システムＬＳＩ設計技術　107－4
(2002. 11. 28)

実時間制約を有する単一バスシステムのJavaによるモデル化およびパラメトリックモデルチェッキングを用いた設計手法の提案

北口　智† 谷本　匡亮†† 中田　明夫† 東野　輝夫†

† 大阪大学大学院情報科学研究科　〒560–8531 大阪府豊中市待兼山町 1–3
†† 株式会社日立製作所半導体グループ　〒187–8588 東京都小平市上水本町 5–20–1
E-mail: †{kitaguti,nakata,higashino}@ist.osaka-u.ac.jp, ††tanimoto-tadaaki@sic.hitachi.co.jp

あらまし　近年、システムLSIの果たす役割はその重要性を増している。また、システムLSIにおいては実時間制約を如何に満たすかが、しばしば問題となる。更に、システムLSIを実装する上で要求性能を満たすように設計する場合、バスシステムの設計が重要となる。しかし、バスシステムを実時間制約を満たすよう効率よく設計するための設計手法がシステムシミュレーションによる方法以外提案されていないのが現状である。本論文では、Javaとパラメトリック・モデルチェッキングを用いた、実時間制約を満たすバスシステムの新規設計手法を提案する。
キーワード　システムLSI, 設計手法, バスシステム, 実時間制約, Java, パラメトリック・モデルチェッキング

Modeling Single Bus System with Real-time Constraints by Java and Design Methodology using Parametric Model Checking

Tomo KITAGUCHI†, Tadaaki TANIMOTO††, Akio NAKATA†, and Teruo HIGASHINO†

† Graduate School of Information Science and Technology, Osaka University
1–3, Machikaneyama, Toyonaka, Osaka 560–8531, Japan
†† Hitachi, Ltd., Semiconductor & Integrated Circuits
5–20–1, Josuihoncho, Kodaira, Tokyo 187–8588, Japan
E-mail: †{kitaguti,nakata,higashino}@ist.osaka-u.ac.jp, ††tanimoto-tadaaki@sic.hitachi.co.jp

Abstract　Recently, the role of system LSIs becomes more important. Moreover, it is often the problem whether system LSIs satisfy real-time constraints. Also, it is important to design bus systems so that we can implement system LSIs which satisfy the required performance. However, currently no other methodologies but system simulation have been proposed so far in order to design such bus systems which satisfy real-time constraints. In this paper, we propose a new design methodology to design bus systems with real-time constraints by modelling them in Java and using parametric model checking.
Key words　system LSI, design methodology, bus system, real-time constraints, Java, parametric model checking

1. まえがき

近年、VLSIなど大規模化するハードウェアシステムを効率よく設計するために、有限状態機械（ＦＳＭ）モデルなどで記述された仕様からハードウェア回路を計算機で自動合成するツールが実用化されてきている。しかし、ソフトウェアの開発と異なり、ハードウェアの開発においては、一旦量産を始めたハードウェアシステムにもし欠陥が見つかれば修正は困難であり、その場合にはハードウェアメーカーは多大な損失を被ることになる。従って、設計仕様の段階でシステムの欠陥がないことを保証することが重要である。そのような背景で、ハードウェアなどのシステム仕様に欠陥がないことを計算機で自動検証する技術の研究は世界中の企業や大学で近年盛んに行われている。

一方、ソフトウェアの分野においてもC言語やJava言語などで記述された実際のプログラムコードから状態遷移モデルを抽出して検証を行う研究が行われるようになり、幾つかの手法及び処理系が発表されている[1]～[4]。特にJava言語はWindows、UNIXなどの各種OS、webサーバ／ブラウザ、携帯端末などさまざまなプラットフォームで用いられており普及率も高く、オブジェクト指向に基づいた記述が可能で、C／C++などと異なり処理の並行性（マルチスレッド機構）を記述する構文を標準で備えている。ハードウェアシステムをJava

言語で記述できれば、設計者は特定のハードウェア記述言語(HDL)を覚える必要が無いため、ソフトウェアプログラマをハードウェア設計業務に一時的に従事させる、またその逆を行うなど、人材活用の面でも有用である。また、設計段階でソフトウェアシミュレーションにより性能評価を行うことも可能である。さらに、システムのうち性能を要求される一部分をハードウェアで、残りをソフトウェアで実現するなどのハードウェア・ソフトウェアコデザインを行う上でも有用である。

そこで本研究は、Java言語によるハードウェアバスシステムのモデル化手法の提案および検証法の開発を目的とする。一般に任意のJava記述をハードウェア合成することは容易ではない。本研究では、まず、バス構成を固定した特定のターゲットアーキテクチャを定め、そのアーキテクチャに基づいたハードウェアを記述するためのJavaのデザインパターンを提案する。

次に、提案するデザインパターンに従ったJava記述から、文献[5]で提案する手法に従って、処理の制御フローおよびそれらの並列実行関係を表すC-CFG(Concurrent Control Flow Graph)(詳細は4.に記述)に変換し、さらにC-CFGから各処理にかかる時間をパラメタ化した並行時間オートマトン(Concurrent Timed Nondeterministic Finite Automaton,C-TNFA)を抽出する。C-TNFAは各モジュールに対応するTNFAが並行に動作するモデルである。パラメタ条件導出中に、C-TNFAから、バス権獲得の排他制御およびモジュール間の静的優先度を考慮して、並列実行される遷移群を逐次順序に変換(インタリーブ)し、単一のTNFAに変換する。得られたTNFA、および、システムが満たすべき望ましい性質(検証性質)を時相論理式RPCTL[6]にて与え、既存のパラメトリックモデルチェッカ[6]を用いて、検証性質を満たすための各パラメタに関する条件式(パラメタ条件)を求める。ただし、パラメタ条件導出の高速化のため、TNFAの抽出の際には検証性質に無関係な、連続した動作群は抽象化(Abstraction[7])によって単一の動作にまとめ、オートマトンの状態数を削減する。また、各遷移の実行回数に制限を設け、その仮定の下で探索する状態空間を削減し、得られたパラメタ条件を満たす解を既存の線形計画問題ソルバーであるLP_SOLVE[8]を用いて求める。解が存在するか、また存在したとしても得られた解が妥当であるか否かを設計者が確認し、もし妥当でなければ仮定を緩めて再度パラメタ条件を求めることを繰り返すことにより、妥当なパラメタ値を求める。同時にJava記述から抽象化を行わないC-CFGも抽出し、これに上記で求めたパラメタ値を代入することにより、要求された実時間制約を満たすハードウェアの合成が可能となる。

2. 提案する設計手法

提案する設計手法の概要を図1に示す。単一バス上のデバイスをJavaのrun()メソッドを用いて記述する。ここで、クロックはバリア同期を用いて実現する。

次いで、Javaコードを読み込み、中間表現に変換する。ここで、中間表現はC-CFG、C-TNFA、TNFAからなる。

TNFAとRPCTLで記述された実時間制約を読み込み、パラメトリック・モデル・チェッキングを実行し、入力RPCTLを満たすパラメータ条件を導出する。

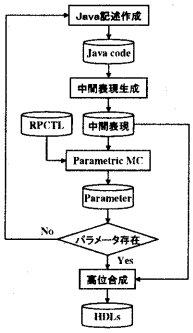

図1 提案する設計手法
Fig. 1 Proposed Design Methodology

パラメータ条件に解がなければ方式変更を行いJava記述を修正し、解があればその解を各パラメタ変数に代入したC-CFGを高位合成によりHDLへ変換する。

3. Javaによる単一バスシステムのモデル化

3.1 Java記述とバスシステムに対する制約

単一バス・システムのJavaによるモデル化を目的とするため、Java記述に対し下記制約を置く。

(1) ダイナミック・インスタンシエーションの禁止

(2) run()メソッドからのstart()メソッドコールの禁止

1.はハードウェアを扱っている事から許容可能な制約であると考えられる。また、2.はバス・プロトコルの検証を目的とする限りに於いては、直接バス動作に関わる部分のみをモデル化すれば良い為、許容可能な制約であると考えられる。

3.2 デザイン・パターン

単一バス・システムのJavaモデルは図2のUMLで与えられるデザインパターンに沿って記述する。バス上の各デバイス動作をDeviceImpl Class内のrun()メソッドに実装し、バスを介してアクセスがあるデバイス内レジスタはRegister Class内のattributeとして実装し、バスを介しての同期通信メソッドをRegister Classのメソッドとして実装する。また、バスに対応するBus Class、バスのロック管理を行うBusController Class、及びクロック同期を管理するClockController Classを実装する。

尚、Javaコードの再利用性を高めるため、下記方針にて、Javaコードの変更が可能となるようデザイン・パターンを定義している。

(1) デバイスの追加・削除

DeviceImpl Classの追加・削除

(2) デバイス動作の変更

DeviceImpl Classのrun()メソッドの変更

(3) 共有変数の追加・削除

Register Classのattributeの追加・削除

(4) バスプロトコルの追加・変更

Register Classの同期通信メソッドの追加・変更

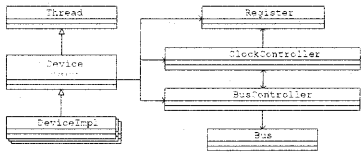

図 2　提案するデザインパターン (UML クラス図)
Fig. 2　Proposed Design Pattern (UML Class Diagram)

3.3　各クラスの説明

（1）Device クラス:　デバイスに共通の要素をまとめた抽象クラスである。実際のバス上に存在するデバイスはデバイスごとにその処理内容が異なると考えられる。よって、デバイスに共通の要素のみをまとめたクラスが必要となる。このクラスでまとめられている共通の要素は、Thread クラスの継承、そのデバイス自らが持つ Register Class オブジェクト、BusController Class オブジェクト、ClockController Class オブジェクトを参照するためのメンバ変数、このデバイスがアクセス可能な共有レジスタの情報を登録する為のメンバ変数、及びこのデバイスがアクセス可能な共有レジスタの情報を登録する為のメソッドである。

（2）DeviceImpl クラス:　実際のデバイスにあたるクラスである。デバイスに必要な要素をまとめた Device クラスを継承し、各デバイスに対応する異なる処理内容を記述する。

（3）Register クラス:　共有レジスタに対応するクラスである。レジスタの値を表すメンバ変数と、その値をリード/ライトするメソッドから成る。

（4）Bus クラス:　バスに対応するクラスである。バスが使用中であるか否かの状態を表すメンバ変数、そのバスに繋がっている共有レジスタを表すメンバ変数、及び、状態を変更するメソッドから成る。

（5）BusController クラス:　バスを制御を行うクラスである。具体的には、バスを介する共有レジスタへのアクセス時のバス権の獲得と解放を行なう。よって、バスを制御する処理は全てこのクラスに対して依頼することになる。また、特に、バスを介するセスが行われたことを表すバスアクセス・フラグメンバ変数（true でアクセスあり）と、その値を true に変更するためのメソッドを含む。これらを用いることにより、同一クロック内での複数回のバスロック動作を回避している。

（6）Clockcontroller クラス:　クロックの管理を行うクラスである。バスシステム上の各デバイスを、クロックに同期動作させます。詳細は次節のクロック同期メカニズムに記す。

3.4　クロック同期メカニズム

図 3 に示すバリア同期によるクロック同期メソッドを用いて、クロックを実現する。具体的には、1 クロック分の処理終了の通知を集め、全デバイスの処理終了が認められたら、次のクロックの処理を行なってよいという通知を行う。また、バスのロックが残っていない場合には、Bus Class のバスアクセス・フラグを false にリセットする。尚、クロック遷移はパラメトリック・モデル・チェッキングへの入力に際して、パラメータ

```
private void consume_1_clock() {
    /* 全デバイス数とこの method を実行したデバイスの数が等しいかチェック */
    if (++this.current_num == this.device_num) {
      this.current_num = 0;
      for (int i=0; i<this.device_num; i++) {
        /* レジスタ代入の実行 */
        registers[i].assignWriteValue();
      }
      /* バスロックを識別するフラグ変数（バスアクセス・フラグ）の初期化 */
     if (this.bc.getBusyCount() == 0) {
       this.bc.initLockDoneOnceFlag();
     }
     notifyAll();
     } else {
       try {
         wait();
       } catch (InterruptedException e) {
       }
     }
}
```

図 3　バリア同期によるクロック同期メソッド
Fig. 3　Clock Synchronization Method using Barrier Synchronization

```
public void assignWriteValue() {
    /* 後述の sync_write メソッド又は、sync_burst_write メソッドにより
       レジスタへの書き込みが実行がなされたかを判定 */
    if (this.write_access) {
        /* 実際に書き込みを行ったレジスタ（配列 index）への書き込みを
           実行 */
        this.current_value[this.update_index] = this.write_value;
        /* ライト・アクセス・フラグをリセット */
        this.write_access = false;
    }
}
```

図 4　レジスタへの値書き込みメソッド
Fig. 4　Method for Assigning Write Value to Registers

表 1　バス権獲得メソッド
Table 1　Bus Lock Methods

メソッド名	処理内容
getBusLock	tryGetBusLock メソッドが true を返せば、lock メソッドをコールすることでバス権を獲得し、false の場合、consume_clock メソッドをコールしクロック消費。
tryGetBusLock	getBusOwner メソッドの返り値が null、且つ getLockDoneOnceFlag メソッドの返り値が false なら、lock メソッドをコールし、getBusLock メソッドに true を返す。それ以外の場合は false を返す。
getBusOwner	現在バスをロックしているスレッドのオブジェクトを返す。ロックしているオブジェクトが無ければ null を返す。
lock	現在 Bus Class オブジェクト (lock メソッド) を実行しているスレッドを、バスをロックするスレッドとして登録する。
consume_clock	consume_1_clock メソッドをコールする。
consume_1_clock	クロック同期メカニズムで既に説明済み。
setLockDoneOnceFlag	引数をバスアクセス・フラグに代入する。
getLockDoneOnceFlag	バスアクセス・フラグの値を取得する。
initLockDoneOnceFlag	引数を false として setLockDoneOnceFlag メソッドをコールする。
assignWriteValue	共有レジスタへのライトアクセスがあった場合、次のクロックへ進む前にレジスタ代入を実行する。

化される為、必ずしも 1 クロック単位であるとは限らない。このクロック遷移のパラメータ化により、サイクル精度でのモデル化よりも抽象度の高いモデル化を実現する。

また、レジスタへの値書き込みには 1 クロックを要する為、それを実現する必要がある。これは、consume_1_clock メソッド内で、Register Class の assignWriteValue メソッド (図 4) をコールする事で実現している。

3.5　バス権管理の同期メカニズム

3.5.1　バス権獲得メカニズム

表 1 に示すメソッドにより、バス権獲得を管理する。バス権

表 2　バス権解放メソッド

Table 2　Bus Free Methods

メソッド名	処理内容
freeBusLock	unlock メソッドをコールする事でバス権を解放し、true を引数として setLockDoneOneceFlag メソッドをコールした後、consumu_clock メソッドでクロックを消費する。
unlock	現在バスをロックしているスレッドが、Bus Class オブジェクト（の lock メソッド）を実行しているスレッドかを確認し、そうであれば、バスをロックしているスレッドを null とする。
consume_clock	consume_1_clock メソッドをコールする。
consume_1_clock	クロック同期メカニズムで既に説明済み。
setLockDoneOnceFlag	引数をバスアクセス・フラグに代入する。
initLockDoneOnceFlag	引数を false として setLockDoneOnceFlag メソッドをコールする。
assignWriteValue	共有レジスタへのライトアクセスがあった場合、次のクロックへ進む前にレジスタ代入を実行する。

```
public void run() {
  Register other_r0 = (Register)super.access_registers.get(0);
  int  read_value;
  while (true) {
    this.do_something_w_or_wo_clock_boundary1();
    lread_value = other_r0.sync_read(super.bc, 1));
    this.do_something_w_or_wo_clock_boundary2();
}
```

図 5　sync_read メソッドの使用例

Fig. 5　Usage Example of sync_read method

の要求は getBusLock メソッド (記述例は [9] 参照) により行う。

3.5.2　バス権解放メカニズム

表 2 に示すメソッドにより、バス権解放を管理する。バス権の解放は freeBusLock メソッド [9] により行う。

3.6　バスへの排他的同期アクセス方式

3.6.1　排他的同期リード・メソッド

シングルリードは sync_read メソッドを DeviceImpl Class で実装する run() メソッド内でコールする事で行う。また、バーストリードは sync_burst_read, endBurstAccess, consume_clock メソッドを run() メソッドの synchronized ブロック内でコールする事で行う。

図 5 に、run() メソッドでの sync_read メソッド記述例を示す。尚、run() メソッドでの記述例に現れる未定義メソッド do_something_w_or_wo_clock_boundary(), 及び do_something_wo_clock_boundary() は、それぞれ、クロック境界を含んでもよい何らかの処理、クロック境界を含まない何らかの処理を意味する。

図 6 に、run() メソッドでのバーストリードの記述例を示す。バーストリードでは、バスのロックはコールする度に重ねて獲得し、バスの解放を行わずに値を返す。バーストリード回数のリードが終了後、重ねたロックを一気に解放する。この実装により、毎サイクル、リード値が返ってくるバーストリードを実現している。

3.6.2　排他的同期ライト・メソッド

シングルライトは sync_write メソッドを run() メソッドでコールする事で行う。また、バーストライトは sync_burst_write, endBurstAccess, consume_clock メソッドを run() メソッドの synchronized ブロック内で用いる事で行う。図 7 に、run() メソッドでの sync_write メソッド記述例を示す。

図 8 に、run() メソッドでのバーストライトの記述例を示す。

```
public void run() {
    Register other_r1 = (Register)super.access_registers.get(1);
    int  read_value[10]
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        /* バースト・リード（10回連続リード） */
        synchronized (this) {
            int i;
            for (i=0; i<10; i++) {
                read_value[i] =
                  other_r1.sync_burst_read(super.bc, i, 1));
                super.cc.consume_clock(1);
                this.do_something_wo_clock_boundary1();
            }
            other_r1.sync_burst_read(super.bc, i, 1);
            other_r1.endBurstAccess(super.bc, 1);
            this.do_something_wo_clock_boundary2();
        }
        this.do_something_w_or_wo_clock_boundary2();
    }
}
```

図 6　バーストリードメソッドの使用例

Fig. 6　Usage Example of Burst-Read Methods

```
public void run() {
  Register other_r0 = (Register)super.access_registers.get(0);
  int  write_value;
  while (true) {
    this.do_something_w_or_wo_clock_boundary1();
    other_r0.sync_write(super.bc, write_value, 1));
    this.do_something_w_or_wo_clock_boundary2();
}
```

図 7　ライトメソッドの使用例

Fig. 7　Usage Example of Write Methods

```
public void run() {
    Register other_r0 = (Register)super.access_registers.get(0);
    int  write_value[10]
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        /* バースト・ライト（10回連続ライト） */
        synchronized (this) {
            int i;
            for (i=0; i<10; i++) {
              other_r0.sync_burst_write(super.bc,
                                        write_value[i], 1));
              super.cc.consume_clock(1);
              this.do_something_wo_clock_boundary1();
            }
            other_r1.sync_burst_write(super.bc,
                                      write_value[i], 1);
            other_r1.endBurstAccess(super.bc, 1);
            this.do_something_wo_clock_boundary2();
        }
        this.do_something_w_or_wo_clock_boundary2();
    }
}
```

図 8　バーストライトメソッドの使用例

Fig. 8　Usage Example of Burst-Write Methods

バーストライトでは、バスのロックはコールする度に重ねて獲得し、書き込み処理終了後バスの解放を行わないで処理を終了する。そして、バーストライト回数のライト終了時に、重ねたロックを一気に解放する。この実装により、毎サイクル、ライト値が書き込めるバースト動作を実現している。

3.7　実装例の概略仕様

実装例として共有メモリ方式の 2 次元グラフィックス描画・表示システムの概略仕様を以下に示す。システム構成は図 9 のようになる。Java による記述例は [9] を参照されたい 各モジュールの仕様は以下の通りである。

（1）Command Interface　外部からの描画コマンドを受付

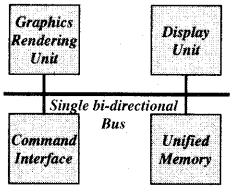

図 9　2 次元グラフィクス描画・表示システム
Fig. 9　Two-dimensional Graphics Rendering and Displaying System

け、バスを介して受け付けた描画コマンドをメモリへ転送する。
（2） Unified Memory　描画コマンド、描画ソースデータ、表示データを一元的に格納するメモリ。モデルを単純化する為、Java では配列で表現する。本デバイスはスレーブデバイスである。
（3） Graphics Rendering Unit　Unified Memory に描画コマンドがあるかをポーリングで調べ、存在する場合は、描画コマンド、描画データをバスを介してリードしながら描画処理を行い、描画結果を内部バッファに格納し、一気に Unified Memory へバースト転送する。モデルを単純化する為、描画コマンド、描画データ転送は配列への連続アクセスで実現する。
（4） Display Unit　表示データをリード後 1 ラインずつ表示を行う。モデルを単純化する為、垂直同期はモデル化しない。また、水平帰線期間はバスアクセスを行わないものとする。
（1）（3）（4）がマスタデバイスであり、（2）がスレーブデバイスである。マスタデバイスが同時にバス権獲得を行った場合の、バス権獲得の優先順位は下記である。

（4） ＞ （1） ＞ （3）

検証性質としては次のものを考える。

「表示用データ取得終了後に描画を開始し、次の表示開始までに描画を終了する事がある。但し、描画開始・終了にデータ転送サイクルは含まれる。」

上記制約を実時間時相論理 RPCTL (Real Time Parametric Computation Tree Logic) [6] で記述すると下記のようになる。

$$EF\{< display\_end > (AF(< render\_begin > true) \&\&\{AF(< render\_end > true)AU(< display\_begin > true)\})\}$$

4. Java 記述から中間表現への変換

4.1 アルゴリズム概要

まず、各 run() メソッドに対応する CFG(Control Flow Graph [10]) を生成する。特に、クロック同期メソッド（consume_clock）はクロック境界として識別し、呼び出しメソッドは呼び出し関係を表すノードを用いて表現する。各 run() メソッドに対応する CFG が生成されると、fork ノードを設け、そのノードから各 CFG の開始ノードへの fork 枝を付加する。fork ノードからの fork 枝による分岐は、分岐先の各制御フローを並列に実行することを表す。fork ノードを付加した CFG を C-CFG(Concurrent CFG) と呼ぶ (図 10 参照)。特に、バス上の各デバイスに run() メソッドが対応するものとして Java が記述されている事を前提とし、メモリデバイスに関しては、CFG 生成対象外である事の指定を与えるものとする。

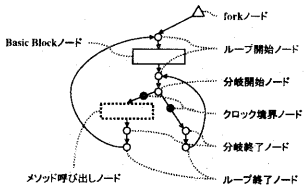

図 10　並行制御フローグラフ (C-CFG)
Fig. 10　Concurrent Control Flow Graph (C-CFG)

表 3　TNFA の記号の意味
Table 3　Explanation of TNFA Simbol

記号	意味
s, s'	状態
a	動作 (省略可能)
@?t	状態 s を訪れてから動作 a を実行するまでの経過時間を t に代入
$P(t)$	時間制約 (t に関する線形不等式の論理結合)

次いで、呼び出しメソッドの CFG を作成し、呼び出しメソッドが synchronized ブロック内に存在する場合は、呼び出しメソッド内の synchronized を CFG から削除する。特に、呼び出しメソッドが Register クラス内の通信メソッドである場合は CFG の作成を行わず、インスタンス関係からどのデバイス内のどのレジスタへのアクセスかと通信メソッドの名前だけを識別し、その情報を CFG に保持する。尚、通信メソッドが内部にクロック同期メソッドを含む場合は、その出力枝にクロック境界をマークする。更に、呼び出しメソッド全体のサイクル制約を導出する以外は、呼び出しメソッドをインライニングする。この例では、Rendering メソッド、Display メソッドはインライニングしない。
synchronized を予め定めた CFG に展開する。ここで、CFG はハード合成用とパラメトリック・モデルチェッキング用の 2 種作成する。

5. C-CFG から C-TNFA への変換

5.1 TNFA

TNFA(Timed Nondeterministic Finite Automaton) は時間オートマトンの一種であり、各遷移が以下の形式で表されるものである。

$$s \xrightarrow{a@?t[P(t)]} s'$$

ここで、各記号の意味は表 3 に示すとおりである。また、セマンティクスは「s から t 単位時間経過後に状態 s' に遷移。ただし、t は時間制約 $P(t)$ を満たす場合に限る」である。

5.2 C-TNFA

C-TNFA は複数の TNFA が並列動作するモデルであり、動

作 sync は高々 1 つの TNFA しか実行できないモデルである。特に連続する動作 sync に対しては他の TNFA が割り込めない。

5.3 アルゴリズム概要

synchronized ブロック内に存在しないクロック境界ノードにステート割り当て候補とし、また synchronized ブロック内であっても検証性質に必要となる信号が Basic Block として与えられている場合はその前後のクロック境界ノードにステートを割り当て候補とし、最後に synchronized ブロック境界をステート割り当て候補とする。得られたステート割り当て候補を重複が無いようにし、ステート割り当てを行い、各ステートからステートまでを DFS でトラバースする事で TNFA への変換を行う。得られた TNFA に対して、synchronized ブロックに対応する遷移を検出し、sync 遷移とする。連続する sync 遷移でない遷移は 1 つの遷移に纏める事でステート数の削減を行う。

6. C-TNFA から TNFA への変換

C-TNFA から積オートマトンを構成することにより動作の等価な単一の TNFA に変換する。変換アルゴリズムの詳細は文献 [5] を参照されたい。ここでは紙面の都合上、詳細説明を割愛する。

7. パラメトリック解析結果

文献 [6] に述べられているアルゴリズムを用いて、遷移を 2 回以上通らない制約で、探索深さ最大値 16 として、3.7 で示した実装例の Java 記述から変換した TNFA および検証性質の RPCTL 記述からパラメタ条件を導出すると、以下の条件を得る。

$$(0 <= kd \;\&\&\; 0 <= kr2 \;\&\&\; 0 <= kr1 \;\&\&\; 0 <= kb3 \;\&\&\; 0 <= kb2 \;\&\&\; 0 <= kb1 \;\&\&\; 0 <= kloop1 \;\&\&\; 0 <= k5 \;\&\&\; 4 <= k4 \;\&\&\; 0 <= k3 \;\&\&\; 0 <= k1 \;\&\&\; 0 <= k2 \;\&\&\; 12 <= kb1 + 9kb2 + kb3 \;\&\&\; 4 <= kl)$$

これと遷移を 2 回以上通らないという制約から得られたパラメータ条件 (求め方の詳細は文献 [5] 参照)

$$0 <= k1+k2+kloop1 <= 7 \;\&\&\; 5 <= kb1+kb2+kb3+kl <= 8 \;\&\&\; 7 <= kb1 + 3kb2 + kb3 + kl <= 11 \;\&\&\; 1 <= kl <= 11 \;\&\&\; 1 <= k1 + k2 + k3 + kloop1 + kl <= 6 \;\&\&\; 3 <= k4 <= 11 \;\&\&\; 5 <= kb1 + kb2 + kb3 + kl <= 11 \;\&\&\; 6 <= kb1 + 4kb2 + kb3 + 3kr1 <= 12 \;\&\&\; 1 <= 3kr2 + kl <= 8 \;\&\&\; 9 <= kb1 + 5kb2 + kb3 + kl <= 12 \;\&\&\; kr1 = kb2 - 1 \;\&\&\; 8 <= kb1+5kb2+kb3 <= 24 \;\&\&\; 1 <= kl+6kd <= 19$$

および、ハード合成の際に意味のある解を得るために、バスアクセス終了通知のサイクル数がそれぞれ $kb1 >= 2, kb2 >= 1, kb3 >= 1$ という制約との論理積を取って、得られたパラメータ条件に対して、目的関数を各パラメタ値の合計として、これを最小化するという線形計画問題をフリーソフト LP_SOLVE にて解かせると、以下の解を得る。

```
Value of objective function: 19
k1                  1
k2                  1
k3                  1
k4                  4
k5                  1
kd                  1
kl                  4
kloop1              1
kb1                 2
kb2                 1
kb3                 1
kr1                 0
kr2                 1
```

同時に Java 記述から抽象化を行わない C-CFG も抽出し、求された実時間制約を満たす C-CFG 記述を得ることができる。得られた C-CFG 記述からのハードウェア合成に関する詳細は文献 [5] を参照されたい。

8. あとがき

本論文では、単一バスシステムの Java によるモデル化およびパラメトリック・モデルチェッキングを用いた、実時間制約を満たすバスシステムの新規設計手法を提案した。

バスシステムの Java によるモデル化に関する今後の課題としては、単一双方向バスのみではなく、単方向バスを含むもの、ローカルバスを持つもの、またバス・ブリッジを介してバスが階層化されているような複数バスシステムを扱えるようにすることなどが考えられる。

文　献

[1] G. J. Holzmann: "The model checker SPIN", IEEE Trans. on Soft. Eng., **23**, 5, pp. 279–295 (1997).

[2] K. Havelund and T. Pressburger: "Model checking Java programs using Java PathFinder", Int. Journal on Software Tools for Technology Transfer, **2**, 4 (2000).

[3] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn and L. Hongjun: "Bandera: Extracting finite-state models from Java source code", Proc. 22nd Int. Conf. on Software Engineering (ICSE2000), ACM Press, pp. 439–448 (2000).

[4] D. Y. W. Park, U. Stern and D. L. Dill: "Java model checking", Proc. of the 1st Int'l Workshop on Automated Program Analysis, Testing and Verification (2000).

[5] 谷本, 北口, 中田, 東野 : "クロックサイクル制約を有する単一バスシステムの java によるモデル化およびパラメトリックモデル検査と高位合成技術を用いた設計手法の提案", 情報処理学会論文誌 (投稿中).

[6] A. Nakata and T. Higashino: "Deriving parameter conditions for periodic timed automata satisfying real-time temporal logic formulas", Proc. of 21st IFIP TC6/WG6.1 Int'l Conf. on Formal Techniques for Networked and Distributed Systems (FORTE2001), Kluwer Academic Publishers, pp. 151–166 (2001).

[7] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng and W. Visser: "Tool-supported program abstraction for finite-state verification", Technical report, Department of CIS, Kansas State University (2000).

[8] M. Berkelaar: "Lp_solve: Linear programming code", http://www.cs.sunysb.edu/ algorith/ implement/ lpsolve/ implement.shtml (1996).

[9] 北口 : "2 次元グラフィクス描画・表示システムの java 記述例", http://www-higashi.ist.osaka-u.ac.jp/~kitaguti/research/ index.html (2002).

[10] 中田 : "コンパイラの構成と最適化"、朝倉書店 (1999).