6, pp.1347-1360, June 1993.

[39] 塩見 彰睦, 竹田 尚彦, 河合 和久, 大岩 元, "HCP チャートエディタ PAN/HCP," 情報処理学会論文誌, Vol.33, No.02, pp.183-194, Feb. 1992.

[40] Frank Slomka, Matthias Dorfel, Ralf Munzenberger, Richard Holfmann, "Hardware/Software Codesign and Rapid Prototyping of Embedded Systems," IEEE Design & Test of Computers, Vol.17, No.2, pp.28-38, 2000.

[41] Jon Postel, "User Datagram Protocol," RFC 768, Information Sciences Institute, 1980.

[42] Jon Postel, "Internet Protocol," RFC 791, Information Sciences Institute, 1981.

[43] K. Keutzer, S. Malik, R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," IEEE Transactions on Computer-Aided Design of Circuits and Systems, Vol. 19, No. 12, Dec. 2000.

[44] Jack B. Dennis, "Dataflow Schemas," Project MAC, pp.187-216, M.I.T., July 1972.

[45] Lester J. Holtzblatt, Richard L. Piazza, Howard B. Reubenstein, Susan N. Roberts, David R. Harris, "Design Recovery for Distributed Systems," IEEE Transactions on Software Engineering, Vol.23, No.7, pp. 461-472, July 1997.

[46] 門田 暁人, 松本 健一, 飯田 元, 井上 克郎, 鳥居 宏次, "Java クラスファイルに対する電子透かし法," 情報処理学会論文誌, Vol.41, No.11, pp.3001-3009, Nov.2000.

[47] 横山 孝典, 納谷 英光, 成沢 文雄, 倉垣 智, 永浦 渉, 今井 崇明, 鈴木 昭二, “組込み制御システムのための時間駆動オブジェクト指向ソフトウェア開発法,” 電子情報通信学会論文誌, Vol.J84-D-I, No.4, pp.338-349, Apr. 2001.

[48] 柴合 治, 岩元 莞二, 藤林 信也, “統一的設計方法論に基づくソフトウェア設計システム,” 情報処理, Vol.21 No.5, 528-538, May 1980.

[49] 岡敦子, 山本修一郎, 磯田定宏 “ソフトウェア開発実験に基づく構造化分析／設計手法の評価,” 情報処理学会論文誌, Vol.34, No.12, pp.2543-2551, Dec. 1993.

[50] 江崎 和博, 山田 茂, 高橋 宗雄, “設計レビューにおけるソフトウェア信頼性に影響を及ぼす人的要因の品質工学的解析,” 電子情報通信学会論文誌, Vol. J84-A, No.2, pp.218-228, Feb. 2001.

[51] 川口 進, 白須 宏俊, 奥田 哲, “階層化データフロー図によるプログラムの設計と実働,” 電子情報通信学会論文誌, Vol. J82-D-I, No.10, pp.1265-1275, Oct. 1999.

[52] Hiroaki Nishikawa and Souichi Miyata, “Design Philosophy of Super-Integrated Data-Driven Processors: CUE,” Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 415-422, July 1998.

[53] Jon Postel, “Transmission Control Protocol,” RFC 793, Information Sciences Institute, 1981.

[54] Object Management Group, “The Common Object Request Broker : Architecture and Specification,” Revision 2.6, Dec. 2001.

[55] Hiroaki Nishikawa, and Kazuhiro Aoki, “Data-Driven Implementation of Protocol Handling,” Proceedings of 1998 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 430-437, July 1998.

[56] 西川博昭, 青木一浩, “プロトコル多重処理のデータ駆動型実現法とその実験的検討,” 電子情報通信学会論文誌 D-I, Vol.J85-D-I, No.7 （2002 年 7 月掲載予定）

[57] 西川 博昭, 寺田 浩詔, 浅田 勝彦, “履歴依存性を許すデータ駆動図式,” 電子情報通信学会論文誌 (D), Vol.J66-D, No.10, pp.1169-1176, Oct.1983.

[58] Arvind and Rishiyur S. Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture,” IEEE Transactions on Computers, Vol.39, No.3, Mar. 1990.

[59] James McGraw, Stephen Skedsielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas, “SISAL: Streams and Iteration in a Single Assignment Language,” Language Reference Manual Version 1.2. Report M-146, Lawrence Livermore National Laboratory, Livermore, California, 1985.

[60] Arvind, Kim P. Gostelow and Wil Plouffe, “An asynchronous programming language and computing machine,” Technical Report TR-114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[61] Steven R. Vegdahl, “A Survey of Proposed Architectures for the Execution of Funtional Languages,” IEEE Transactions on Computers, Vol. C-33, No.12, Dec. 1984.

[62] Jean Luc Gaudiot, ”Structure Handling in Data-Flow Systems,” IEEE Transactions on Computer, Vol.C-35, No.6, 1986.

[63] 佐藤 三久, 児玉 祐悦, 坂井 修一, 山口 喜教, “並列計算機 EM-4 の並列プログラミング言語 EM-C,” 情報処理学会論文誌, Vol.35, No.4, pp.551-560, Apr. 1994.

[64] 林 健志, 五十嵐 滋, “プログラムの検証理論,” 情報処理, Vol.17, No.5, pp.437-447, May 1976.

[65] 玉井 哲雄, 福永 光一, “記号実行システム,” 情報処理, Vol.23, No.1, pp.18-28, Jan. 1982.

[66] 西川 博昭, 我孫子 泰祐, “タグ操作を許すデータ駆動プログラムの開発・再利用支援手法,” 電子情報通信学会論文誌 D-I, Vol.J85-D-I, No.3 （2002 年 3 月掲載予定）

[67] Hiroaki Nishikawa and Yasuhiro Wabiko, “A Reuse-Oriented Specification Environment Based on Novel Data-Driven Paradigm,” Proceedings of the Third World Conference on Integrated Design and Process Technology, Society for Design & Process Science, pp. 313-320, July 1998.

[68] Hiroaki Nishikawa and Yasuhiro Wabiko, “Prototype Data-Driven Specification Environment and Its Evaluations,” Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 446-453, July 1998.

[69] Dolores R. Wallace and Roger U. Fujii., “Software Verification and Validation: An Overview,” IEEE Software, Vol.6, No.5, pp.10-17, May 1989.

[70] James D. Arthur, Markus K. Groner, Kelly J. Hayhurst, C. Michael Holloway, “Evaluating the Effectiveness of Independent Verification and Validation,” IEEE Computer, Vol.32, No.10, pp.79-83, Oct. 1999

[71] Yasuhiro Wabiko, Takuji Urata and Hiroaki Nishikawa, “A Specification and Prototyping Environment for Super-Integrated Data-Driven Processor Systems,” Proceedings of the Fifth World Conference on Integrated Design and Process Technology, Society for Design & Process Science (IDPT 1999), CD-ROM, June 2000.

[72] Yasuhiro Wabiko and Hiroaki Nishikawa, “Verification and Emulation Facility for Data-Driven Realtime Processing Systems,” Proceedings of the Fifth World Conference on Integrated Design and Process Technology, Society for Design & Process Science (IDPT 2000), CD-ROM, June 2000.

[73] Masashi Ohtsuki, Yasuhiro Wabiko and Hiroaki Nishikawa, "Real-time Execution System for CUE series Data-Driven Processors; RESCUE ," Proceedings of the 2000 Int'l Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1969-1975, June 2000.

[74] Ryuichi Kudo, Yasuhiro Wabiko and Hiroaki Nishikawa, "Performance Verification Scheme for Data-Driven Real-time Processing," Proceedings of the 2000 Int'l Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1977-1983, June 2000.

[75] Yasuhiro Wabiko and Hiroaki Nishikawa, "A Data-Driven Paradigm to Develop and Tune Data-Driven Realtime System," Proceedings of the 2001 Int'l Conference on Parallel and Distributed Processing Techniques and Applications, pp. 350-356, June 2001.

[76] George A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," Psychological Review, Vol.63, pp.81-97, Mar. 1956.

[77] Richard M. Shiffrin and Robert M. Nosofsky, "Seven Plus or Minus Two: A Commentary On Capacity Limitations," Psychological Review, Vol.101, No.2, pp.357-361, 1994.

[78] 浜田 雅樹, 安達 久人, "設計プロセス情報を利用したソフトウェア修正支援方式," 情報処理学会論文誌, Vol.35, No.05, pp.887-896, May 1994.

[79] 鈴木 英明, 高橋 直久, "プロセス分解代数に基づくデータフロー図の段階的詳細化," 情報処理学会論文誌, Vol.34, No.3, pp.511-522, Mar.1993.

[80] 古川 忠始, 本位田 真一, 大須賀昭彦, 津田淳一郎, "代数的仕様記述と図式仕様記述の相補的役割について — 複眼的システムモデル —," 情報処理学会論文誌, Vol.31, No.2, pp.182-193, Feb. 1990.

[81] 河井 淳, 西山 由高, “組み込みシステム用ソフトウェア開発環境,” 情報処理, Vol.37, No.9, pp.872-879, Sep. 1996.

[82] 大原 茂之, “木構造化チャートによるプログラム開発・保守技法,” 情報処理学会論文誌, Vol.27 No.10, pp. 1019-1026, Oct. 1986.

[83] 所 真理雄, 松岡 聡, 垂水 浩幸, “オブジェクト指向コンピューティング,” 岩波書店, Nov. 1993.

[84] Rajeev Alur, Thomas A. Henzinfer, Per-Hsin Ho, “Automatic Symbolic Verification of Embedded Systems,” IEEE Transactions of Software Engineering, Vol. 22, No. 3, Mar. 1996.

[85] Patrick Cousot, “Abstract Interpretation Based Formal Methods and Future Challenges,” Informatics. 10 Years Back. 10 Years Ahead, Lecture Notes in Computer Science 2000, Springer-Verlag Berlin Heidelberg, pp.138-156, 2001.

[86] 吉岡信和, 鈴木正人, 片山卓也, “抽象解釈に基づく仕様の段階的具体化法,” 情報処理学会研究報告「ソフトウェア工学」, Vol.95, No.25, pp.137-144, Mar. 1995.

[87] John McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I,” Communications of the ACM, pp.184-195, Apr. 1960.

[88] 阿部 睦, 嶋野 淳子, 渥美 幸雄, “端末におけるサービス品質保証方式の提案と評価,” 電子情報通信学会論文誌, Vol.J82-B, No.5, pp.711-721, May 1999.

[89] Andreas Vogel, Brigitte Kerherve, Gregor von Bochmann, Jan Gecsei, “Distributed Multimedia and QOS: A Survey,” IEEE Multimedia, Vol.2, No.2, pp. 10-19, Summer 1995.

[90] Jurg Bolliger, Thomas Gross, “A Framework-Based Approach to the Development of Network-Aware Applications,” IEEE Transactions on Software Engineering, Vol.24, No.5, pp.376-390, May 1998.

[91] 阪田 史朗,“マルチメディア通信プロトコル標準化動向,” 電子情報通信学会誌, Vol.80, No.10, pp.1036-1042, Oct. 1997.

[92] 浦田 卓治, 樽林 亮介, 西川 博昭, “オンチップマルチプロセッサ型データ駆動アーキテクチャの評価手法とその実験的検討,” 情報処理学会論文誌, Vol.42, No. SIG 9 (HPS 3), pp. 135-144, Aug. 2001.

[93] ITU, “40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM),” ITU Recommendation G.726, 1990.

付録 A

RESCUE のシステム構成

A.1　RESCUE の外部システム構成

RESCUE の外部システム構成を図 A.1 に示す．ユーザは，データ構造，データ流量，データ依存性，ターンアラウンドタイム，プリミティブ割当を，プロセッサ割当，ならびに，パイプライン構成を仕様として仕様記述環境に与える．

ユーザインタフェースとしては，2 次元の図的な表現を記述するのに現状で最も適しているデバイスであるマウスを採用し，また，仕様上のブロック名などの文字列を記述するのに現状で最適なデバイスであるキーボードを採用した．

仕様記述環境は，ユーザから与えられた情報を基にして入力ストリーム，ストリーム制御情報，パイプライン構成の情報を生成し，CUE プロセッサエミュレータに送信する．エミュレータはパイプライン上でパケットがステージ間を移動したイベントを記録しこれをトレース情報として仕様記述環境に送信する．

エミュレーションでなく CUE プロセッサ上でプログラムを直接実行する場合は以下のようである．仕様記述環境がアセンブラソースコードを生成する．これをアセンブラがローダブルオブジェクトへとアセンブルする．ローダブルオブジェクトが CUE プロセッサに送り込まれ，実行される．

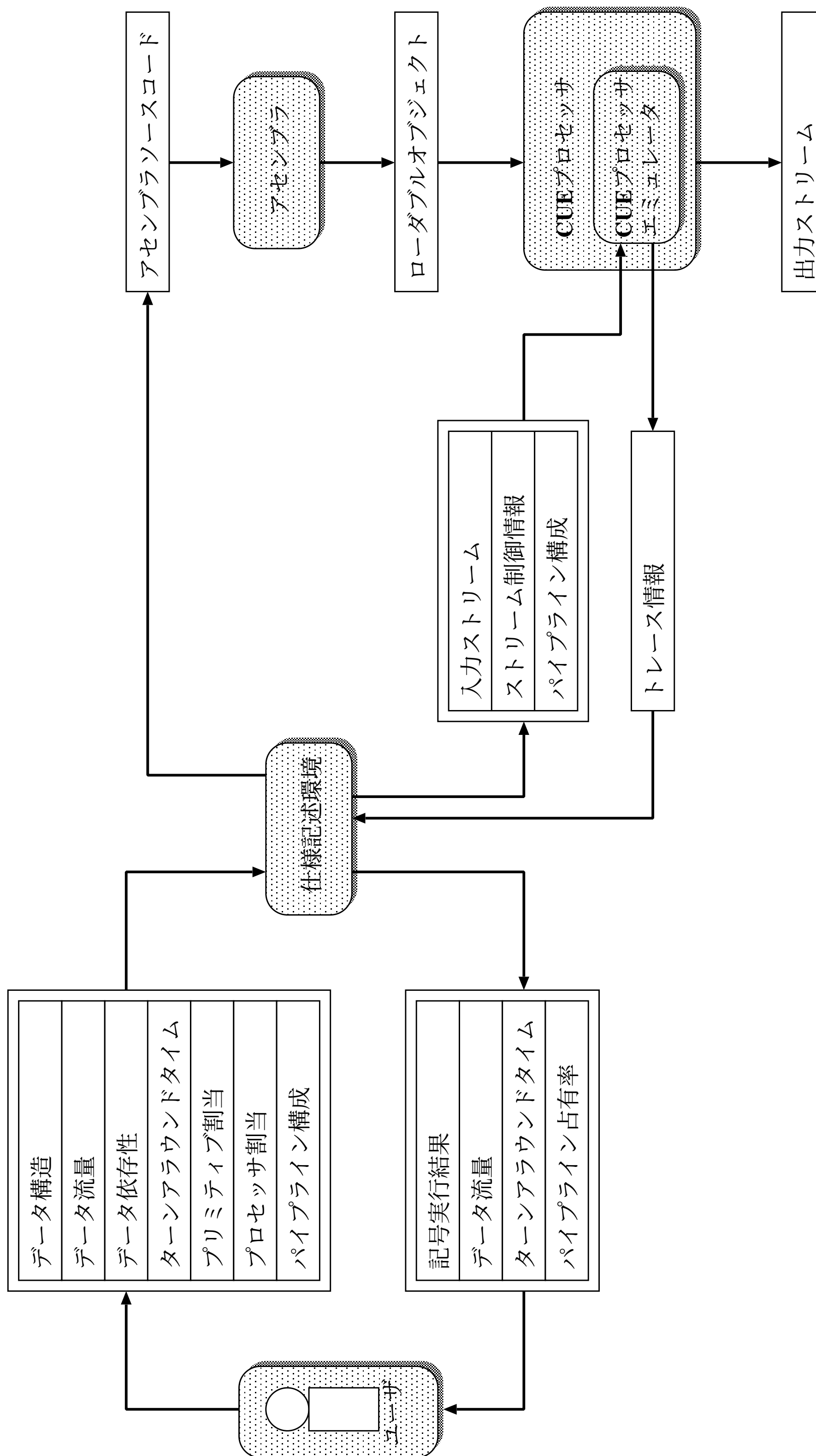


図 A.1: RESCUE の外部システム構成

A.2 RESCUE の内部システム構成

RESCUE の実現に利用したツールおよびライブラリの一覧を表 A.1に示す.

表 A.1: RESCUE の実現に利用したツール・ライブラリ

種類	名称	バージョン
インタプリタ	JPerl	5.005-03
GUI ツールキット	perl/Tk	800.022-jp
オペレーティングシステム	Linux	2.4.2
v ウィンドウシステム	XFree86	3.3.3.1-jp
日本語入力フロントエンド	kinput2	3.0β2
かな漢字変換サーバ	Canna	3.2

付録 B

動画圧縮アプリケーションの仕様記述例とプログラム

B.1 RGB 形式から YCrCb 形式への変換処理の仕様記述例

RESCUE による仕様記述の一例として，動画圧縮アプリケーションにおける RGB（赤，緑，青）から YCrCb（輝度信号，色差信号）への変換処理を示す．図 B.1は，その処理の最上位ブロックである．R,G,B 形式から Y,Cr,Cb 形式に変換する．Y,Cr,Cb の個々の生成処理は，それらの間でのデータ依存性がないために，並列処理可能である．

図 B.1のブロック中の左側の 3 つのノードは，図 B.2に示すような，信号が格納されたトークンを 3 つに複製する処理である．この処理では，Y,Cr,Cb の生成処理を並列に行うため，R,G,B の各パケットをそれぞれ 3 つずつ複製する．

他方，図 B.1のブロック中の右側の 3 つのノードは，図 B.3に示すような，輝度信号・色差信号の生成処理である．この処理では，入力された R,G,B データをもとに R→Y, G→Y, B→Y テーブルを参照し, それぞれを足し合わせることによって Y を生成する．Cr, Cb についても同様に，テーブルを参照して生成する．

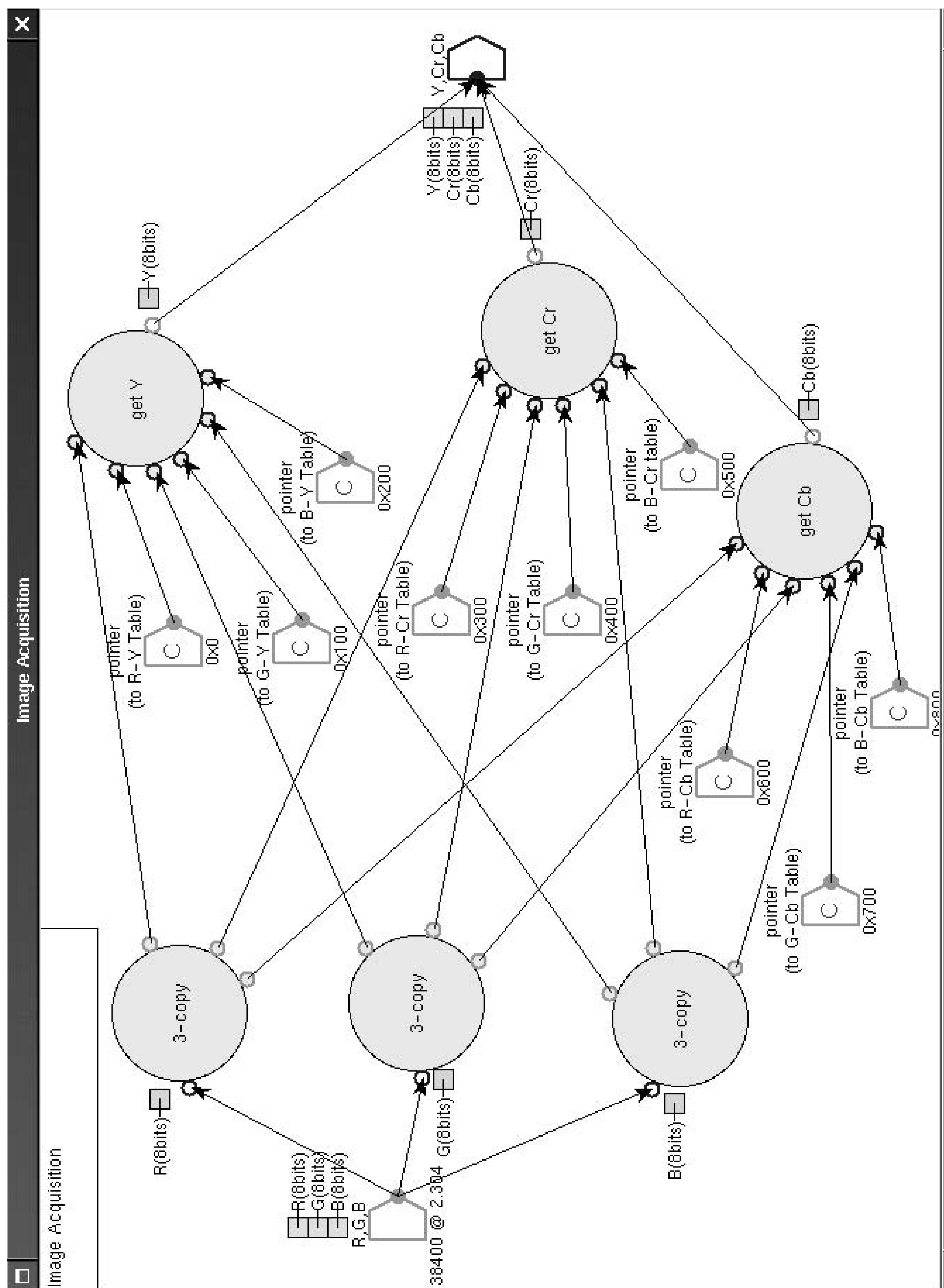


図 B.1: RGB 形式から YCrCb 形式への変換処理 (1) 全体像

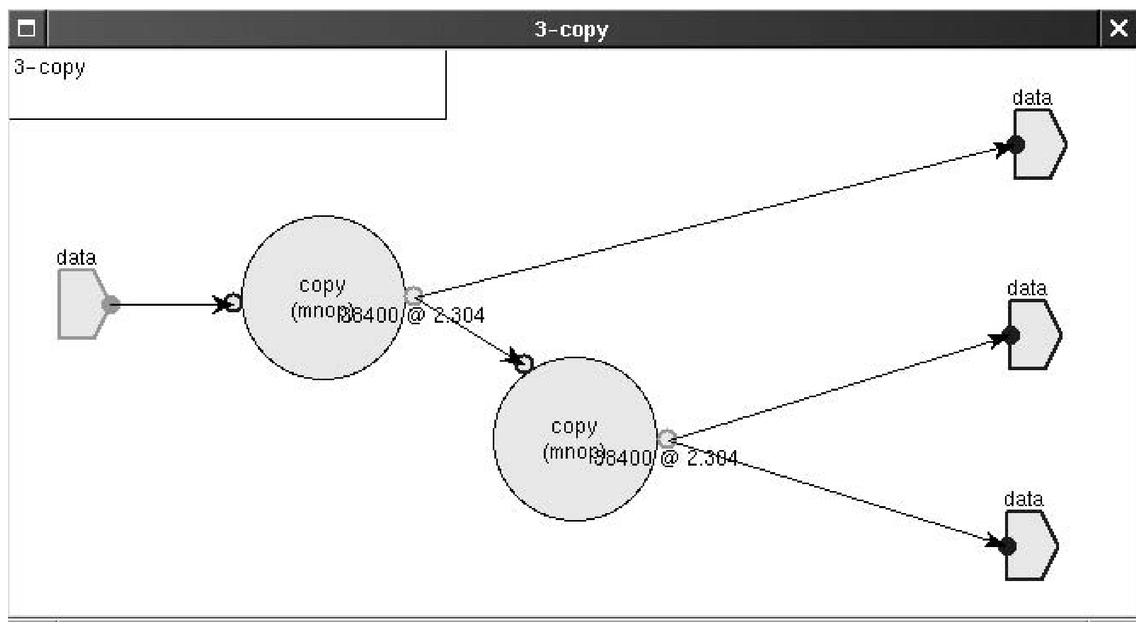


図 B.2: RGB 形式から YCrCb 形式への変換処理 (2) パケット複製処理

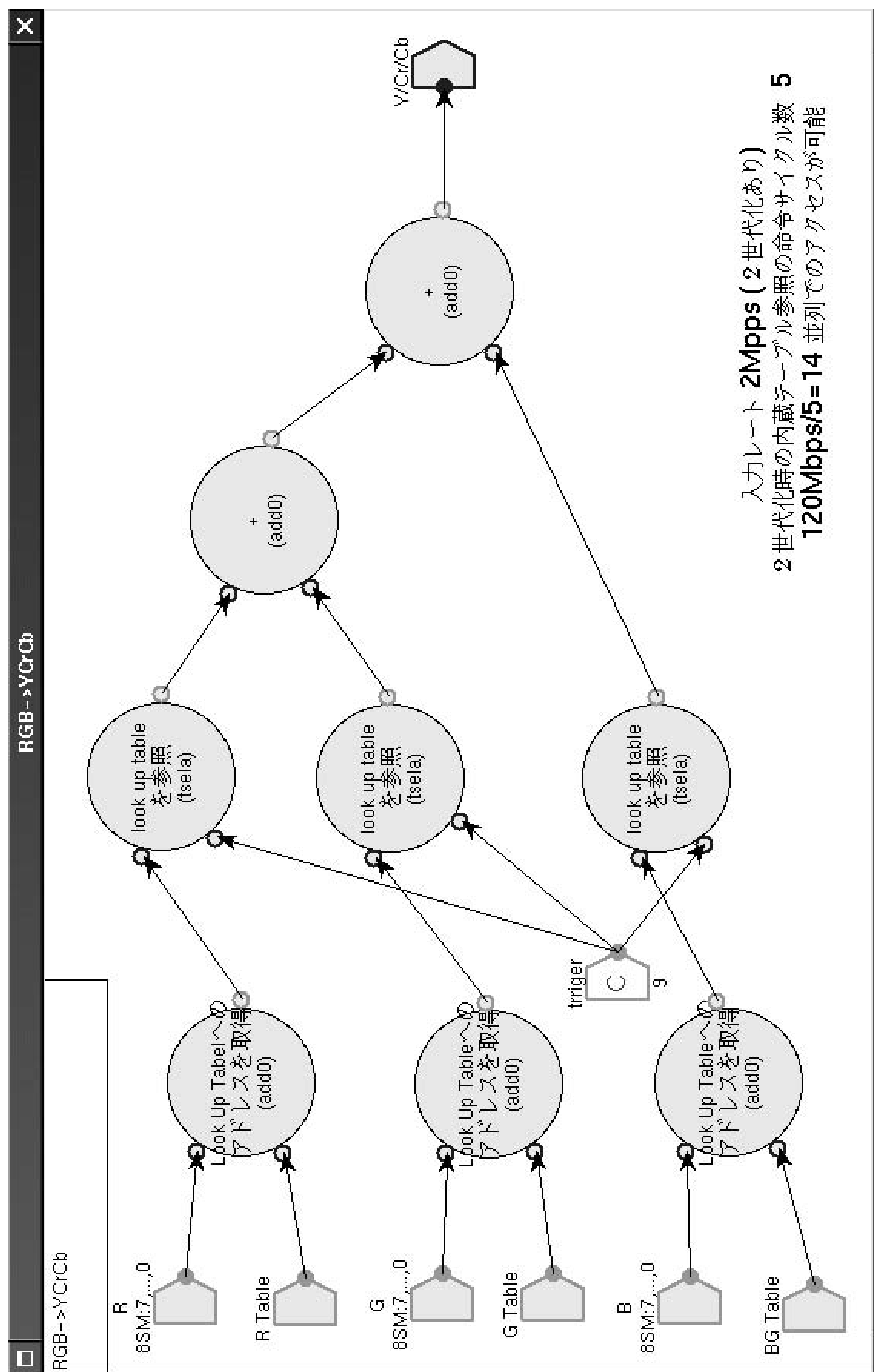


図 B.3: RGB 形式から YCrCb 形式への変換処理 (3)YCrCb 生成処理

B.2　生成されたプログラムのアセンブラソースコード

RESCUE の出力として得られる，アセンブラソースコードは以下のようである．

プログラム B.4　(ia.fs)

```
 1 :
 2 :
 3 :     ocp4g   32
 4 : input         g32_8
 5 : input         g32_0
 6 : input         g32_1
 7 : input         g32_5
 8 : input         g32_7
 9 : input         g32_2
10 : input         g32_2*
11 : input         g32_4
12 : input         g32_4*
13 : input         g32_6
14 : input         g32_6*
15 :
16 : g32_2:  add0    out(33,3)
17 : g32_4:  add0    out(33,5)
18 : g32_6:  add0    out(33,7)
19 :
20 : g32_0:  mnop    out(33,0),g32_3
21 : g32_1:  mnop    out(33,1),out(33,2)
22 : g32_3:  add0(0x100)*    out(33,4)
23 : g32_5:  add0(0x400)*    out(33,6)
```

```
24 : g32_7:  add0(0x700)*    out(33,8)
25 : g32_8:  mnop    out(33,9),out(33,10)
26 :
27 :     ocp4g   33
28 : input        g33_0
29 : input        g33_1
30 : input        g33_5
31 : input        g33_3
32 : input        g33_7*
33 : input        g33_9
34 : input        g33_13*
35 : input        g33_15
36 : input        g33_19*
37 : input        g33_20
38 : input        g33_4
39 :
40 : g33_3:  add0    out(HOST,0)
41 : g33_9:  add0    out(HOST,0)
42 : g33_15: add0    out(HOST,0)
43 :
44 : g33_0:  mnop    out(32,3),out(32,4)
45 : g33_1:  mnop    g33_11,g33_17
46 : g33_2:  tsela(9)    g33_3*
47 : g33_4:  add0(0x0)*  g33_6*
48 : g33_5:  add0(0x200)*    g33_2*
49 : g33_6:  tsela(9)    out(32,5)
50 : g33_7:  tsela(9)    out(32,6)
51 : g33_8:  tsela(9)    g33_9*
```

```
52 : g33_10: add0(0x300)*    g33_12*
53 : g33_11: add0(0x500)*    g33_8*
54 : g33_12: tsela(9)    out(32,7)
55 : g33_13: tsela(9)    out(32,8)
56 : g33_14: tsela(9)    g33_15*
57 : g33_16: add0(0x600)*    g33_18*
58 : g33_17: add0(0x800)*    g33_14*
59 : g33_18: tsela(9)    out(32,9)
60 : g33_19: tsela(9)    out(32,10)
61 : g33_20: mnop    g33_10,g33_16
```

B.3 RGB 形式から YCrCb 形式への変換処理のターンアラウンドタイム・データ流量の性能予測例

ターンアラウンドタイム・データ流量の予測結果を図 B.5〜B.7 に示す.

図 B.5では，RGB 形式から YCrCb 形式への計算処理全体として，太いアークで示されているパスがクリティカルパスであり，ターンアラウンドタイムは通信時間を 0 とした最短の場合で 331 nsec, 通信時間を加味した場合で 1841nsec となることが予測されている.

図 B.6では，最短ターンアラウンドタイムが 100nsec，実際のターンアラウンドタイムが 717nsec であることが予測されている．また，この処理はパケットを複製する処理であるため，3 つのシンクにそれぞれ 6.912 MPacket/sec のデータ流量で 115200 個のパケットが流れ込むことが予測されている.

図 B.7では，最短ターンアラウンドタイムが 281nsec，実際のターンアラウンドタイムが 1365nsec であることが予測されている．また，この処理の出力となるシンクには，6.912 MPacket/sec のデータ流量で 115200 個のパケットが流れ込むことが予測されている.

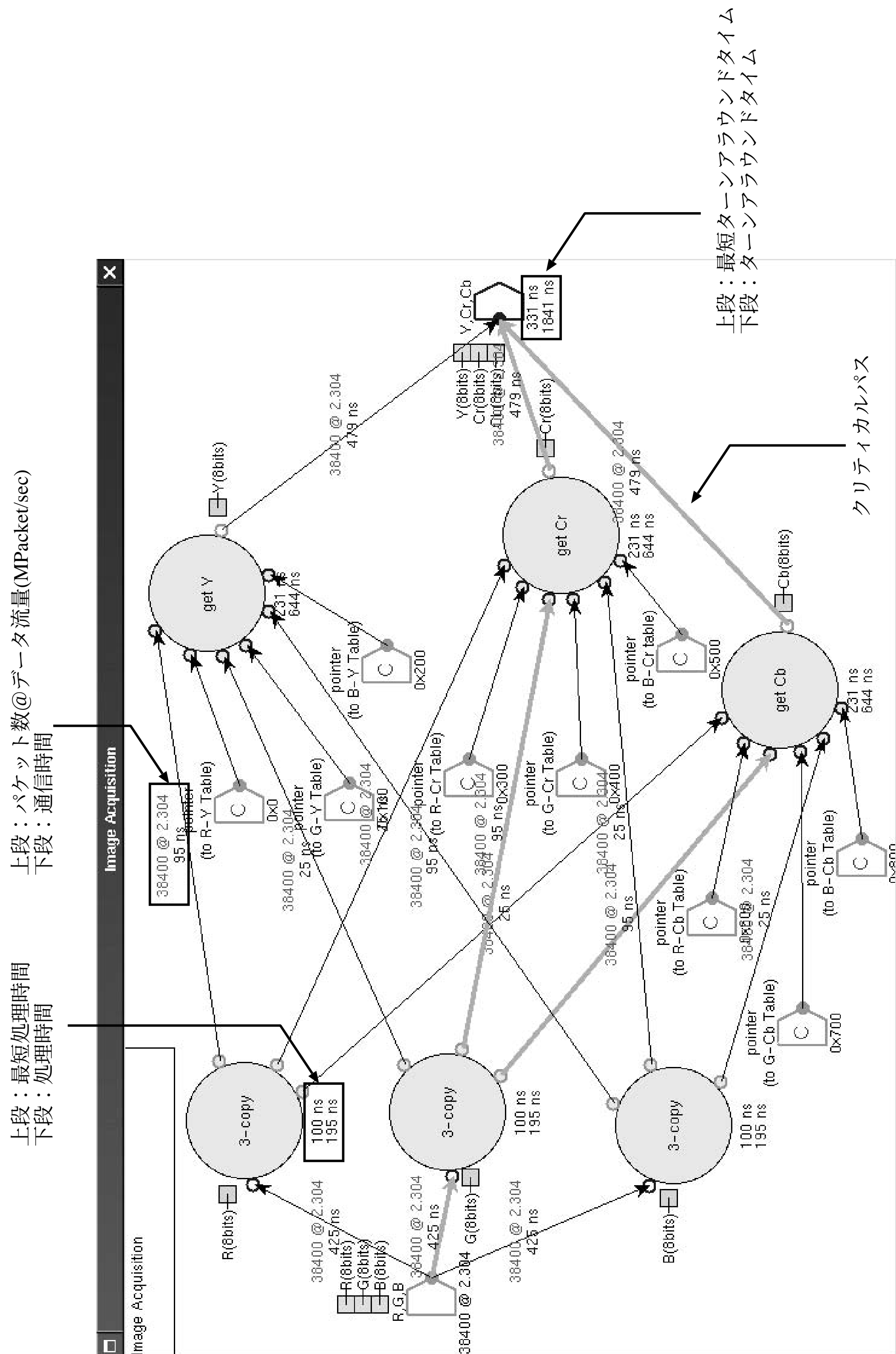


図 B.5: RGB 形式から YCrCb 形式への変換処理の性能予測例 (1) 全体像

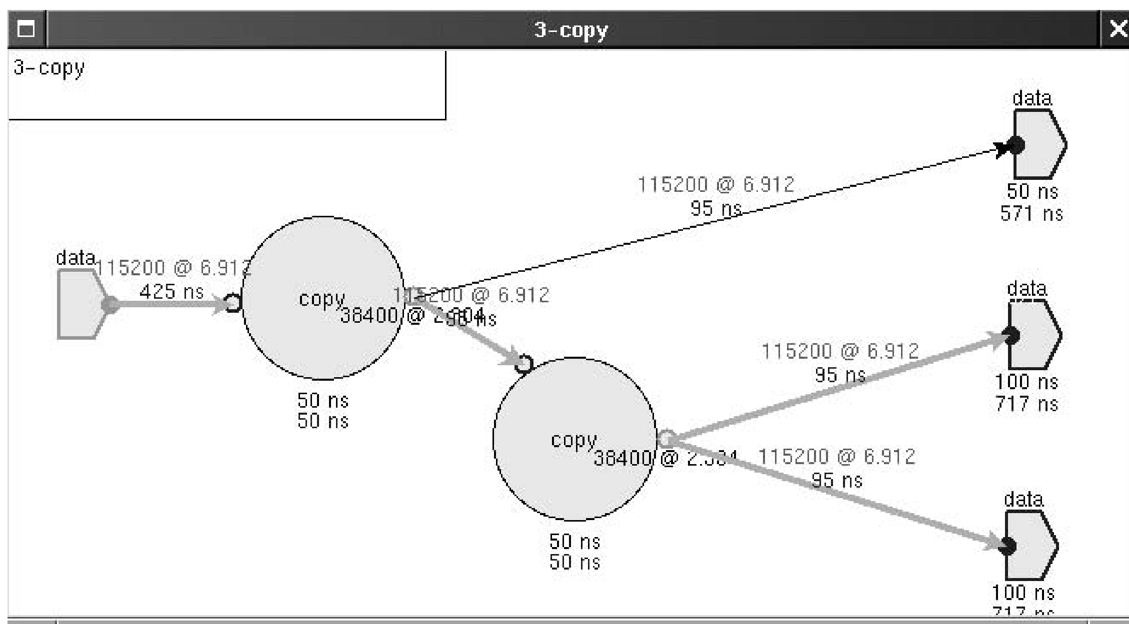


図 B.6: RGB 形式から YCrCb 形式への変換処理の性能予測例 (2) パケット複製処理

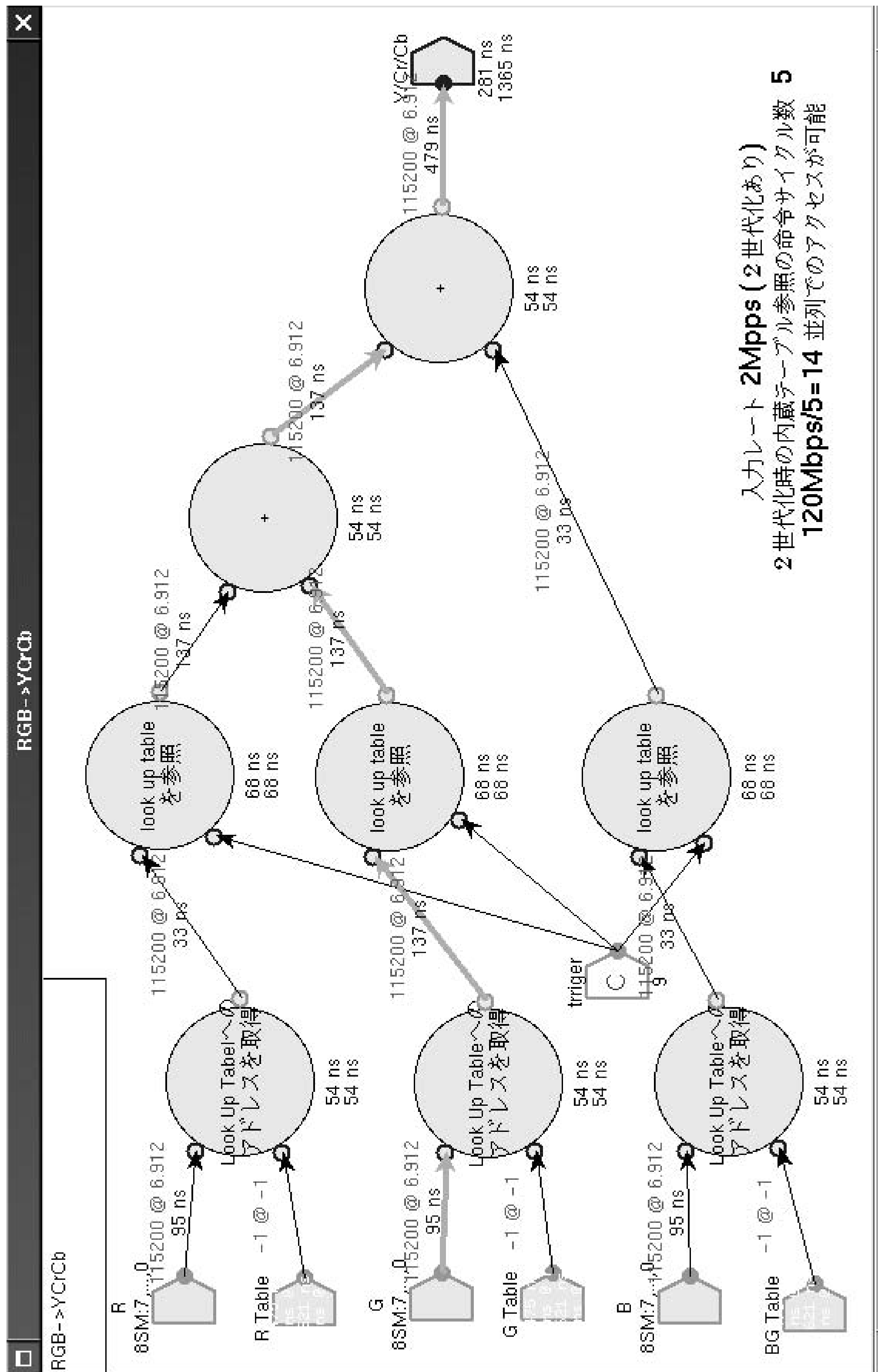
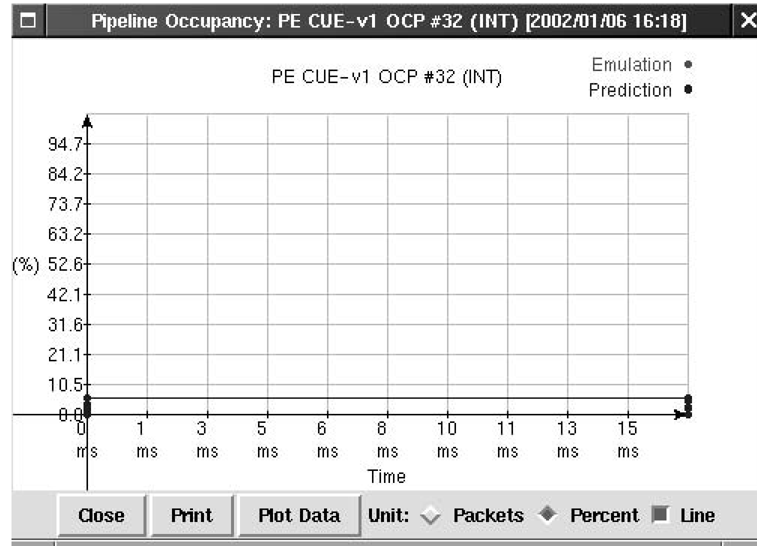


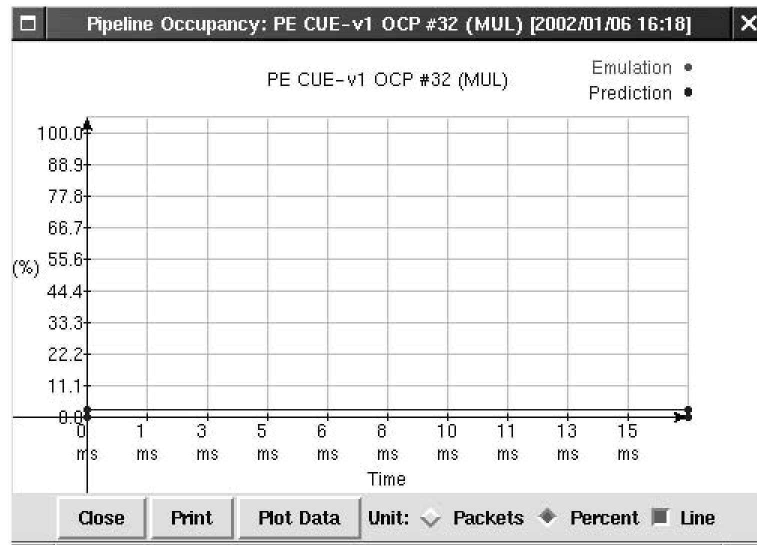
図 B.7: RGB 形式から YCrCb 形式への変換処理の性能予測例 (3)YCrCb 生成処理

B.4 RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例

性能予測結果を図 B.8〜B.10 に示す．どのプロセッサも定常的には高々10%以下のパイプラン占有率となることが予測されている．

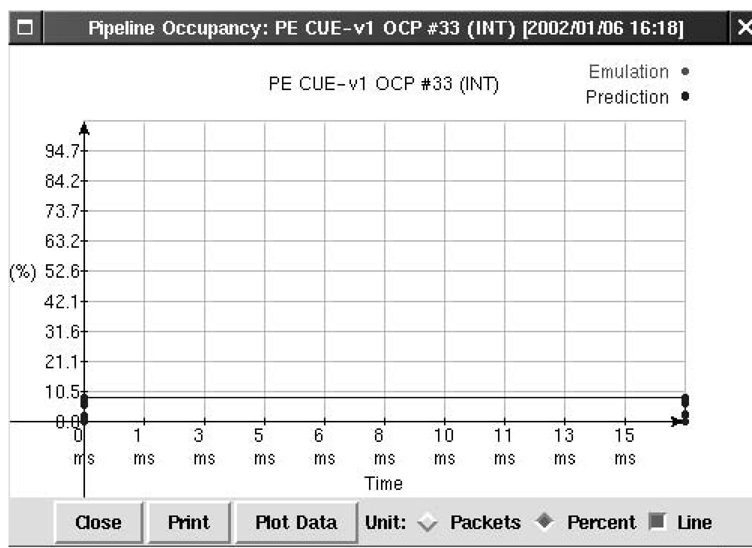


(a) チップOCP32 内の PE INT のパイプライン占有率の予測結果

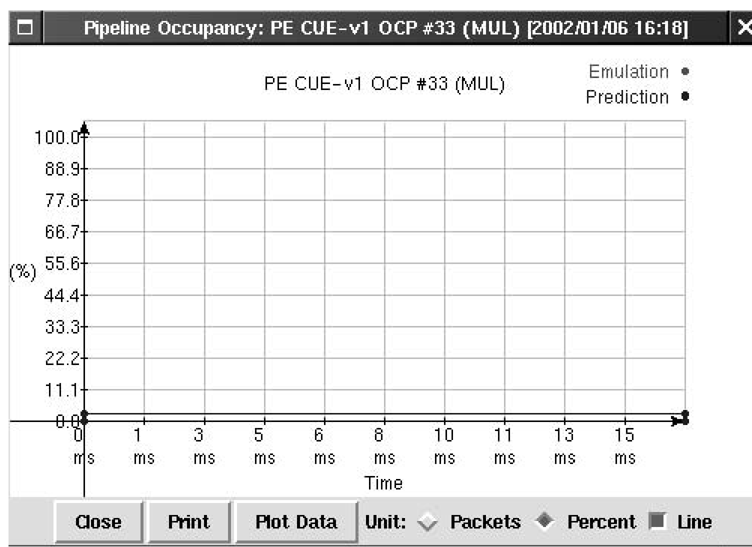


(b) チップOCP32 内の PE MUL のパイプライン占有率の予測結果

図 B.8: RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (1) チップ OCP 32

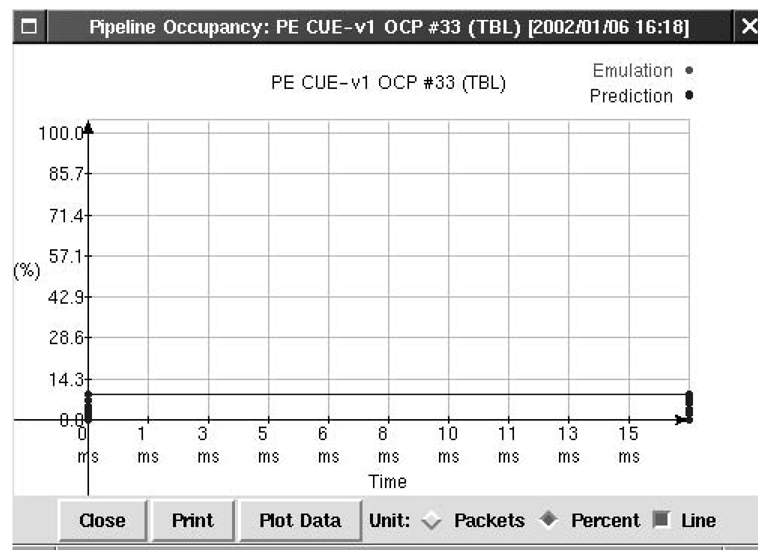


(a) チップOCP33 内の PE INT のパイプライン占有率の予測結果



(b) チップOCP33 内の PE MUL のパイプライン占有率の予測結果

図 B.9: RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (2) チップ OCP 33 の PE INT/MUL



(c) チップOCP33 内の PE TBL のパイプライン占有率の予測結果

図 B.10: RGB 形式から YCrCb 形式への変換処理のパイプライン占有率の予測例 (2) チップ OCP 33 の PE TBL

付録 C

プロトコル処理アプリケーションへの RESCUE の適用例

C.1 TCP/IP プロトコル処理のチェックサム計算の仕様記述例

RESCUE を用いて，TCP/IP プロトコル処理の中のチェックサム計算部分の性能予測ならびにエミュレーションを行った．当該部分の仕様を図 C.1に示す．この仕様には，データ長 512Byte のパケットを OC-9 相当の 408Mb/s で CUE-v1 に入力することが要求として記述されている．仕様上のノードはすべて命令プリミティブであり，3 つの PE に分散して割り当てられている．

C.2 ボトルネックとなっている PE の検出とボトルネックの解消例

TCP/IP プロトコル処理のチェックサム計算処理の性能予測・エミュレーションの結果として得られたパイプライン占有率のグラフを図 C.2,C.3に示す．この例では，図 C.2(a) の GNT とよばれる PE において，予測結果からは実行可能なことが

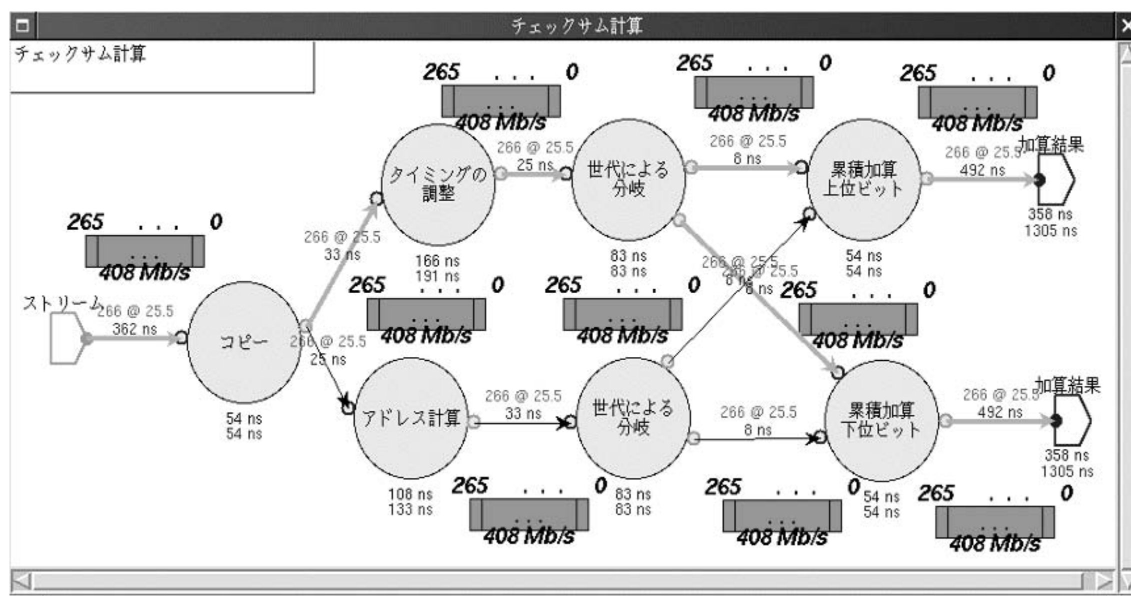
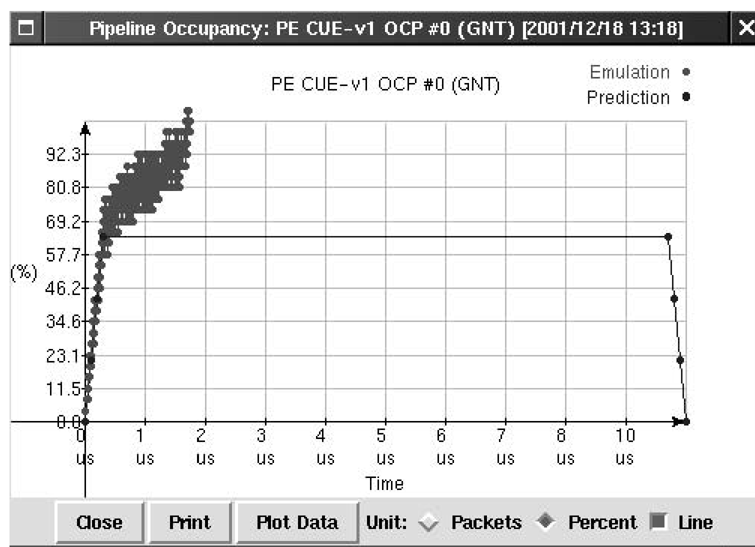
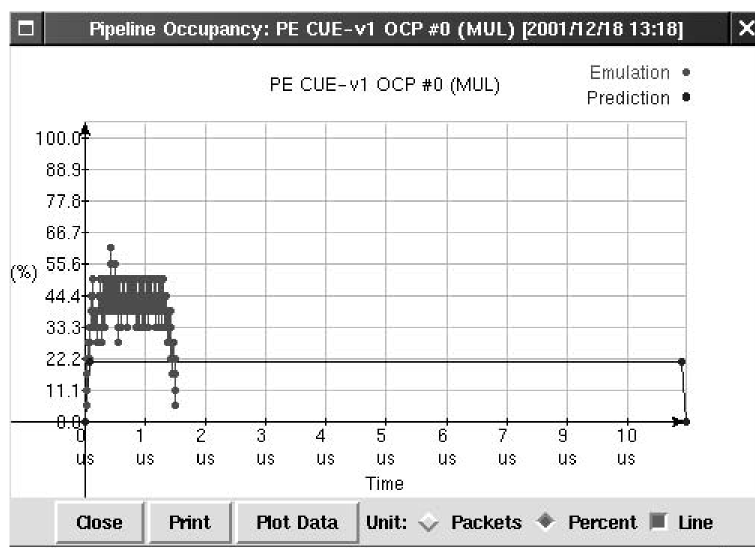


図 C.1: チェックサム計算処理の性能予測・検証の適用例： 仕様記述

示されている一方で，エミュレーション結果からは，負荷が集中した結果オーバーフローを招き，実行が停止することが示されている．そのため，当該 GNT に割当てられていたノードの命令プリミティブを，INT とよばれる PE の命令セットに追加した状況を想定し，再度性能評価・エミュレーションを実施した．その結果を図 C.4,C.5に示す．この改善の結果，すべての PE がオーバーフローすることなく実行されることが示された．

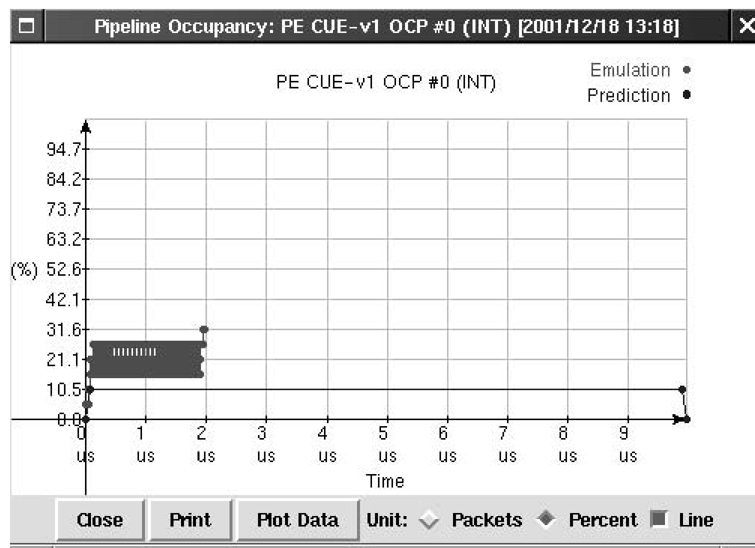


(a) GNT のパイプライン占有率の予測・エミュレーション結果



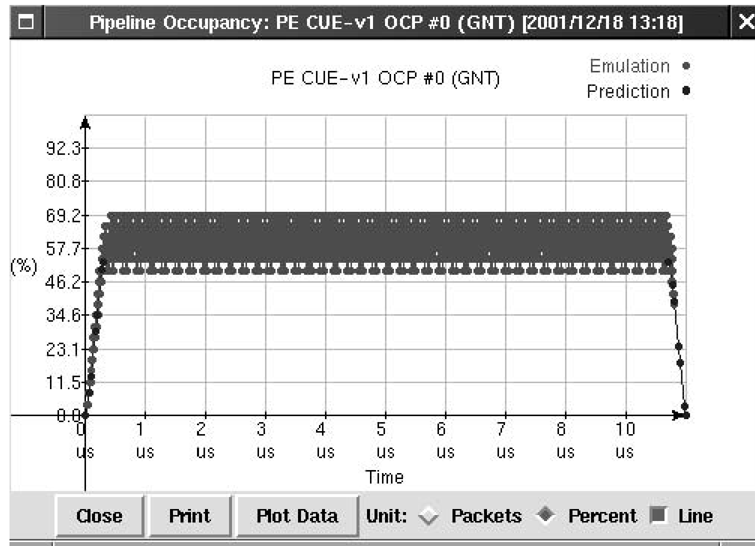
(b) MUL のパイプライン占有率の予測・エミュレーション結果

図 C.2: チェックサム計算処理の性能予測・検証の適用例： ボトルネックの検出 (1)

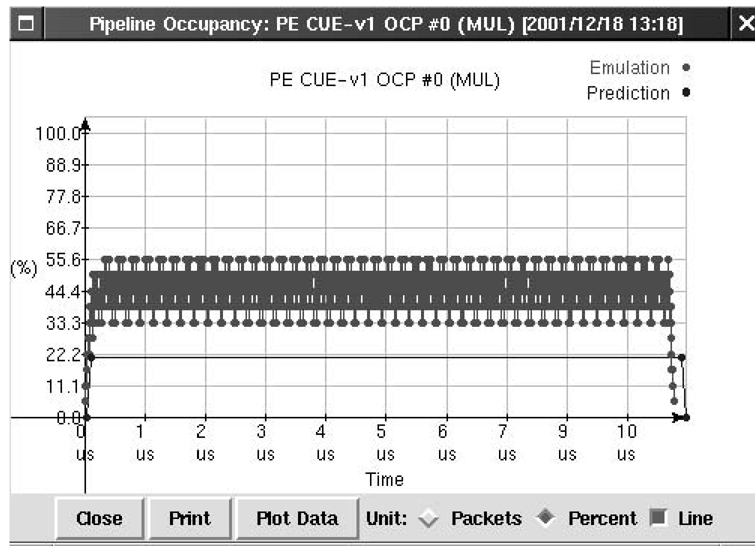


(c) INT のパイプライン占有率の予測・エミュレーション結果

図 C.3: チェックサム計算処理の性能予測・検証の適用例： ボトルネックの検出 (2)

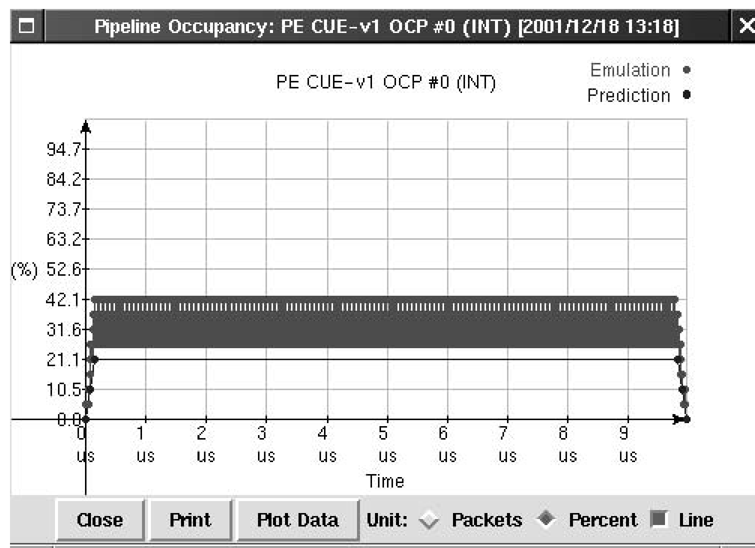


(a) GNT のパイプライン占有率の予測・エミュレーション結果



(b) MUL のパイプライン占有率の予測・エミュレーション結果

図 C.4: チェックサム計算処理の性能予測・検証の適用例： ボトルネックの解消 (1)



(c) INT のパイプライン占有率の予測・エミュレーション結果

図 C.5: チェックサム計算処理の性能予測・検証の適用例： ボトルネックの解消 (2)