社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－SLDM－139 (7)
2009－ＥＭＢ－ 12 (7)
2009／3／5

システムレベル設計における並列動作の同期に関するデバッグ支援手法

原田 裕基[1] 西原 佑[2] 松本 剛史[3] 藤田 昌宏[3,4]

1 東京大学工学部電子工学科 〒113–8656 東京都文京区本郷 7–3–1
2 東京大学大学院工学系研究科電子工学専攻 〒113–8656 東京都文京区本郷 7–3–1
3 東京大学大規模集積システム設計教育研究センター 〒113–0032 東京都文京区弥生 2–11–16
4 科学技術振興機構 戦略的創造研究推進事業
E-mail: harada@cad.t.u-tokyo.ac.jp, tasuku@cad.t.u-tokyo.ac.jp
matsumoto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

あらまし 上位設計記述においては、並列動作や同期・通信を用いて設計が記述される。本研究では、同期設計の誤りによりデッドロック等の不具合がある設計のデバッグを支援する手法を提案する。具体的には、デッドロック等の不具合の原因になっている同期設計が、どのような条件下で同期されるか（または、デッドロックとなるか）を静的に解析し、その条件を出力する。SpecC で記述された並列エレベータコントローラによる実験を通して、提案手法により出力された条件によって、並列動作するプロセス間の同期が設計者の意図通りになっているかを確認することが出来ることを確認した。
キーワード システムレベル設計、同期検証、記号シミュレーション

Debugging Support for Synchronization of Parallel Execution in System Level Designs

Hiroki HARADA[1], Tasuku NISHIHARA[2], Takeshi MATSUMOTO[3], and Masahiro FUJITA[3,4]

1 Dept. of Electronics Engineering, Faculty of Engineering, The University of Tokyo
2 Dept. of Electronics Engineering, Graduate School of Engineering, The University of Tokyo
3 VLSI Design and Education Center, The University of Tokyo
4 Core Research for Evolutional Science and Technology, Japan Science and Technology Agency
E-mail: harada@cad.t.u-tokyo.ac.jp, tasuku@cad.t.u-tokyo.ac.jp
matsumoto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract There are many high-level designs contain parallel execution, synchronization, or communication, and they are often error-prone. In this work, we propose a debugging support method for designs with improper synchronization. We derive a safe condition by static analysis in which synchronization works properly under the target design. The experiment with an elevator controller containing parallel executions shows that we can easily check whether synchronization works as the designer intended or not, by using the derived condition.
Key words System level design, Synchronization verification, Symbolic simulation

1. はじめに

半導体の製造における設計生産性を向上させるためには設計期間の短縮が求められるため、抽象度の高い設計段階から設計を行われるようになってきている。抽象的な段階から設計を開始することにより設計者が記述する量を少なくすることが可能であるため、VLSI の設計期間全体を大幅に短縮することが出来るためである。抽象度の高い設計段階において検証を行えるようにすることは、手戻りのコストを減少させ設計期間を短縮するために非常に重要である。現在用いられている高位設計段階の一つであるシステムレベルにおいては、互いに並列動作するプロセス間で通信・同期を行うことが多い。従ってシステムレベル設計において同期に関する検証を行えるようにすることは重要である。本研究ではこの同期に関する検証を扱う。

この同期に関する検証手法としては、モデル検査等による形式的手法、静的チェックによる手法等が既に提案されている。

しかし、これらのほとんどの同期検証の手法ではどのような入力に対しても同期が行われることを前提としており、入力の値によって同期状況が変化する場合に対して正しくチェックが行えないという問題がある。つまり、特定の条件が満たされた場合にのみ同期が起こる場合等が考慮されていない。例えば、ユーザーによる入力が与えられるまで待機するシステム等は、ユーザーにより入力が与えられることがこの条件に当たる。実際にはこのような例は多く、この条件を考慮出来るようにすることは重要である。また、条件を指定出来るものであってもその条件を実際に与えることは難しい。この条件を与えるためには、ユーザーが設計を理解していることが求められるが、並列動作のモデルは複雑で理解が難しいためである。そこで本研究では、並列実行するプロセス同士が同期するための条件を求めることを目的とする。

システムレベル設計における同期は、同期がかかるのを待つ文 (wait 文) とそれを再開させる文 (notify 文) により表現される。すなわち、wait 文が実行されたプロセスは、対応する notify 文が実行されるまで、その実行を停止する。従って、ある wait 文に関してデッドロックが発生しないための条件は、その wait 文が実行されないか、実行されても対応する notify 文が必ず実行される条件に一致する。本研究では、指定された wait 文に対して、この条件を求める。

まず、システムレベル設計による記述をシステム依存グラフ (SDG) と呼ばれるデータ構造に変換する。次に、このグラフ上で静的に解析することにより、指定された wait 文に到達する実行パスを求める。このパスに関して記号シミュレーションを行うことにより、このパスを通るための条件 (パスコンディション) を求める。ここでは、この実行条件を A とする。同様に、対応する notify 文に到達する実行パスを求め、その実行条件を B とする。この場合、「A ⇒ B」という条件が、指定された wait 文が実行されないか、実行されても対応する notify 文が必ず実行される条件となる。

この条件は、条件を考慮した同期検証において、そのまま利用出来る。さらに、同期に関する設計誤りがあった場合には、この条件を求めることにより同期状況が設計者の意図通りになっているかどうかを確認することが出来る。このように、提案手法によって同期に関する設計誤りに対するデバッグを支援することも可能である。

本稿の構成は以下の通りである。まず、第 2 節でシステムレベル設計記述に対する同期検証手法の既存研究を紹介する。第 3 節で並列動作するプロセスが互いに同期するための条件を導出する手法を提案する。第 4 節で並列エレベータコントローラの例題に対する本研究の提案手法による実験結果を示し、第 5 節で結論を述べる。

2. システムレベル設計を対象とした同期検証手法

システムレベル記述に対する同期に関する検証の既存研究としては、Moy [1]、安藤 [2]、Sakunkonchak [3] による研究がある。

Moy らは、システムレベル設計記述言語の一つである SystemC [4] の記述を抽象化することで、同期検証等の検証を容易に行えるようにするツールである LusSy を開発した。SystemC に対しては LusSy により抽象化されたシステムレベル記述に対して本研究の手法を適用することで検証が行える。

安藤は、プロセスが止まったままになってしまうデッドロックと呼ばれるバグを、SDG を解析することで自動で検出する手法を提案した。しかし、この手法は待ち合わせ状態となったプロセスは無条件に再開されなければならないことを前提としており、同期するための条件を考慮していないため、正しく検出が行えない場合がある。この手法を用いる際、本研究で求めた条件をあらかじめ前提条件として与えておけば、最初から条件を考慮した検証を行うことができ、効率的に検証が行える。本研究では、この手法と同様に SDG を静的に解析することで同期するための条件を求める。

Sakunkonchak は、システムレベル設計記述言語の一つである SpecC [5] の記述の条件分岐を抽象化した状態でエラーに至るパスを求め、エラーに至るパスが実際に起こり得るかどうかを条件分岐を具体化して確認する手法を提案した。この手法では条件分岐を考慮しているものの、検証の最初の段階では無条件に同期が起こることを前提とするため、同期が起こらないパスの方が多い場合には効率的な手法とは言えない。この手法に関しても、本研究で求めた条件を最初から前提条件として与えておけば最初から条件を考慮した検証を行うことができ、より効率的に検証が行える。

3. 並列実行されるプロセスが互いに同期される条件の導出

この節では、並列実行されるプロセスが互いに同期されるための条件の導出手法を示す。3.1 節で対象とする設計記述言語である SpecC について、3.2 節で、本研究で実際に解析対象として扱う SDG について説明する。3.3 節で基本的な導出方法を示し、3.3 節では簡単な例題、3.5 節では少し複雑な例題について手法の適用の様子を示す。3.6 節、3.7 節、3.8 節、3.9 節では特殊な扱いが必要な設計についてその扱いを説明する。

3.1 対象とする設計記述言語

本研究では、システムレベル設計言語の一つである SpecC による記述を主な対象とする。ただし、同様の手法は他のシステムレベル設計言語にも適用出来る。SpecC は C 言語を拡張する形で作られており、あるまとまった動作をビヘイビアと呼ばれるクラスに格納することが出来る。このビヘイビアを互いに違う変数を入出力として持つインスタンス化することにより、同じ動作をするプロセスが容易に複数生成出来る。また、SpecC ではプロセスの並列動作を表現するために par 文を用いる。par{} の中に記述された文は全て互いに並列動作する。SpecC には同期を表現する機能も備わっている。これは wait 文、notify 文により表現され、イベント型変数 e に対して wait(e) を実行すると、並列実行されている他のビヘイビアで notify(e) が実行されるまで現在のビヘイビアの実行が中断される。つまり、wait(e) の後に書かれた記述が notify(e) が実

行された後に実行されることが保証される。

図 1 は簡単な SpecC 記述の例である。ビヘイビア Main では、ビヘイビア B1、B2 をインスタンス化して par 文により並列動作させている。また、wait 文、notify 文によりビヘイビア B1 の B の部分の記述はビヘイビア B2 の C の部分の記述より後に実行されることが保証される。

```
behavior B1{event e1}
  void main(){
    ・ ・ ・ //A
   wait(e1);
    ・ ・ ・ //B
  }
};

behavior B1{event e1}
  void main(){
    ・ ・ ・ //C
   notify(e1);
    ・ ・ ・ //D
  }
};

behavior Main(){
  event e1, e2;
  B1 b1(e1);
  B2 b2(e2);
  int main(){
    par{
      b1.main();
      b2.main();
    }
  }
};
```

図 1　SpecC 記述の例

3.2　解析に用いるデータ構造

さらに、設計を静的に解析するにあたり SpecC の記述をシステム依存グラフ (SDG) に変換する。SDG は、Horwitz [6] によって提案されたデータ構造で、設計記述中の文や表現をノードとして、依存関係をエッジとして表現したグラフである。SDG ではエッジで表現される主要な依存関係はデータ依存、制御依存、宣言依存の 3 つだが、Krinke [7] は、並列性を考慮し通信依存と干渉依存を表現するエッジを新たに導入した SDG を提案した。本研究ではこれにさらに制御フローエッジを加えた ExSDG [8] を用いて検証を行う。ここで、エッジの中で本研究で特に用いるものとして通信依存エッジ及び制御フローエッジについて説明する。通信依存エッジはイベント変数等を通してプロセス間の通信が行われることを示している。例えば SpecC においては notify ノードから wait ノードに向けてエッジが引かれる。制御フローエッジは実行文の実行順を示している。あるプロセスにおいて文 s1 の次に s2 が実行される可能性がある場合に s1 のノードから s2 のノードに向けてエッジが引かれる。

図 2 は、図 1 の記述を SDG の形式に変換したものの一部を抽象化して表したものである。実線は制御フローエッジ、破線は通信依存エッジを表す。

3.3　基本的な導出方法

ここでは、あるビヘイビアがある wait 文で実行が止まらないか、止まっても確実に実行が再開されるような条件を求めることを目的とする。ここでは簡単のため、SpecC による記述が対象であるとする。なお、あらかじめ SDG 上で、インスタンス化されたビヘイビア毎にノードを作っておく。つまり、同じ wait ノード (又は notify ノード) に別々のイベント変数が代入されていても、互いに区別出来るようにしておく。

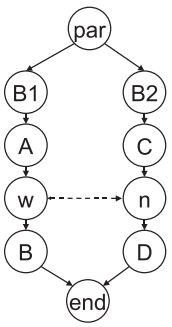

図 2　SDG の例

ある wait ノードに対して、その wait ノードを含むプロセスが、指定した wait ノードで止まらないか、止まっても実行が確実に再開されるような条件を求める手順は以下の通りである。

（1） 与えられた設計記述に対し、チェックの対象とする wait ノードを一つ指定する。この wait ノードを w とする。

（2） SDG 上で w から制御フローエッジを par ノードが見つかるまで辿る。なお、wait 文、notify 文は並列実行されるビヘイビアが存在するビヘイビアの内部でのみ使用されるため、par ノードは必ず見つかる。見つかったこの par ノードを pw とする。

（3） SDG 上で w から通信依存エッジを辿り、wait ノードに対応する notify ノードをリストアップする。これらの notify ノードをそれぞれ n1、n2 ••• とする。それぞれの notify ノードから制御フローエッジを par ノードまで辿る。これらの par ノードをそれぞれ pn1、pn2 ••• とする。

（4） pw、pn1、pn2 ••• を互いに比較し、同じノードを指しているものが存在した場合はこれらをまとめ、par ノード群をそれぞれ p1、p2 ••• と置き直す。なお、全てが同じノードを指していた場合はこれを p とする。

（5） p1、p2 ••• からそれぞれ制御フローエッジをプログラムのトップノードまで辿る。p1 からトップノードまで辿る際、途中にあった par ノードの集合を P1 とする。同様に P2、P3 ••• を求める。

（6） P1、P2 ••• の全てに共通する par ノードのうち、プログラムのトップノードから最も遠いもの求め、これを p とする。

これまでの操作で、w、n1、n2 ••• のどれから制御フローエッジを辿っても辿り着くことの出来る par ノード p が求まった。ここからは、この p を始点として wait が同期されるための条件を求める。

（7） p から w に行き着くまでのパスを求める。

（8） このパスに対して記号シミュレーションを行うことにより、その wait ノードに行き着くための実行条件を求める。得

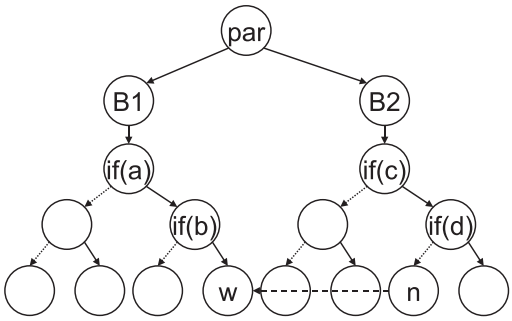

図 3　簡単な同期検証の例題

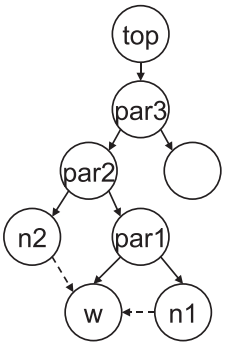

図 4　par 文の構造が入れ子状になっている例題

られたこの条件を A とする。

（9） 同様にして p から n1、n2 ··· に行き着くための実行条件をそれぞれ求める。得られたこれらそれぞれの notify ノードに行き着くための実行条件の論理和を取ったものを B とする。

（10） 以上により求められた A と B を用いて、「A ⇒ B」という条件を求める。この条件は、指定した wait ノードを含むプロセスが、指定した wait ノードで止まらないか、止まっても実行が確実に再開されるような条件に一致する。

3.4　簡単な例題

ここでは、簡単な例題を用いて具体的に説明する。

図 3 は、あるプログラムの SDG の一部を抽象化して表したものである。図のノードはプログラム上のある文を表す。ビヘイビア B1 とビヘイビア B2 は並列に動作している。if 文のノードの中の文字は条件文を表す。if 文のノードからは二本のエッジが出ており、実線は if 文が true であった場合の、点線は false であった場合のフローを表す。例えば a = true であれば、if(a) のノードから右のエッジを辿って実行が進む。破線は通信依存エッジで、w のノードは wait 文を、n のノードは notify 文を表す。

このプログラムには wait 文は一つしかないので、この wait 文に対しチェックを行う。まず、w から制御フローエッジを辿って図 3 の par のノードまでさかのぼる。次に、w から通信依存エッジを辿ると、対応する notify ノード n が見つかる。この n から制御フローエッジを辿って同じく par ノードまでさかのぼる。二つの par ノードは一致するので、図 3 の par のノードを p とする。par(=p) から w に行き着くための実行条件を求めると、$a \wedge b$ となる。同様に par から n に行き着くための実行条件を求めると、$c \wedge \neg\, d$ となる。以上より、$a \wedge b \Rightarrow c \wedge \neg\, d$ という条件が求める条件である。さらにこの式は、

$$\begin{aligned} a \wedge b \Rightarrow c \wedge \neg\, d &= \neg\, (a \wedge b) \vee (c \wedge \neg\, d) \\ &= (\neg\, a \vee \neg\, b) \vee (c \wedge \neg\, d) \\ &= \neg\, a \vee \neg\, b \vee (c \wedge \neg\, d) \end{aligned}$$

と簡単化出来る。以上より、$\neg\, a \vee \neg\, b \vee (c \wedge \neg\, d)$ が、B1 が止まらないか、止まっても確実に再開されるための条件である。

3.5　par 文の構造が入れ子状になっている場合

図 4 のように、par 文の構造が複雑な例題について説明する。ここで、top はプログラムのトップノードを表す。まず、w から制御フローエッジを辿ると par1 が見つかるので、par1 を pw とする。w から通信依存エッジを辿ると、対応する notify ノード n1、n2 が見つかる。n1 から制御フローエッジを辿ると par1 が見つかるので、par1 を pn1 とする。同様に、n2 から制御フローエッジを辿ると par2 が見つかるので、par2 を pn2 とする。pw、pn1、pn2 を互いに比較すると pw=pn1 であることが分かるので、pw=pn1=p1、pn2=p2 とする。p1、p2 からそれぞれ制御フローエッジをトップノードまで辿り、途中で見つかった par ノードの集合を P1、P2 とする。p1 の場合 par1、par2、par3 が見つかるので、P1(pa1,par2,par3) となる。同様に P2(par2,par3) である。P1、P2 に共通する par ノードは par2 と par3 だが、par2 がよりトップノードから遠いので、p=par2 とする。以上により実行条件を求める際の始点となる par ノード p が求まった。以降は図 3 の例と同様に実行条件を求めていく。

3.6　wait ノード、notify ノードの上流に他の wait ノードが存在する場合

実行条件を求める際、そのパス上に他の wait ノードが存在した場合、この上流の wait ノードの同期条件を求める必要がある。この上流の wait ノードを w' とする。w' から通信依存エッジを辿り対応する notify ノードの集合を求め、それぞれ n'1、n'2 ··· とする。n'1、n'2 ··· からそれぞれ制御フローエッジをプログラムのトップノードまで辿る。トップノードから n'1、n'2 ··· へ辿り着くための条件をそれぞれ求め、これらの条件の論理和を B' とする。wait ノードを無視した状態の実行条件と B' の論理積が、この wait ノードを考慮した場合の実行条件となる。

図 5 の例では wait ノード w1 を対象とすることにする。まず、w2、w3 を無視した状態で考えると、p=par となる。w2 を無視した場合の、p から w1 へ辿り着くための条件を求めると ¬ a となる。ここで、w2 に対応する notify ノードが実行されるための条件を求める。w2 から通信依存エッジを辿ると対応する notify ノード n2 が見つかる。n2 が実行されるための条件を求めると c となる。よって、w1 が実行される条件は $\neg\, a \wedge c$ である。次に、w1 から通信依存エッジを辿ると、n1

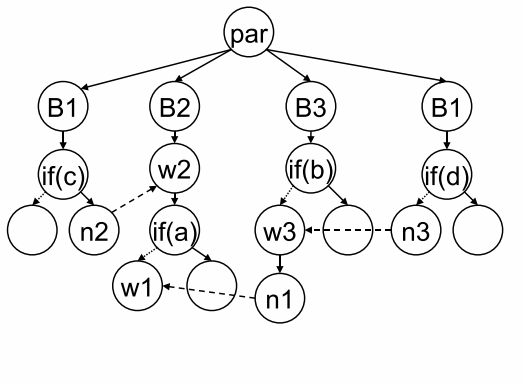

図 5　wait ノード、notify ノードの上流に別の wait ノードが存在する例

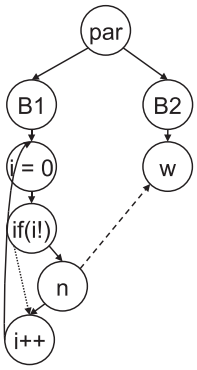

図 6　ループ文を含む例題

```
behavior B1{
    waitfor(1);
    notify(e1);
};

behavior B2{
    wait(e1);
    ・ ・ ・ //A
};
```

図 7　時間経過を含む例題

が見つかる。w3 を無視した場合の n1 が実行されるための条件は¬ b である。ここで、w3 に対応する notify ノードが実行されるための条件を求める。w3 から通信依存エッジを辿ると対応する notify ノード n3 が見つかる。n3 が実行されるための条件を求めると ¬ d となる。よって、n1 が実行される条件は ¬ $b \wedge \neg\, d$ である。以上より、¬ $a \wedge c \Rightarrow \neg\, b \wedge \neg\, d$ という条件が求める条件である。図 3 の例と同様に簡単化すると、$a \vee \neg\, c \vee (\neg\, b \wedge \neg\, d)$ となり、これが B2 が w1 によっては止まらないか、止まっても確実に再開されるための条件である。

3.7　ループ文の扱い

ループ文の先頭 (for のノード、while のノード等) へは、ループの終点のノードとループ文の一つ上の実行文のノードからの二つの制御フローエッジが伸びている。このうち、ループの終点のノードへ戻るエッジを辿ると、実行が無限にループする場合やループ回数が入力によって変わる場合等に問題が生じる。パスを求める際に、実行が何回繰り返されるか決まっていないと実行条件を求めるべきパスが静的に決定されないためである。従ってここではループの終点のノードへ戻るエッジは無視することにし、ループ文の一つ上の実行文のノードにのみ戻ることにする。ところが、このように仮定すると図 6 のような例題を扱う際に問題が生じる。

ここで、i は int 型の変数である。この例では、i=0 の時に notify が実行される。ループに入る前に i=0 としてあるので、1 周目の実行の際に必ず notify は実行される。つまりこの例では、wait が実行されないか、実行されても確実に再開されるための条件は常に true である。しかし、本研究のアルゴリズムで条件を求めると i == 0 という結果が得られてしまう。このような条件を出力してしまうと、特定の条件下でしか同期が起こらないと誤解されてしまう恐れがあるので、適切な結果が得られているとは言えない。このように、ループ文は扱えるものの、実際よりも厳しい条件を出力してしまう場合が存在する。

3.8　時間経過を含む設計について

次に、図 7 のような例題を考えてみる。ここで waitfor(1) は、1 単位時間経過するという意味の記述である。つまり、B2 の A の部分の記述はプログラムの実行開始から 1 単位時間後に実行開始される。この例では本研究のアルゴリズムにより条件を求めると常に true という条件が得られる。しかしこの条件では、プログラムの実行開始から 1 単位時間後に実行開始されるという仕様が理解できない。このように、条件は求まるものの実際に検証/デバッグの支援としては不十分な条件しか求まらないという例が存在する。

3.9　部分的な検証を行うことによる制約

図 8 のように、実行条件を求める際の始点とする par ノード (=p) の上流に条件分岐が存在した場合、厳密な条件を求めるためには実行条件を求める前にこの上流の条件分岐を考慮する必要がある。具体的には、実行条件を求める前に、プログラムのトップノードから p に辿り着くための条件を求め、これを前提条件として与えた上で実行条件を求める必要がある。

しかし、提案手法では p の上流の条件分岐を無視しているため、p の上流に条件分岐があった場合は厳密な条件よりも厳しい条件が求まってしまう。例えば、p 以下が実行されない時は wait 文が実行されないため、同期するための条件は true であるが、提案手法ではそのような場合でも wait 文が実行されることを仮定して条件を求めてしまう。

しかしながら、提案手法ではこのような部分的な解析を行っているため、計算量を抑え、より大規模な設計に対応出来る可能性がある。つまり、p に到達する条件をどの程度正確に考慮するかということと、計算量の増大はトレードオフの関係にあり、手法のスケーラビリティを犠牲にする代わりに条件をより正確に求める手法も考えられる。

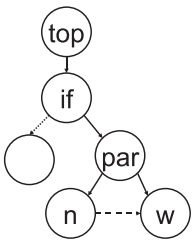

図 8　par ノードの上流に条件分岐がある場合

4. 実　　験

4.1 対象とする例題

本節では、実験の対象として並列エレベータコントローラの例題を用いる。これは 10 階建てのビルで 3 台のエレベータが並列に運用されることを想定したものであり、SpecC 記述で約 2000 行程度である。2〜9 階にそれぞれ「上」ボタンと「下」ボタンが、1 階には「上」ボタンのみ、10 階には「下」ボタンのみがそれぞれある。3 台のエレベータの中にはそれぞれ行き先の階を指定する「1」〜「10」のボタンが存在している。

プログラムは 120 個程度の互いに並列動作するビヘイビアで構成されている。各々のビヘイビアの内部の実行は無限にループしている。そのうち一つが入力を受け付けるビヘイビアであり、ユーザーが入力を与えるまで実行が進むことはない。入力が与えられると notify 文が実行される。その際用いられるイベント変数の値は入力の内容によって変わる。その他のそれぞれのトップノードの真下には wait 文がある。つまり、入力が与えられるまでは全てのビヘイビアが実行を停止しており、入力を与えられると入力の内容に応じて特定のビヘイビアが実行を再開する。プログラムにはその機能により分けて 6 種類の wait ノードがある。本研究では同じ wait ノード (又は notify ノード) に別々のイベント変数が代入されていても互いに区別出来ると仮定している。これを考慮に入れると例題の内部には約 70 個の wait ノードと notify ノードが存在していることになる。

4.2 実 験 結 果

プログラム内に存在する 6 種類の wait ノードのうち、4 種類については本研究のアルゴリズムを用いることで、指定した wait を含むビヘイビアが指定した wait で止まらないか、止まっても実行が確実に再開されるような条件を適切に求められる。

残り 2 種類のうち 1 種類は、条件は求まるものの、実際にはその条件では不十分である。これは、時間経過による自動的な値の変化を考慮出来ないためである。一度入力を与えると、一部のビヘイビアは特定の条件が満たされるまで 1 単位時間ごとに 1 周実行される。例えば、全てのエレベータが 1 階にある時に 10 階のボタンを押すと、どれか一つのエレベータが 10 階に辿り着くまでは入力が与えられずとも実行が進む。これは、ループの始点からループの終点に戻るパスを考慮しないことによる発生する問題である。

残りの 1 種類についてもやはり条件は求まる。この wait ノードは競合アクセスを防止する目的で用意されている。従って条件分岐に関係無い内部変数を考慮に入れて競合アクセスが発生しないかどうかを確認する必要があり、この意味で十分な結果を得られているとは言えない。

5. ま と め

本研究では、ある wait 文が実行されないか、実行されても対応する notify 文が必ず実行されるための条件の導出手法を提案した。

まずシステムレベル設計記述言語の一つである SpecC の記述を SDG と呼ばれるグラフ構造に変換する。これを解析することで並列実行されるプロセスが互いに同期するために通るべきパスを求める。このパスについて記号シミュレーションを行うことでそれぞれのパスコンディションを求め、求まったこれらの条件を用いてあるプロセスがあるパスで止まらないか、止まっても確実に実行が再開されるような条件を求める。

この条件を用いることにより条件を考慮した同期検証の支援が行えるようになり、より多くの設計の同期検証が行えるようになる。また、同期に関するバグがあった場合に、実際の設計における同期するための条件を提案手法により調べることでそのバグが起こる原因を確認し、バグを修正する支援を行うことも可能になる。

しかし、時間経過やループの回数を考慮に入れることが難しいといった問題があり、今後はこれらの問題を解決する新たな手法の提案を目指したい。また、これらのアルゴリズムによるチェッカの実装も同時に目標としたい。

文　　献

[1] M.Moy, F.Maraninchi, “LusSy: A Toolbox for the Analysis of System-on-a-Chip at the Transactional Level” In *Proc. of the International Conference on Application of Concurrency to System Design*, pages 26-35, June 2005.

[2] 安藤大介, 松本剛史, 西原佑, 藤田昌宏, “システムレベル設計に対する拡張システム依存グラフを利用した記述チェッカ,” 電子情報通信学会技術研究報告 Volume 106, Number 547, pages 37-42, March 2007

[3] T. Sakunkonchak, S. Komatsu, and M. Fujita, “Synchronization verification in system-level design with ilp solvers,” In *Proc. of the Formal Methods and Models for Co-Design*, pages 121.130, July 2005

[4] SystemC, http://www.systemc.org/.

[5] SpecC, http://www.cecs.uci.edu/ specc/ .

[6] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” In *Proc. ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 12, Issue 1, pages 26-60, January 1990.

[7] J. Krinke, “Advanced Slicing of Sequential and Concurrent Programs,” In *Proc. of the 20th IEEE International Conference on Software Maintenance (ICSM ’04)*, pages 464-468, September 2004.

[8] T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita, “ExSDG : Unified Dependence Graph Representation of Hardware Design from System Level down to RTL for Formal Analysis and Verification,” In *Proc. of International Workshop on Logic and Synthesis 2007*, pages 83-90, May 2007.