シミュレーションによるバス通信構造の設計改善を容易化するバスシステム設計支援ツールの提案

谷本匡亮[†] 北口 智[†,☆]
中田明夫[†] 東野輝夫[†]

システム LSI の性能向上を行ううえで，バス通信構造の高速化は重要な要因の 1 つである．バス通信構造の性能を向上させるためには，シミュレーションなどを介して，適切なバストポロジ，バス調停方式，バスアクセス方式，バスプロトコルなどのバス設計パラメータを決定する必要がある．本論文では，設計者によるバス設計パラメータの設計変更を支援するバスシステム設計支援ツールを提案する．提案ツールでは，設計者は与えられたシステムで利用されるモジュール群の各機能を Java で記述するとともに，支援ツールに用意された典型的なバスプロトコル，バス調停方式のライブラリから，そのシステム用のバス設計パラメータを指定する．それらの指定に従って，対応するバス通信構造のシミュレーションコードが自動生成される．このバス通信構造と設計者が記述した機能モジュール群のシミュレーションコードを組み合わせてシミュレーションを行う．バスの使用率やデッドラインミスの回数など，提案ツールで生成された出力を用いて，バスシステム全体の性能を評価・改善する．具体的なバスシステムの設計例に対し，様々なバス通信構造を想定したシミュレーションコードを自動生成し，性能改善を図った例について報告する．

A CAD Tool for Modeling and Simulation of Bus Systems to Facilitate Refinement of Bus Communication Structure

Tadaaki Tanimoto,[†] Tomo Kitaguchi,[†,☆] Akio Nakata[†]
and Teruo Higashino[†]

In recent years, we can specify various bus systems by system level languages in order to simulate and check whether given real time constraints are satisfied. When such real time constraints are not satisfied, we have to refine various design parameters of the specified bus system such as bus topologies, bus arbitration policies, bus access methods, and bus protocols, to make the real time constraints satisfied. In this paper, we propose a CAD tool to help designers' manual refinement of a bus system description written in Java. The proposed tool generates automatically a bus system simulation code from the bus system design parameters specified by a designer. The bus arbitration policies and bus protocols can be chosen from the predefined library provided by the tool and customized by a designer. By combining the generated bus system simulation code with the manually described simulation codes of functional modules connected with the bus, we can check the number of the real time constraint violations of the total bus system and collect some other information such as bus utilization during the simulation. A case study is also presented.

1. はじめに

近年システム LSI は，携帯電話や携帯情報端末，車載情報システム，情報家電など様々な組み込み機器に応用されている．そのようなシステムの高速化を図るためには，バス通信構造の最適化が重要である．

文献 1)〜3) では自動でバス通信構造の最適化を行う手法が提案されている．文献 1) ではバスプロトコルなどを固定したうえで，コスト最小のネットワークトポロジを合成する手法を提案している．文献 2) では，共通のバスプロトコルの使用を前提に，プロトコルパラメータの自動調整を行おうとしている．文献 3) では，与えられた通信構造に対して，動作中に適応的にプロトコルパラメータを調整する回路を構成する方式を提案している．しかし，これらの手法ではあらかじめバスプロトコルなどが固定されており，必ずしも対象回路に適切なバスプロトコルに最適化されるわけ

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
☆ 現在，松下電器産業株式会社
Presently with Matsushita Electric Industrial, Co., Ltd.

ではない．一方，文献 4)，5) では，手動でのバス通信構造の最適化を支援するツールが提案されている．文献 4) では，トランザクションがシステム部品を流れる経路（ワークフロー）を設計者が与えることにより，様々なバスアーキテクチャによるシステム性能を見積もる手法が提案されている．文献 5) では，様々なバス調停方式を階層的な組合せで表現し，そこからバス上でのデータ転送をシミュレーションするコードを自動生成する方式が提案されている．しかし，いずれの方式もバスにアクセスを行うモジュール群の動作はシミュレーションの対象としないため，与えられたモジュール群の動作に特化した形での最適化には対応していない．SoC や組み込みシステムなどの設計においては，つねに標準的なバスプロトコルやバスアーキテクチャに基づいた設計が行われるわけではなく，むしろ対象システムに適した専用バスの設計を行うことにより，より厳しい性能要求を満たすことが求められている．

一方，近年の SoC の少量多品種化にともない，設計から市場出荷までの時間（time to market）の短縮への要求が年々高まってきている．設計期間の短縮のためには，既存の設計資産（IP: Intellectual Property）を再利用し，できるだけ速く，かつ，既存設計からできるだけ少ない設計変更量で，要求性能を満たす設計を発見することが重要と考えられる．しかし，既存の多くの設計方式や設計ツールは「最適設計の発見」を主目的としており,「要求性能を満たす設計の短時間での発見」を目標としていない．このことを目標とする場合，従来手法のように，設計の大きな部分から細かい部分へ順番に改善を行うのではなく，変更量が少なくかつ性能改善効果が大きい（細かい）部分から変更量が多い部分へ順番に改善を試みることが望ましい．これは，要求性能を満足する設計が見つかれば，たとえその先にもっと良い設計があったとしてもこれ以上の設計改善の必要がないためである．

このため本論文では，自動最適化が困難な抽象度のシステムレベル記述を対象とし，バス調停方式，バスアクセス方式，バスプロトコル，バストポロジなどの設計パラメータの指定からシミュレーション用コードを自動生成し，各項目の設計変更による最適化を支援する設計支援ツールを提案する．提案ツールではバスシステムのシステムレベル記述言語として，典型的なバス調停方式やバスプロトコルに対応するライブラリや，プログラムのどこからどこまでの実行に何クロックサイクルかかるかを指定（以下，**サイクル精度記述**と呼ぶ）できるようなライブラリ関数を用意した Java 言語を採用する☆．各モジュールおよびバス調停回路の並列動作は Java のスレッド機構を用いてシミュレートする．

提案する方式では，まず設計者が動作周波数（バスクロックサイクル），モジュール–バス間の接続情報，バス調停方式，バスプロトコル，各モジュールのレジスタ数，バッファサイズなどからなるシステム構成情報を記述すると，提案する支援ツールにより，システム構成情報に対応するバス構成部品のシミュレーションコードが自動生成される．得られたコードと設計者が記述した各機能モジュール群の動作記述を結合することにより，対象システム全体のシミュレーションコードを得ることができる．得られたシミュレーションコードを用いてシミュレーションを行うと，バスの使用率やデッドラインミスの回数などの出力が得られる．その結果，もし要求仕様を満たさない場合には，設計者はバス通信構造の改善を行う．提案ツールでは，複数バスをサポートするが，バスブリッジを持たない，各バスのバスクロックサイクルおよび各モジュールの動作クロックサイクルはすべて同一でなければならない，などの制約があるが，バス調停方式として，FIFO，固定優先度，EDF，TDMA，FTDMA などの調停方式を任意に階層的に組み合わせた調停方式を指定できる．また，単一リード/ライト・バーストリード/ライト・スプリットトランザクションによるリード/ライトなどのバスアクセス方式や，バスを使用せずモジュール間をフリップフロップを介して結線で直接接続する単方向通信などもサポートしている．

提案手法の適用実験として，提案ツールを共有メモリ方式の二次元グラフィクス描画表示システムの設計に適用し，その性能改善を行った結果を示す．

提案手法/ツールと従来手法/ツールとの比較は次のとおりである．CoWare, Inc. の ConvergenSC[6] をはじめとする多くの商用ツールでは，我々が知る限り Bus Compiler[7] を除いて，バスプロトコルとして AMBA[14] などの標準的なバスプロトコルのみをサポートし，ユーザが自由にバスプロトコルをカスタマイズすることを許していないため，バスプロトコルの変更によるバス通信の改善を行うことができない．また，文献 8) などのような，機能モジュールの再分割も

☆ 並行スレッドをサポートしたオブジェクト指向言語であれば，他の言語にも適用可能な手法となっており，たとえば SystemC にも容易に移植可能である．SystemC ではサイクル精度記述のためのライブラリ関数は標準で用意されており，自前で開発する必要はない．それにもかかわらず Java を採用した理由は，単に我々の研究グループが SystemC によるプログラミングを行える人材を欠いていたためである．

含めた包括的な最適化手法は新規設計には向いているが，機能モジュールの IP 再利用を行うには不向きであり，そのような場合には，通信のみを改善するアプローチ（communication refinement）が有効である．通信の改善に関する既存研究のうち，文献 9) はネットワークオンチップ（NoC）が対象であり，単純なバスシステムは対象としていない．また，文献 10) で提案されているバスシステムを対象とした通信の自動改善手法は，改善の方法が固定であり，複数の候補から最も良い設計を選択するといった場合は考慮されていない．文献 7) では，バスをパイプラインや FSM で記述し，シミュレーションコードを自動生成するツールを提案している．文献 7) は設計対象がマルチプロセッサコア SoC である点が本研究と若干異なるが，バスのモデル化に関しては，プロトコルなどの既存 IP を変更可能としている点などで，提案手法と目指す目標が類似している．文献 7) ではバス自体も機能モジュールとして記述するため，記述能力は一般の NoC を記述できるほど大きく，探索できる設計空間は提案ツールより広いと考えられる．一方，提案ツールは単純なバスと階層バス調停方式によるバス通信構造記述に特化しており，バス調停方式の具体的な制御を直接記述する必要がないため，このクラスのバスを用いて低コスト志向のバスシステム設計を行う場合は文献 7) のツールよりも使い勝手が良いと考える．

本論文の構成は以下のとおりである．まず 2 章で提案手法が対象とするバスシステムについて述べる．3 章で提案する支援ツールの概要を説明し，4 章で適用実験の概要とその結果を示す．最後に，5 章で結論と今後の課題について述べる．

2. 設計対象バスシステム

本章では，提案手法が対象とするバスシステムのクラスについて述べる．設計対象となるシステムは，**図 1** のように複数個のモジュールがバスに接続されており，各モジュールはバスを介して他のモジュールのレジスタを読み書きすることにより通信を行うようなバスシステムである☆．バストポロジは一般に複数本のバスを用いて自由に構成可能である．各モジュールは，図 1 の Module のように動作内容を記述した制御回路，コマンド/ステータスなどの外部との入出力データを保持するためのレジスタ（以後，**入出力レジスタ**と呼ぶ），および接続しているバスへの出力バッファからなり，入出力レジスタの個数と出力バッファサイズは自由に指定可能である．また，バスごとにメッセージ競合時の調停を行う調停回路が存在する．さらに，バスを使用せずモジュール間を直接接続する単方向通信路の存在も許す．

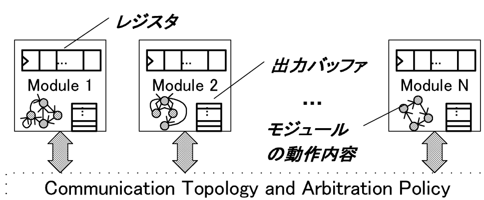

図 1 設計対象システム
Fig. 1 Target system.

2.1 モジュール

バスに接続される各モジュールは，様々な機能を実現する部品であり，たとえば CPU，DMA コントローラ，メモリ，DSP，各種コーデック，入出力インタフェースなどが該当する．各モジュールはその仕様によって機能が定められた入出力レジスタ，有限制御部（有限状態機械）および内部レジスタから構成される制御回路，および，バスへの出力データを一時的に格納する出力バッファを持つものとする．

2.1.1 モジュール制御回路および動作

モジュール制御回路は，一般に有限制御部および内部レジスタを持ち，その動作内容は，内部レジスタ上の演算・代入，自モジュールの入出力レジスタへのアクセス，バスを介した他モジュールの入出力レジスタへのアクセス，if，while，for などの制御文を用いたサイクル精度記述で与えられるとする．本手法での動作記述は，バスアクセス動作やサイクル精度記述をサポートするライブラリを持つ Java プログラムで記述する．ただし，ハードウェアでの実現の容易性を考慮して，プロセスの再帰呼び出しを行わないなど，一定の文法制限を与えている（詳細は 3.6 節参照）．

2.1.2 入出力レジスタ

各モジュールはそれぞれ入出力レジスタを持つ．他のモジュールへのデータ転送は，後述するバスアクセスメソッド（2.4 節参照）に送信元（入出力）レジスタと送信先（入出力）レジスタなどを指定して行う．なお，一般に入出力レジスタは他のモジュールからのアクセスも許される．他モジュールからアクセス中の場合，自モジュールからのアクセスは無視される．アクセスできたか否かの判定が必要な場合は，設計者がモジュールの動作として明示的に記述するものとする．送信元レジスタと送信先レジスタ間にはデータ転送の

☆ レジスタの読み書きによってモジュール間通信を行うアーキテクチャは文献 11) など多くのバスシステムで用いられている．

ためのバスが必要であり，バスが存在しない場合はエラーとなる．

なお，モジュールの仕様に依存して必要な入出力レジスタ数が一般に異なるため，本手法では各入出力レジスタは一般にレジスタファイルで構成されるものとし，レジスタファイルに含まれるレジスタの個数を設計者が明示的に与えるものとする．さらに，入出力レジスタ名とインデックス（＝レジスタファイル中の各レジスタのアドレス）の組によって送信元/送信先レジスタ指定を行うものとする．

2.1.3 出力バッファ

出力バッファは一般にバスへの出力データを一時的に格納するストレージ部品であり，論理的にはFIFOやEDFなどの優先度に従って出力順が決定される有限長のキューであるとする．また，バッファサイズ（＝キューの長さ）はパラメータとして設計者が明示的に与えるものとする．

2.2 バストポロジ

バストポロジは，バスとモジュールの接続関係を表すものである．提案ツールでは，複数本のバスを用いて自由にトポロジを構成可能である．ただし，バスブリッジは考慮していない．

2.3 バス調停方式

調停は複数個のメッセージ間でバスの使用権を争う場合に必要となる．調停方式に従い調停が行われ，最高優先度のものにバスの使用権が与えられる．調停方式としては，FIFOや固定優先度，EDF，TDMA，FTDMAが利用可能である．固定優先度では，データ転送ごとに優先度を指定できる．EDFでは，デッドラインの近いものから順にデータ転送される．

2.3.1 階層バス調停方式

高性能なバス調停を実現するためには，階層的なバス調停方式が指定できることが望ましい.[5] 一般に階層バス調停方式は，その階層構造を利用して各ノードで勝ち抜き戦を行い（トーナメント方式），バスを使用できるメッセージを1つ選ぶ．ただし，1つのメッセージがトーナメントの複数の対戦に参戦してもよい．このような調停方式は**調停木**と呼ばれる木構造のグラフまたはDAG（Directed Acyclic Graph）で表現される．**図2**に調停木の例を示す．図2(a)が木構造による調停木の表現であり，図2(b)は(a)と等価で，かつ，共通部分木を1つにまとめたDAGによる調停木の表現である．この例では，最初の10サイクルはMessage1がバス権を取り，次の30サイクルはMessage2とMessage3のいずれか一方が優先度に従いバス権を取り合い，次の20サイクルはMessage3

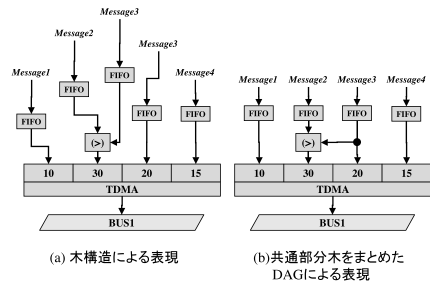

図2 階層バス調停方式の例
Fig. 2 Example of hierarchical bus arbitration policy.

が，最後の15サイクルはMessage4がバス権を取り，これを繰り返す．なお，調停木の葉ノード（図2ではFIFOと示されたノード）は2.1.3項で説明した各モジュールの出力バッファに対応し，「(>)」[「(<)」]で示されたノードは，図において左側[右側]の入力を優先的に選択する固定優先度調停方式を表すとする．

2.4 バスアクセスメソッド

バスアクセスメソッドとは，バスアクセスのために用いる手続き（メソッド）であり，リード/ライトの別，送信元/送信先レジスタ，転送するデータ単位（ワード）数，転送に対する実時間制約（デッドライン）などを指定する．バスシステムは，バスによって接続された複数個のモジュールがバスを介して互いに通信を行い動作するため，バスを用いてデータ転送を行うにはバスアクセスメソッドを用いる．

バスアクセスメソッドはデータ転送にかかるクロックサイクル数に影響を与える．たとえば，バスアクセスメソッドには，データを1ワードずつ転送するシングル転送方式と，連続したアドレスに対する複数ワードの連続転送であるバースト転送方式があるが，いくつかのシングル転送をまとめて単一のバースト転送に変更すると，シングル転送では1ワードごとに必要なアドレス転送が不要となり，転送にかかるクロックサイクル数を抑えることができる．

2.5 バスプロトコル

バスプロトコルは，バスを用いるデータ転送のための手順やデータフォーマットなどを規定したものである．提案ツールでは，バスへのアクセス動作を実装したメソッド（バスアクセスメソッド）の集合をプロトコルと考える．

バスプロトコルは使用可能なデータ転送の種類や，そのデータ転送に必要なクロックサイクル数，メソッドを呼び出す回数に影響を与える．たとえば，マルチキャスト可能なプロトコルと不可能なプロトコルでは，複数の入出力レジスタに同じデータを転送する場合

に，単純にメソッドコール回数が異なる．ただし，複雑なプロトコルは実装が困難なだけでなく，回路面積やデータ転送にかかるオーバヘッドが大きくなる可能性がある．

2.6 モジュール間を直接接続する単方向通信路

バスを使用せずモジュール間を直接接続する単方向通信路を，必要に応じて用いることができる．この通信路は概念的には次のような性質を持つ共有変数である．

- 共有変数に書き込み（送信）可能なモジュールと読み込み（受信）可能なモジュールは異なるモジュールであり，書き込み可能なモジュールはたかだか 1 つのみである．
- 共有変数に書き込んだ値は，次のクロックサイクル以降に別モジュールから読み出せる．

ハードウェアにおいては，モジュール間をフリップフロップを介して直接結線で接続することにより実装可能である．本論文では，後述する WAIT 信号を用いたバスプロトコルの実装に使用する．

3. バスシステム設計支援ツール概要

提案ツールは，シミュレーションコードの構築および変更の際に，設計者によるコード変更量を削減するものである．本章では，提案ツールの概要について順に述べる．

3.1 要求される機能

提案ツールは，バスシステムのシミュレーションコードを Java のスレッド機構を用いた並行プログラムとして自動生成するものである．一般に，並行に動作するハードウェアシステムでは，複数のモジュールがクロックに同期して並行に動作を行っている．モジュール間のバス権獲得のタイミングや競合などの状況を正確に再現するためには，(1) 対応する Java プログラム上で複数モジュールの時間経過を正確にシミュレートできる必要がある．また，(2) シミュレータの実行速度に依存しない各メッセージ転送の実時間違反検出や，(3) バスアクセスの排他制御も同時に要求される．

(1) は，文献 12) と同様に，全モジュールが 1 クロックサイクル終了するまで待ち合わせ，その後，次のクロックサイクルに進行させることで（バリア同期方式），複数モジュールの動作をクロックの進行にあわせて正確に再現している．ただし，このために設計者はサイクル精度記述を行う必要がある．(2) は，上記クロックサイクルを数え，バス上のデータ転送に関する実時間制約の違反判定を行うことにより実現している．また，各モジュールごとの実時間制約違反メッセージ数など，設計者が設計改善を行うにあたり有用な情報なども収集する．(3) は，調停木の各ノード（**調停ノード**と呼ぶ）に対応するソフトウェア部品を用意し，調停ノードは子ノードからバス権獲得を待つ複数のメッセージを入力し，指定した調停方針に従って最も優先度の高いものを選び，それを親ノードに出力することによって実現している．

3.2 提案ツールを用いた設計手順

提案ツールを用いた設計手順を**図 3** に示す．まず，設計者は，動作周波数，バストポロジ，バス調停方式，およびバスプロトコルなどの指定からなる**システム構成情報**（詳細は 3.3 節）を記述し，**Java スケルトンコード生成器**（Java Skeleton Code Generator: 以下，**JSCG** と呼ぶ）に入力する．JSCG は入力されたシステム構成情報に基づいて，シミュレータに必要な機能を実装した Java クラスライブラリ（以下，**バスシステムモデル化パッケージ**と呼ぶ．詳細は 3.4 節）を利用して Java コードを自動生成する．得られた Java コードではバスに接続する各モジュールの動作内容が空欄となっている．以下，このような一部が空欄となっている Java コードを **Java スケルトン**と呼ぶ．これに対して，設計者はバスシステムモデル化パッケージのメソッド，および Java に標準で用意されている演算子や制御文などを用いて，空欄になっている各モジュールの動作内容をサイクル精度記述で与える．ただし，バスアクセスメソッドは，「read/write(”バスアクセスメソッド ID”，送信元レジスタインデックス，送信先レジスタインデックス，デッドライン☆，メッセージ ID)」の形式で仮指定しておき，仮指定に対して**メソッド/データ転送バス対応表**を用いて外部から設定を与えることにより，実際に用いるバスアクセスメソッドの実装および転送に用いるバスが指定される（詳細は

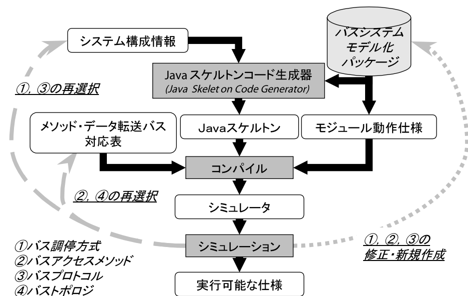

図 3 提案ツールを用いた設計手順
Fig. 3 Design process using proposed CAD tool.

☆ バスアクセスの実時間制約（デッドライン）は通信メソッドの引数として与える．

3.7 節参照）．これにより，設計者が直接 Java コードを変更することなく，メソッドの種類やデータ転送に用いるバスを変更することが可能である．

3.3 システム構成情報

システム構成情報は，JSCG の入力情報であり，動作周波数や，リード/ライト命令の実行に必要なクロック数，モジュール，入出力レジスタの個数，バス接続情報，出力バッファサイズ，調停木などからなる．

図 4 に記述例を示す．この例では，ModuleA〜ModuleD までの 4 モジュールが 2 つのバス（BUS1, BUS2）に接続されており，各モジュールの入出力レジスタ個数および出力バッファサイズがそれぞれ指定されている．また，BUS1 経由では入出力レジスタ regA，regB，regC1，regD1 にのみアクセス可能で BUS2 経由では入出力レジスタ regC2，regD2 にのみアクセス可能であることが指定されている．さらに，システム全体の動作周波数の指定☆，BUS1 および BUS2 がそれぞれ用いるバスプロトコル，バースト長，リード/ライト命令に必要なクロック数，調停方式を表す調停木（BUS1 の調停木は図 2 に同じ）が記述されている．なお，図 4 の MUX は，他モジュールからのバスを介した入出力レジスタへの入力がつねに自モジュールからの入力よりも優先されるように設定されたマルチプレクサを表す．

一般に，システム構成情報においては，システム全体は複数個のモジュールと複数個のバスおよび調停回路の組合せで構成し，各モジュールには複数個の入出力レジスタと複数個の出力バッファが存在し，それらが調停回路を経由してバスへと接続されている．また，調停回路は，木構造で表現されるトーナメント方式を用いて実現されており，調停ノードと葉ノードの組合せで記述される．これらは XML ファイル☆☆で記述し，JSCG にファイル引数として与えることにより入力される．

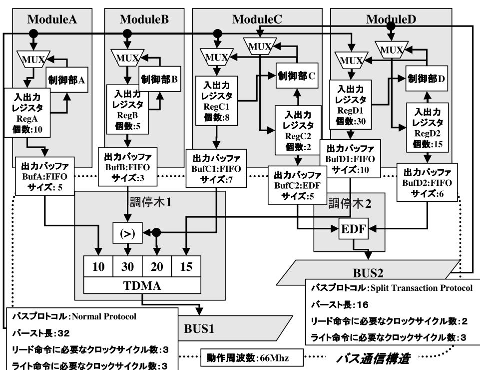

図 4 システム構成情報の記述例
Fig. 4 Example of system configuration.

3.4 バスシステムモデル化パッケージ

バスシステムモデル化パッケージは，シミュレータに要求される機能，バス調停方式，およびバスプロトコル（バスアクセスメソッドの集合）を実装した Java クラスライブラリである．

提案ツールでは，定義済みのバス調停方式として，2.3 節にあげた調停方式を，また，バスプロトコルとして，シングル・リード/ライトおよびバースト・リード/ライト命令を実装した Normal Protocol，Normal Protocol におけるリード命令をスプリット・トランザクションに変更した Split Transaction Protocol，WAIT 信号☆☆☆をサポートした WAIT Protocol などをライブラリとして用意する．設計者は，調停方式の組合せや，どのプロトコルを用いるかをシステム構成情報として指定する（図 4）．

3.5 Java スケルトンコード生成器

Java スケルトンコード生成器（JSCG）は，設計者によるコード変更量を削減することを目的として，システム構成情報から，それに対応したシミュレーションコードを自動生成する．具体的には，システム構成情報からシミュレータを構成する各ソフトウェア部品の接続関係の抽出および，その接続関係を実現する Java コードの生成を行う．

提案ツールでは，まずシミュレータの各ソフトウェア部品の接続関係をシステム構成情報から自動的に導出する．たとえば，図 4 から，モジュール ModuleB は入出力レジスタ regA，regB，regC1，regD1 および出力バッファbufB に接続していることが分かる．

一般に，どのレジスタがどのモジュールから見えるかなどの情報は，与えたバストポロジとモジュール・バス接続情報，および各モジュールが持つレジスタ情報から導くことができる．これらの情報に基づいて各モジュールが使用するソフトウェア部品（オブジェクト）への参照（ポインタ）を持つような Java コード

☆ 各バスアクセスの時間制約はクロックサイクル数ではなく実時間で与えるため，一般に同じ設計であっても動作周波数によって時間制約が満たされるか否かが変化する．したがって，動作周波数も設計改善のパラメータの 1 つであるため，システム構成情報に含めている．

☆☆ 具体的な文法や記述例は省略するが，本質的には図 4 に示すモジュール，バス，バッファ，調停ノードの相互接続関係や，バッファ長，プロトコル名などの各種属性を記述したものになっている．

☆☆☆ 入出力レジスタへのアクセスを禁止する信号であり，かつ，バスを用いずに直接全モジュールに通知する信号．

```
public class ModuleB extends Module {
    /* 出力バッファbufB への参照 */
    private LeafNode bufB;
    /* 入出力レジスタ regA,regB,regC1,regD への参照 */
    private NormalProtocolRegister regA;
    private NormalProtocolRegister regB;
    private NormalProtocolRegister regC1;
    private NormalProtocolRegister regD1;

    public ModuleB (ClockController cc,
                    ...
                   LeafNode bufB,
                   Register regA,
                   Register regB,
                   Register regC1,
                   Register regD1) {
        ...
        this.bufB = bufB;  /* bufB への参照の設定 */
        this.regA = (NormalProtocolRegister)regA;
        this.regB = (NormalProtocolRegister)regB;
        this.regC = (NormalProtocolRegister)regC1;
        this.regD = (NormalProtocolRegister)regD1;
                         /* regA,regB,regC1,regD1 への参照の設定 */
        regB.setOwner(this); /* regB が ModuleB に属することを指定 */

    }

    public void run() {
      /* ここにモジュール動作を記述 */
    }
}
```

図 5　図 4 のシステム構成情報から生成された Module テンプレート（ModuleB.java）
Fig. 5　Module template (ModuleB.java) generated from system configuration in Fig. 4.

を自動生成する．

生成される Java コードのうち，特に個々の機能モジュールに対応して生成されたコードを **Module テンプレート**と呼ぶ．Module テンプレートはスレッド動作記述部（*run()* メソッド）を持ち，そのモジュールからバスを介してアクセス可能なすべての入出力レジスタ，および，利用可能な出力バッファが参照可能となっている．このクラスは，各モジュールごとに作成される．なお，JSCG からの出力時点では，各モジュール処理内容は空であり，設計者がバスシステムモデル化パッケージの手続きを用いて，各モジュール動作内容を記述する．たとえば，図 4 の ModuleB は入出力レジスタ regA，regB，regC1，regD1 および出力バッファbufB にアクセス可能であり，自モジュールに属する入出力レジスタは regB であるので，**図 5** のような Module テンプレートが生成される．

このように，Java のようなオブジェクト指向プログラミング言語を用いることにより，ハードウェア部品とソフトウェア部品（オブジェクト）を直接対応付けられるため，提案ツールのようなシミュレーションコード自動生成が可能となる．

3.6　モジュールの動作記述

モジュールの動作は，**図 6** に示すように，各モジュールの動作に対応するスレッド動作記述部分（Module テンプレートの *run()* メソッド内）に記述する．動作内容は，設計者がサイクル精度で記述する．動作のど

```
public class ModuleB extends Module {
  /*
   入出力レジスタ，出力バッファへの参照などの初期設定
   (提案ツールにより自動生成)
   */
public void run() {
  /* ここからが設計者によるモジュール動作記述 */
  while(true) {
    /* 局所変数（内部レジスタに対応）の宣言文など */
    ...
    /* 局所変数に対する演算や代入 */
    int x=y*z;  /* (A) */
    ...
    /* x の値を自モジュールの入出力レジスタ regB の
       アドレス 0 に対して書き込み（バスは使用しない）       */
    regB.write(x,0);  /* (B) */
    /* 1 クロック消費（他のモジュールと同期）*/
    cc.consumeNClock(1); /* (C) */
    ...
    /* ModuleA の入出力レジスタ regA のアドレス 0 から 15 までを
       regB のアドレス 1 以降にデッドライン 0.63 秒以内に読み込む
       文字列"ModuleA_..."はバスアクセス ID                    */
    regA.read("ModuleA_regA_r16s0d1_0",0,15,0.63,...); /* (D) */
    if(regB.read(0)==0) { /* (E) */
      regA.write("ModuleA_regA_w1s0d5_1",...); /* (F) */
    }
    /* バスアクセスにかかるクロック消費は明示的に書かなくてもよい
       (手続き regA.read() や regA.write() の中で記述されている) */
    ...
  }
  /* ここまでが設計者によるモジュール動作記述 */
}
```

図 6　モジュール動作記述例
Fig. 6　Example of module behavior description.

こからどこまでが 1 クロックであるかを指定するために，1 クロック経過を待ち合わせる動作（クロック消費動作）を *cc.consumeNClock()* メソッドの呼び出しによって指定する．そのほか，演算子や制御文，バスシステムモデル化パッケージのバスアクセスメソッド（仮指定）を用いて自由に記述することが可能である．たとえば図 6 の記述は次のような動作を意味する．まず，局所変数 x に y と z の積を代入し（図 6 (A)），x の値を自モジュールの入出力レジスタ regB のアドレス 0 に書き込み（図 6 (B)），ここまでで 1 クロック消費する（図 6 (C)）．次に，ModuleA の入出力レジスタ regA のアドレス 0 から 15 までを regB のアドレス 1 以降にデッドライン 0.63 秒以内に読み込む（図 6 (D)）．最後に，もし regB のアドレス 0 の値が 0 ならば（図 6 (E)），regA に対して何らかの値を書き込む（図 6 (F)）．

ただし，*run()* メソッド内での手続きの再帰呼び出しや動的なソフトウェア部品生成（= オブジェクトのインスタンシエーション）を禁止している．これは，バスシステムの動作記述制約である．もし，これを許可する場合には，バスシステムの動作中に新たにハードウェアモジュールを生成する必要があるため，その記述を禁止している．なお，バスアクセスによるクロック消費に関しては，バスアクセスメソッド内で必要に応じて *cc.consumeNClock()* メソッド呼び出しが自動的に生成されているので，設計者はそれを意識する必要がない．

バスアクセスID	バスアクセスメソッド	データ転送バス
ModuleA_regB_r1s5d4_0	singleRead	bus0
ModuleA_regC_w1s0d5_1	singleWrite	bus0
ModuleA_regD_w20s0-20d0-20_3	burstWrite32 * 1	bus1
ModuleB_regC_w64s0-64d0-64_4	burstWrite32 * 2	bus1
ModuleC_regB_r16s0-16d0-16_5 ModuleC_regD_w16s0-16d0-16_6	readModifyWrite32 * 1	bus1

□ バスアクセスメソッドIDの形式
[(モジュール名)_(送信先レジスタ)_(r/w)(ワード数)
s(送信元インデックス)d(送信先インデックス)_(番号)]

バス名	データ幅
bus0	16
bus1	32

図 7　メソッド/データ転送バス対応表の例
Fig. 7　Example of Method-Bus table.

3.7　メソッド/データ転送バス対応表

メソッド/データ転送バス対応表とは，バスアクセスメソッドの仮指定を置き換えるための情報である．各モジュール動作記述内のバスアクセスメソッドに ID を振り，その ID を持つバスアクセスに実際にどの実装メソッドを与えるか，また，どのバスを用いるかを指定する．さらに，各バスのデータ幅を指定するための表も含む．**図 7** にメソッド/データ転送バス対応表の例を示す．ここで，バスアクセス ID は，[(モジュール名)_(送信先レジスタ)_(r/w)(ワード数) s(送信元インデックス)d(送信先インデックス)_(番号)] という形式で与えられているものとする．ここで，(モジュール名) はアクセス先の入出力レジスタが存在するモジュールの名前，(送信先レジスタ) はアクセス先の入出力レジスタ名，(r/w) は読み出し (r)/書き込み (w) いずれかの指定，(ワード数) は転送ワード数，(送信元インデックス) はアクセス元（すなわち自モジュール内の）入出力レジスタのアドレス，(送信先インデックス) はアクセス先の入出力レジスタのアドレス，(番号) は設計者が自由に指定する自然数である．たとえ (モジュール名) など ID を構成する他の要素がすべて同一であっても，(番号) が異なるものに対しては，異なるバスアクセスメソッドの実装を割り振ることが可能となる．

3.8　シミュレーションコードの変更方法

本節では，設計者による種々の設計変更に対応するシミュレーションコードの変更方法について述べる．提案ツールでは，設計変更のたびにシミュレーションコード全体を書き換える必要がなく，以下の変更により容易に可能である．

- **バス調停方式の再選択**：システム構成情報の調停木の指定を変更する．
- **バス調停方式の新規作成**： 調停ノードの実装コードを新規作成しバスシステムモデル化パッケージ内にライブラリとして追加する．
- **バスアクセスメソッドの再選択**：メソッド/データ転送バス対応表のメソッドの指定を変更する．
- **バスアクセスメソッドの新規作成**： バスアクセスメソッドの実装コードを新規作成し，バスシステムモデル化パッケージ内の入出力レジスタもしくは出力バッファライブラリにメソッドを追加する．
- **バスプロトコルの再選択**：システム構成情報のプロトコル，およびメソッド/データ転送バス対応表のメソッドの指定を変更する．
- **バスプロトコルの新規追加**： バスプロトコルはクロック制御やレジスタなどの複数のソフトウェア構成要素が連携して実現しているため，新しいバスプロトコルに対応するそれらの構成要素の実装コードを新規作成し，バスシステムモデル化パッケージ内にライブラリとして追加する．
- **バストポロジの再選択**：システム構成情報のバス接続情報を変更しバスを追加する．次いで，メソッド/データ転送バス対応表を変更する．なお，その際にモジュールの構成も変更する場合には，上記に加えシステム構成情報のモジュール指定および動作記述を変更する．

3.9　シミュレーションでの評価基準

本節では，性能改善の指標として用いる評価基準を述べる．

- **Deadline Missed**：デッドラインを超過したメッセージ数．
- **Bus Utilization**：バスの使用効率（= バスに実際にデータが流れていた時間の割合）．
- **Av. DLS（Average Deadline Slack）**：デッドラインとして指定した時間制約から実際のバスアクセスに要した時間を引いた値（= デッドラインスラック）の平均値．デッドラインスラックは，実際の処理がデッドラインと比較してどれだけ時間的余裕があるかを示す指標として用いることができる．
- **Min. DLS（Minimum Deadline Slack）**：デッドラインスラックの最小値．

なお，これらの値を求めるコードは，自動生成したシミュレーションコードにあらかじめ組み込まれており，シミュレーションコードの実行時に出力される．

4. 適用実験

提案ツールを共有メモリ方式の二次元グラフィクス描画表示システムに適用した結果を示す．ただし，例題のシミュレーションコードはバスの性能解析を高速に実行するために各機能モジュールの機能の抽象化を行ったモデルを用いる．ここでは**図 8** に示すように，バス調停方式に TDMA 方式を用いるものとして，その性能改善を行った流れを示す．

4.1　二次元グラフィクス描画表示システム

実験に用いた共有メモリ方式の二次元グラフィクス

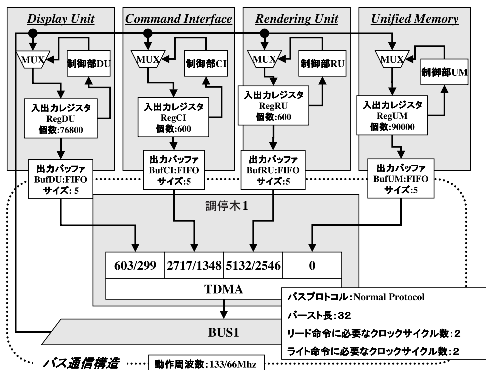

図 8 二次元グラフィクス描画表示システム
Fig. 8 Two-dimensional graphics rendering and displaying system.

描画表示システムの概要について述べる．このシステムでは，描画コマンドに従って表示データに描画を行い，320×240 の解像度（32 ビット/ドットとする）を持つ CRT ディスプレイに描画済み表示データを表示するものである．水平/垂直走査周期や帰線期間などの数値は文献 13) のものを用いる．水平走査周期は 63.56 μs であり，これを TDMA 周期として用いる．バスデータ幅は 32 ビットとし，バースト長を最大 32 個とする．また，1 回のリード/ライト命令に必要なクロックサイクル数をそれぞれ 2 とし，バーストリード/ライトでは，先頭データ以外は 1 データあたり 1 クロックサイクルでデータ転送できるとする．システム構成を図 8 に示す．図 8 において，TDMA の各タイムスライスの値は動作周波数 133 Mhz の場合と 66 Mhz の場合を併記している（詳細は次節で述べる）．各モジュールの仕様は以下のとおりである．

Unified Memory モジュール（以下，Unified Memory と呼ぶ）は，表示データと描画コマンドをまとめて格納する共有メモリである．これは VRAM とコマンド用メモリを合わせたものである．スレーブモジュールである．

Command Interface モジュール（以下，Command Interface）は，外部から受け付けた描画コマンドを Unified Memory に転送する．描画コマンドは，座標，コマンド，描画ソースデータ（使用するテクスチャ・ビットマップなど）の組で構成される．ここではモデルを単純化するため，すべての描画コマンドは同じデータサイズとし，一律 2.25 キロバイトとする．また，Unified Memory には描画コマンドを 5 つまで格納することができ，使用状況に合わせて更新する．各描画コマンドの使用状況を表すフラグとしては，入出力レジスタを用いる（フラグレジスタと呼ぶ）．フラグレジスタの値が 1 の場合には，対応するアドレスに書き込まれた描画コマンドは未実行あるいは実行中であり，新しい描画コマンドを上書きすることはできない．

Graphics Rendering Unit モジュール（以下，Rendering Unit）は，Unified Memory 内の描画コマンドに従い表示データを描画する．具体的には，描画コマンドを読み込み，指定されている個所に描画ソースデータを用いて描画処理を行う．描画の処理単位は 16×16 ドットとし，1 画面あたり 600 個☆の描画コマンドを処理する必要があるとする．また，1 ドット描画するのに 1 クロックかかるものとする．

Display Unit モジュール（以下，Display Unit）は，Unified Memory から表示データを読み込み，それを表示する．

4.2 TDMA スケジューリング

TDMA 周期としてディスプレイの水平走査周期の 63.56 μs を用いる．タイムスライスを割り当てるために，1 画面あたりの各モジュールの処理にかかるバスサイクル数を計算すると，下記のようになる．

Display Unit（DU）： DU が水平走査周期以内で読む必要のあるデータは $(240 \times 320) \times 32$ bit である．バス幅 32 bit でバースト長 32 個であるので，$(240 \times 320)/32$ 回のバーストリードを行う必要がある．1 回あたりのバーストリードは，バースト長 32 個であるので $2 \times 1 + 1 \times 31 = 33$ クロックかかる．よって，79,200 バスサイクル必要となる．

Command Interface（CI）： まず，描画コマンド 2.5 KB はバス幅 32 bit（4 Byte），バースト長 32 個であるので，1 回の描画コマンドの転送に $((2.25 \times 1{,}024)/4)/32 = 18$ 回のバーストライトが必要である．1 画面あたり 600 個の描画コマンドを処理するので，1 画面あたり $18 \times 600 = 10{,}800$ 回のバーストライトを行う．よって，$33 \times 10{,}800 = 356{,}400$ バスサイクル必要となる．

Rendering Unit（RU）： まず，描画コマンドをバーストリードする．これは CI の処理と同じ（リードライトが違うだけ）なので，356,400 クロック必要である．次に，描画処理の実行を行う．1 ドット 1 クロックかかり，処理単位 16×16，600 個の描画コマンド処理であるので，$16 \times 16 \times 600 = 153{,}600$ クロック必要である．最後に，UM に描画したデータを転送する．描画処理単位を 1 行ずつ転送する必要があるの

☆ 描画の重なりを考えて 2 画面分描画するものとして，$((320 * 240)/(16 * 16)) * 2 = 600$.

で，16 個ずつ転送を行う．16 個のバーストにかかるクロック数は $2 \times 1 + 1 \times 15 = 17$ クロックである．したがって，$17 \times 16 \times 600 = 163{,}200$ クロック必要である．よって，以上を合計すると，673,200 バスサイクル必要となる．

Unified Memory（UM）： スレーブモジュールであり，自らはバス権を取得しないので，0 バスサイクルである．

以上より，これらの比をとると，2 : 9 : 17 : 0 となる．動作周波数 133 MHz，66 MHz では，TDMA 周期 63.56 us が何クロックか計算すると，133 MHz では 8,454 クロック，66 MHz では 4,195 クロックとなる．したがって，各モジュールに割り当てるタイムスライスのクロック数は以下のようになる．

$$DU : CI : RU : UM = \begin{cases} 603 : 2{,}717 : 5{,}132 : 0 \text{（133 MHz の場合）} \\ 299 : 1{,}348 : 2{,}546 : 0 \text{（66 MHz の場合）} \end{cases}$$

ここで，1 画面あたりに各モジュールが行う処理時間（クロック数）を 262（水平走査が 262 ラスタある）で割り，1 水平走査周期で行う処理時間を算出すると，DU : CI : RU : UM = 302 : 1,364 : 2,569 : 0 となり，上記で求めた 133 MHz の場合の TDMA では十分に実行できる．一方，66 MHz では若干性能不足であることが分かる．

次節以降で，図 3 の設計手順に従い，バス調停方式の変更やプロトコルの変更を行うことにより性能の改善を試みる．

4.3 バス通信構造改善の方針

本実験におけるバス通信構造の改善の方針としては，要求性能を満足する設計をできるだけ短期間で求めることを主目的とし，回路実装の単純なものから出発し，徐々に複雑化させる方針をとる．この方針の場合，記述変更量が少なく，かつ回路面積やデータ転送にかかるオーバヘッドの増大などシステムの複雑さに与える影響の少ないものから順に試みるのが望ましい．したがって，具体的には以下の優先度順で改善を試みることにした．

(1) バス調停方式の再選択/新規作成
(2) バスアクセスメソッドの再選択/新規作成
(3) バスプロトコルの再選択/新規作成
(4) バストポロジの再選択

(1)〜(3) の再選択に関しては，3.8 節で示したとおり，設計者はシミュレーションコードをいっさい修正することなく変更を行うことが可能である．また，新規作成に関しては，一般に (1)〜(3) の順でコード変更（作成）量が多い（具体的な変更方法は 3.8 節参照）．よって，この方針での設計改善の適用優先度は，上記の番号 (1)，(2)，(3)，(4) の順となる．

4.4 適用結果

各構成および設定での 1 画面分の描画表示処理終了後のシミュレーション結果を**表 1** に示す．まず，図 8 に示した動作周波数 133 MHz，バースト長 32，TDMA，および Normal Protocol のシステムで，1 画面分の描画処理のシミュレーションを行った結果，実時間制約を満たすことを確認した（表 1-①，Total の Deadline Missed の数が 0）．次に，動作周波数を 66 MHz に変更してシミュレーションを行った結果，見積りからの予測どおり実時間制約を満たさなかった（表 1-②，Total の Deadline Missed の数が 0 でない）．

そこで，まずは最も記述変更量の少ないバス調停方式の再選択による設計変更によって実時間制約を満たすことを試みた．表 1-②の結果より，特に Command Interface や Rendering Unit でデッドラインを超過しているメッセージ数が多く，TDMA のタイムスライスに余裕がないと考えられる．そこで，これらのモジュールが余裕のある他のモジュールのタイムスライスを（もし空いていれば）共有可能とするために，**図 9** の (a)，(b) に示す調停木に変更し，再びシミュレーションを行った．しかし，それでも実時間制約を満たさなかった（表 1-③，④）．これは，共有するタイムスライスに空きがないためだと考えられる．各モジュールの転送にかかるサイクル数を抑え，タイムスライスに空きを作る必要がある．よって，バースト長を大きくし，データ転送にかかるクロックサイクル数を抑えることを試みた．バースト長を 32 から 64 に変更した．これはバスアクセスメソッドの再選択による設計変更である．3.8 節に示すとおり，バースト長を変更したメソッドの追加はメソッドバースト長を表す数値を変更するのみであり，記述変更量は小さい．変更の結果，DU のメッセージは制約を満たすことを確認した（表 1-⑤，Display Unit の Deadline Missed の数が 0）．そこで，バースト長の変更が効果的であった DU のみバースト長を 320 まで変更した☆（表 1-⑦，⑧）．さらに，同一の処理を少ないクロックサイクル数で行うために，バスプロトコルの再選択による設計変更を行った．ここでは，フラグレジスタの確認のための無駄なバスアクセスをなくすため，WAIT Protocol に変更した．これにより，バス上には有効なメッセージ

☆ バースト長は 64 程度までが通常上限であるが，本研究では標準バスを用いるのではなくシステムにあった専用バスを作ろうとしているので，ここでは非標準的なバースト長も許すものとして考えている．

表 1 シミュレーション結果
Table 1 Simulation results.

	Total					Display Unit		Command Interface		Rendering Unit	
	# of Message	Deadline Missed	Bus Utili (%)	Av. DL Slack(ms)	Min. DL Slack(ms)	# of Message	Deadline Missed	# of Message	Deadline Missed	# of Message	Deadline Missed
① 133MHz bl32 TDMA Normal	1562841	0	54.137	12.915	8.063	130304	0	585794	0	846743	0
② 66MHz bl32 TDMA Normal	1622581	31716	84.239	8.678	−0.665	137184	1120	608156	11767	877241	18829
③ 66MHz bl32 TDMA+FIXED0 Normal	1641260	119270	92.758	7.792	−2.458	137376	1056	709338	53740	794546	64474
④ 66MHz bl32 TDMA+FIXED1 Normal	1621473	136823	92.257	7.597	−5.138	106784	17952	720133	54461	794556	64410
⑤ 66MHz bl64 TDMA+FIXED0 Normal	1518835	339595	88.969	5.286	−7.548	139968	0	607827	147221	771040	192374
⑥ 66MHz bl64 TDMA+FIXED1 Normal	1551851	308290	90.458	5.695	−6.261	112064	14976	624903	124826	814884	168488
⑦ 66MHz bl320-64 TDMA+FIXED0 Normal	1529622	337270	89.487	5.376	−7.471	145280	0	610538	146226	773804	191044
⑧ 66MHz bl320-64 TDMA+FIXED1 Normal	821611	288944	70.648	2.877	−13.650	145280	0	619323	277594	57008	11350
⑨ 66MHz bl320-64 TDMA+FIXED0 WAIT	1502284	120640	76.677	8.830	−8.217	145280	0	691200	0	665804	120640
⑩ 66MHz bl320-64 TDMA+FIXED1 WAIT	1500032	114436	76.562	8.936	−7.945	144320	0	691200	0	664512	114436
⑪ 66MHz bl320-64 TDMA+FIXED2 WAIT	1723565	0	88.075	9.768	0.903	144320	0	691200	0	888045	0

(＊)「番号　動作周波数　バースト長　バス調停方式名　バスプロトコル名」の順

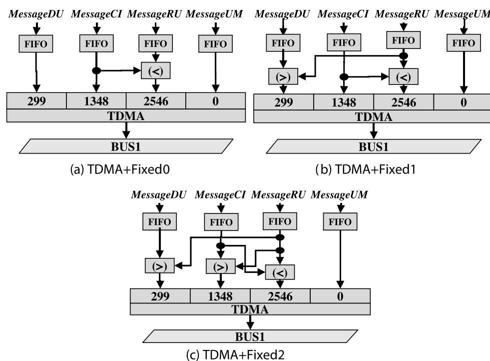

図 9 改善を行った調停木
Fig. 9 Refined arbitration trees.

のみが流れバスに空きができる．バスプロトコルの変更は，3.8 節で述べたようにシステム構成情報の指定を変更するのみでよい．変更の結果，デッドラインを超過するメッセージ数が減少した（表 1-⑨，⑩，Total の Deadline Missed の数が減少）．また，各モジュールのタイムスライスにわずかながら空きが増えていた（Total の Av. DL Slack が増加し，Display Unit に加え Command Interface の Deadline Missed も 0 になった）．よって，ここでもう一度，デッドラインを超過しているメッセージ数が多くタイムスライスに余裕の無い Rendering Unit が，余裕のある Display Unit や Command Interface のタイムスライスを（もし空いていれば）利用可能な調停木を考えた．図 9 (c) にそれを示す．変更の結果，実時間制約を満たすことに成功した（表 1-⑪）．

なお，上記のような流れによりシミュレーションによる動作確認が行われた Java コードは，実行可能な仕様として扱うことができる．

5. おわりに

本論文では，バス調停方式，バスアクセスメソッド，バスプロトコル，およびバストポロジの設計改善におけるシミュレーションモデルの変更を容易化したバスシステム設計支援ツールを提案した．これを用いることにより，バス通信構造の仕様検討を効率良く行うことが可能となる．

提案ツールにおいてはシステムのモジュール動作をすべてシステムレベルで記述し，バス部分を生成して得られたシミュレーションコード自体を実行可能な仕様として扱うため，この仕様どおりに実装されるという前提であれば，シミュレーションによって得られた結果と実装システムでの結果を（誤差なしに）一致させることは可能である．ただし，実際にはバス性能の見積りのみを目的として各モジュールの記述を若干抽象化した形で実装する方が，コード記述の労力が削減され，シミュレーション速度も向上する．その場合，4 章の表 1 で示した実験のように，実際のシステム設計よりもワーストケースになるように抽象化したシミュレーションコードを記述すれば，シミュレーション結果が実時間制約を満たせば必ず実際のシステム設計でも実時間制約を満たすことを保証することができる．ただし，抽象化の度合いが大きすぎると，実際よりも回路コストの大きいバス通信構造を得る可能性がありうる．したがって，シミュレーション結果の誤差を小さくすることと，コード記述の労力削減およびシミュレーション速度向上とはトレードオフの関係にあるといえる．

今後は，バスブリッジへの対応，および AMBA[14] などの SoC 標準バスや階層的 CAN バス[15] の実装を行い，提案ツールのサポート範囲を拡大していく予定である．

一方，本研究によって，バス通信構造をシステム構成情報によってパラメータ化し，それが要求性能を満たすか否かの評価をシミュレーションコード自動生成により効率良く行うことが可能になったため，システム構成情報の各パラメータ値を変えながらシミュレーションによる評価を繰り返し行うことにより，各種パラメータのチューニングを自動的に行うことも可能と思われる．具体的な自動化手法については今後の課題である．

また，静的なワーストケース解析を行ってシミュレーションを行うことなく実時間制約を満たすか否かを検証するアプローチも考えられる．現実的には考慮すべき状態数が多すぎて，シミュレーションよりも効率的な検証が行えない可能性があるが，特にデッドラインを超えることが致命的なハードリアルタイムシステムに対しては 100%の保証が得られる静的解析が有効と思われるため，静的解析によるワーストケース見積り手法の開発も今後の課題としたい．

参 考 文 献

1) Pinto, A., Carloni, L. and Sangiovanni-Vincentelli, A.: Constraint-Driven Communication Synthesis, *Proc.39th Design Automation Conference* (2002).
2) Ortega, R. and Borriello, G.: Communication Synthesis for Distributed Embedded Systems, *Proc. Int. Conf. on Computer Aided Design*, pp.437–444 (1998).
3) Lahiri, K., Raghunathan, A., Lakshminarayana, G. and Dey, S.: Communication Architecture Tuners: A Methodology for the Design of High-Performance Communication Architectures for System-On-Chips, *Proc. 37th Design Automation Conference*, pp.513–518 (2000).
4) 高橋美和夏，宮嶋浩志，福井正博：大規模 SoC バス・アーキテクチャ性能評価手法，情報処理学会論文誌，Vol.44, No.5, pp.1225–1231 (2003).
5) Meyerowitz, T., Pinello, C. and Sangiovanni-Vincentelli, A.: A Tool for Describing and Evaluating Hierarchical Real-Time Bus Scheduling Policies, *Proc.40th Design Automation Conference* (2003).
6) CoWare, Inc.: ConvergenSC Technical Overview. http://www.coware.com/products/convergensc_techoverview.php
7) Wieferink, A., Leupers, R., Ascheid, G., Meyr, H., Michiels, T., Nohl, A. and Kogel, T.: Retargetable Generation of TLM Bus Interfaces for MP-SoC Platforms, *Proc. CODES+ISSS'05*, pp.249–254 (2005).
8) Wieferink, A., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G. and Nohl, A.: A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms, *Proc. DATE'04* (2004).
9) Kogel, T., Doerper, M., Wieferink, A., Leupers, R., Ascheid, G., Meyr, H. and Goossens, S.: A modular simulation framework for architectural exploration of on-chip interconnection networks, *Proc. CODES+ISSS'03* (2003).
10) Abdi, S., Shin, D. and Gajski, D.: Automatic Communication Refinement for System Level Design, *Proc. 40th Design Automation Conference*, pp.300–305 (2003).
11) 株式会社ルネサステクノロジ：SH7040 シリーズユーザズマニュアル Rev.6.0. http://www.renesas.com/avs/resource/japan/jpn/pdf/mpumcu/rjj09b0031_sh7040.pdf
12) Liao, S., Tjiang, S. and Gupta, R.: An Efficient Implementation of Reactivity for Modeling Hardware in Scenic Design Environment, *Proc. 34th Design Automation Conference* (1997).
13) Renesas Technology Europe: HD64413A Q2SD User's Manual Rev.2.0. http://www.eu.renesas.com/documents/automotive/hd64413a.pdf
14) ARM Holding plc: AMBA 2.0 Specification. http://www.arm.com/products/solutions/AMBA_Spec.html
15) Robert Bosch GmbH: CAN Homepage of Robert Bosch GmbH. http://www.can.bosch.com/

(平成 17 年 2 月 7 日受付)
(平成 17 年 12 月 2 日採録)

谷本　匡亮

平成 5 年北海道大学理学部数学科卒業．平成 7 年大阪大学大学院理学研究科数学専攻博士前期課程修了．同年（株）日立製作所半導体事業部に入社．平成 15 年より（株）ルネサステクノロジへ転籍．同年大阪大学大学院情報科学研究科情報ネットワーク学専攻博士後期課程に社会人入学．動作合成，実時間並行システムの設計手法および形式的検証手法に関する研究に従事．

北口　　智

平成 14 年会津大学コンピュータ理工学部コンピュータソフトウェア学科卒業．平成 16 年大阪大学大学院情報科学研究科博士前期課程修了．現在，松下電器産業（株）勤務．大阪大学在学中，実時間並行システムの設計手法に関する研究に従事．

中田　明夫（正会員）

平成 4 年大阪大学基礎工学部情報工学科卒業．平成 9 年同大学大学院博士課程修了．博士（工学）．同年広島市立大学情報科学部助手．平成 12 年より大阪大学大学院基礎工学研究科助手．現在，同大学大学院情報科学研究科助教授．実時間システム，並行分散システムの設計手法および形式的検証法に関する研究に従事．

東野　輝夫（正会員）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．工学博士．同年同大学助手．現在，同大学大学院情報科学研究科教授．分散システム，形式記述技法，モバイルコンピューティング，通信プロトコル等の研究に従事．電子情報通信学会，ACM 各会員．IEEE Senior Member.