非同期システムにおける時相論理式の帰納的導出法

Inductive Description Method for Temporal Formulas on Asynchronous Systems

長田 康敬
Yasunori Nagata

琉球大学 工学部 電気電子工学科
University of the Ryukyus Dept. of Elc'l & Electronics Eng.
E-mail: ngt@eee.u-ryukyu.ac.jp

Abstract: Recently design verification have been played an important role in the design of large scale and complex systems. In this article, we especially focus on model checking methods. Behaviors of modeled systems are generally specified by temporal formulas of computation tree logic. However, users must know well temporal specification because the specification might be complex. We proposed method that temporal formulas are gained inductively and amounts of memory and time are reduced. Finally, we will show verification results using our proposed method.

1 はじめに

近年, 設計検証は高集積かつ複雑なシステムの設計において重要な役割を果たすようになってきている. その検証は, 設計されたシステムが正しく動作するかどうかの検査を行う. また, いくつかの検証手法が数多く研究されてきた [1][2][3][4].

本稿では, 特に形式的検証手法におけるモデル・チェッキングに着目し, その仕様記述に関して考察する. モデル・チェッキングでは, 所望の回路またはシステムをモデリングし, 検証において重要でなく, かつ関係のない動作を省略する. まずはじめに設計者は, どの階層 (アーキテクチャ, RTL, またはゲート) レベルで検証するかを決定する. 本研究では, 非同期ハンドシェーク回路を主に考察するため, 信号遷移グラフ (STG : Signal Transition Graph) で回路またはシステムをモデリングする. そして, モデリングされたシステムの動作は, 一般に CTL(Computation Tree Logic) の時相論理式により記述される. 最終的には, モデリングされたシステムが, その時相論理式で記述された動作を満たすかどうかを検証する.

時相論理式で動作仕様を記述する際, その動作が複雑になり過ぎると記述が複雑になるため, 設計者は時相論理記述を熟知しなければならない. そこで本稿では, 特にモデル・チェッキングの記述における時相論理式を帰納的に導出する手法を提案する. また, 得られた記述には, strong/weak temporal order の概念も含まれており, そのため入出力関係のみならず, 複数の入力間の時間的な順序関係も表現できることを示す. 最後に, その検証結果を示し, 本手法の有効性を議論する.

2 準備

2.1 モデル・チェッキング

システムを形式的に検証する際, モデル検査 (Model Checking) 手法が広範囲に用いられている. この Model Checking の過程 (Process) は,

- モデル・チェッキングツールが処理できるようにモデリングする. このとき抽象化 (Abstraction) をし, 処理に無関係な, または重要でないタスク (task) を省く.
- モデリングしたシステムがどのような振舞いをするかを記述 (Specification) する. 記述では主に時相論理式を用いる.
- 記述したシステムのプロパティを自動的に検証 (Verification) する.

上記の過程でモデリングし記述したシステムが仕様を満たすかどうかを自動的に検証する [1].

また, 検証では, 主に次のプロパティを検査する.

Safety property

Safety property(安全性) は, 回路が要求しない出力を生成しないことを意味する.

Reachability property

Reachability property(可達性) は, (ペトリネットにおいて) マーキング M_0 からマーキング M_n へ変換する発火の系列が存在するとき, マーキング M_n はマーキング M_0 から可達であるという性質を意味する. この可達問題は, 決定可能であることが示されているが, 一般的に, その検証に少なくとも指数関数的な空間 (時間) を必要とする. また, Reachability と (後述する)Liveness は等価 (一方が決定可能であれば他方も決定可能) であることも示されている. さらに, プレースの数の等しい任意の二つのペトリネット N と N' に対して, それらが同値 (equality) であるかどうかは決定不可能である (決定するアルゴリズムは存在しない)[11][12].

Liveness property

回路の正しさを保証するための性質一つである. これは, 回路が期待される出力を必ず生成するという性質である. 同期式回路では任意の出力の最大遅延が予測可能なので, liveness を満たすかどうかを調べるためには, 最大遅延時間内での期待される出力の有無を調べるだけでよい. しかし, 非同期式回路の遅延モデルの中には, 遅延時間は "有限だがその上限値は未知とする" ものもあり, そのようなモデルで設計された非同期式回路が liveness を満たすかどうかは, 同期式とは異なる手法を用いて調べる必要がある. また, クロック単位で状態を調べればよい同期式に対して, 信号遷移の順序を考慮する必要のある非同期式では, 本質的に調べる必要のある状態数は増大するため, 従来のテスト方法では liveness を保証するには十分ではない [10].

Fairness property

Liveness と似た概念であり, 入力遷移と出力遷移の区別しない点が異なる [10].

Deadlock-freedom

回路が期待される出力を生成しないときシステムの動作としては, 出力が期待されるが一切の遷移が停止してしまう場合を Deadlock という. それに対し, 期待される出力が生じないまま無限に遷移し続ける場合を Livelock という [10].

これらは [6] で詳しく述べられている. ちなみに, SMV[5] は, safety, liveness, fairness, deadlock freedom を検証できる (記述は, CTL[1]). また, SPIN[6][9] は safety, liveness を検証できる (記述は, LTL[2]).

2.2 Computation Tree Logic

回路またはシステムを検証する際, CTL(Computation Tree Logic) により, その動作仕様を記述するのが一般的になってきている. CTL は時相論理における分岐時相論理の一つである. CTL のオペレータは, **G**(*globally*), **F**(*sometime in the future*), **X**(*next time*) を用い過去を表すオペレータは仕様しない. また, CTL では, **A**(for all computation paths), **E**(some computation path) の path quantifier を用い分岐を表現する. 例えば, **AG** p は, "全てのパスにおいて, 常に p が真であるなら, **AG** p は真である" ということを示す.

3 提案する手法

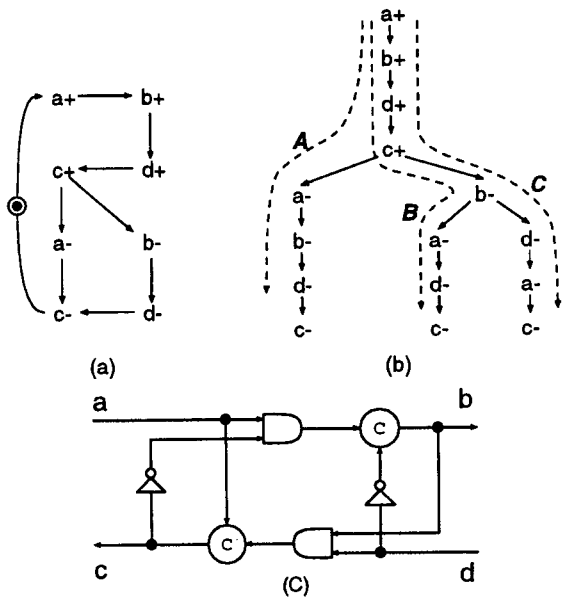

図 1: 非同期ハンドシェーク回路: (a) 信号遷移グラフ, (b) 分岐表現, (c) 検査する回路 ('+' は立上り, '-' は立ち下がり)

ここでは, モデルチェッキングにおける時相論理式を

[1] Computation Tree Logic
[2] Linear Temporal Logic

帰納的に導出する手法を示す. 特に, 図.1 に示すような非同期ハンドシェーク回路について考察する. 記述手順を以下に示す.

[記述手順]

[1.] 全てのパスを抽出する (図.1(b)).

(A) a+ b+ d+ c+ a- b- d- c-
(B) a+ b+ d+ c+ b- a- d- c-
(C) a+ b+ d+ c+ b- d- a- c-

[2.] 各々のパスに対し, 入力-出力関係を抽出する. まず, あるイベント (信号線) の戻り値 (a+ → a-) が検出されなければ, そのイベントの入出力関係の探索を終了し, 他のイベントの入力-出力関係を探索する. ここで, 全信号線の初期値は "0", すなわち "-" とする.

(A) {(a+ , b+), (a+ , c+), (d+ , c+), (d+ , b-), (a- , b-), (a- , c-), (d- , c-), (d- , b+)}

ここで, a+ と d+ を比較する. a+ の後の出力信号 (successor) は, b+ と c+ であり, d+ の successor は c+ と b- である. しかし, d+ の successor b- は, 次のサイクルを示しているので, a+ と d+ では, a+ のイベントの後に d+ のイベントが起こることがわかる. ここで, a+ と d+ のような入力間の順序関係のことを *weak temporal order relation* とする.

(B) {(a+ , b+), (a+ , c+), (d+ , c+), (d+ , b-), (a- , c-), (d- , c-), (d- , b+)}

(C) {(a+ , b+), (a+ , c+), (d+ , c+), (d+ , b-), (a- , c-), (d- , c-), (d- , b+)}

ここで (B) と (C) のパスは同一のものであることがわかる. よって入力-出力関係としては, 等価なパスであるということになる. また, (A) と (B) のパスを比較する. (A) では (a- , b-) という入力-出力関係があるが,(B) では, その入出力関係はない. しかし (d+ , b-) として, イベント b- は起こることが示されている. これは出力イベント b- の後に, a- が起こることを示している. このように異なるパスどうしで, 入力-出力関係が逆になる場合, これを *strong temporal order relation* とする.

[3.] 入力-出力関係に, 時間的なオペレータを導入する. パス (A) において, (a+ , b+) という入力-出力関係は, b+ が a+ の直後の successor となるので **AX**(a+ , b+) とする. ここで, 時相オペレータ **AX** を用いるのは, 一つのパスのみに着目しているためである. また, 入力-出力関係 (a+ , c+) において, c+ は a+ の successor であるが, 直後の successor ではないので, **AF**(a+ , c+) とする. 同様に, 全ての入力-出力関係に時相オペレータを導入すると以下のようになる.

(A) {**AX**(a+ , b+), **AF**(a+ , c+), **AX**(d+ , c+), **AF**(d+ , c-), **AX**(a- , b-), **AF**(a- , c-), **AX**(d- , c-), **AF**(d- , c+)}

(B) {**AX**(a+ , b+), **AF**(a+ , c+), **AX**(d+ , c+), **AF**(d+ , b-), **AF**(a- , c-), **AX**(d- , c-), **AF**(d- , b+)}

(C) {**AX**(a+ , b+), **AF**(a+ , c+), **AX**(d+ , c+), **AF**(d+ , b-), **AX**(a- , c-), **AF**(d- , c-), **AF**(d- , b+)}

これにより, 手順 [2.] でパス (B) とパス (C) は区別されなかったが, 時相オペレータを導入することで, 区別することができる.

[4.] 全てのパスを時相論理式で記述していく. まず, 全てのパスで時相オペレータと入力-出力関係がともに同一のものを抽出していく.

{**AX**(a+ , b+), **AF**(a+ , c+), **AX**(d+ , c+), **AF**(d+ , b-), **AF**(d- , b+)}

これらの関係は, 全てのパスで常に成立する. よって, 時相オペレータは以下のように **AG** を用いる.

AG[**AX**(a+ , b+) ∨ **AF**(a+ , c+) ∨ **AX**(d+ , c+) ∨ **AF**(d+ , b-) ∨ **AF**(d- , b+)]

また, **AF** は "結局, 最後には真になる" ことを表現するので, 次の時点を表現する *next* オペレータ **AX** は, **AX** ⊆ **AF** となり, **AF** に包含される. よって, パス (*B*) と (*C*) における入出力関係 (a- , c-) と (d- , c-) に対し, 時相論理式は以下のように記述される.

AG[**AF**(a- , c-) ∨ **AF**(d- , c-)]

したがって,

AG[**AX**(a+ , b+) ∨ **AF**(a+ , c+) ∨ **AX**(d+ , c+) ∨ **AF**(d+ , b-) ∨ **AF**(a- , c-) ∨ **AF**(d- , c-) ∨ **AF**(d- , b+)].

となる.

[5.] ここで, **AX**(a+ , c+) と **AF**(d+ , c+) は **AF**(a+ ∧ d+ , c+) のように結合できる. それは, トランジション c+ は a+ と d+ が起こった次の時点で起きるためである. したがって, このように縮約していくと以下のような記述となる.

AG[**AX**(a+ ∧ d- , b+) ∨ **AF**(a+ ∧ d+ , c+) ∨ **AF**(d+ , b-) ∨ **AF**(d+ , b-) ∨ **AF**(a- , c-) ∨ **AF**(d- , c-)]

したがって, この記述が liveness を表現する記述となり, 帰納的に導出することができる.

4 記述例

ここでは, 前節で述べた導出法を用い, 図.2 に示すような非同期パイプラインの時相論理式を記述していく. 図.2(middle) は `ctrl` モジュールだけの信号遷移グラフを示している. それは, 非同期システムにおいてはこの `ctrl` モジュールが信号伝播において重要な役割を果たす仲裁モジュールであるからである. まずはじめに, 提案する手法を用いずに時相論理式を記述していく.

[導出法を用いない場合]

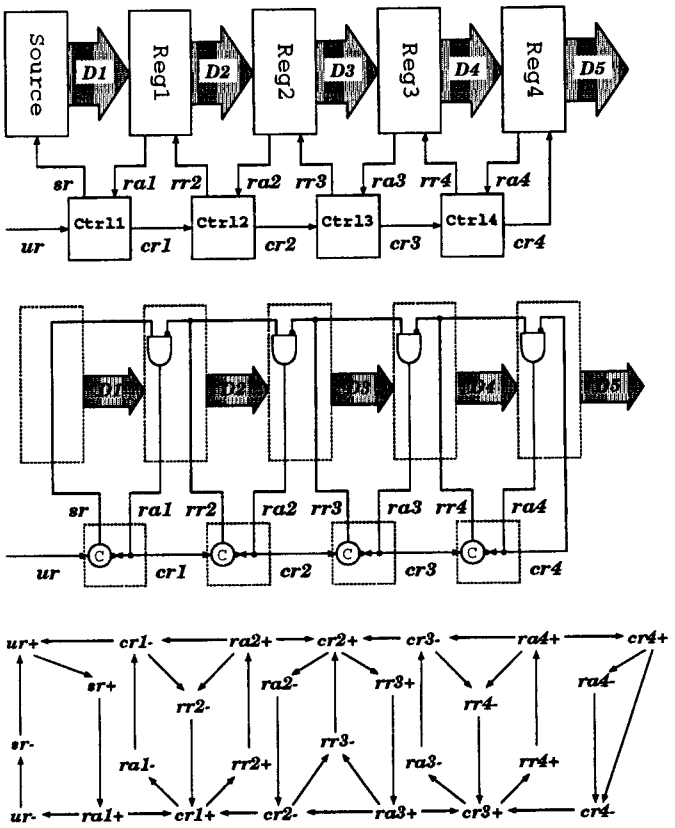

図 2: 非同期パイプライン: (upper) RTL レベル, (middle) 内部構造, (lower) 信号遷移グラフ.

[ctrl1] **AG**[**AX**(ur+ , sr+) ∨ **AF**(ur+ ∧ sr+ , ra1+) ∨ **AF**(ur+ ∧ sr+ ∧ ra1+ , cr1+ ∧ ur1-) ∨ **AF**(ra1+ , sr-) ∨ **AF**(cr1+ , ra1-) ∨ **AX**(ra1- , cr1-) ∨ **AF**(sr- ∧ cr1- , ur+)]

[ctrl2] **AG**[**AX**(cr1+ ∧ cr2- , rr2+) ∨ **AF**(cr1+ ∧ rr2+ , ra2+) ∨ **AX**(cr1+ ∧ rr2+ ∧ ra2+ , cr2+) ∨ **AF**(cr2+ , ra2+) ∨ **AF**(rr2+ ∧ ra2+ , cr1-) ∨ **AX**(ra2- ∧ cr1- , rr2-) ∨ **AF**(cr2- , cr1+)]

[ctrl3] **AG**[**AX**(cr2+ ∧ cr3- , rr3+) ∨ **AF**(cr2+ ∧ rr3+ , ra3+) ∨ **AX**(cr2+ ∧ rr3+ ∧ ra3+ , cr3+) ∨ **AF**(rr3+ ∧ ra3+ , cr2-) ∨ **AF**(ra3+ ∧ cr2- , rr3-) ∨ **AF**(cr3+ , ra3-) ∨ **AX**(ra3- , cr3-) ∨ **AF**(rr3- ∧ cr3- , cr2+)]

[ctrl4] **AG**[**AX**(cr3+ ∧ cr4- , rr4+) ∨ **AF**(cr3+ ∧ rr4+ , ra4+) ∨ **AX**(cr3+ ∧ rr4+ ∧ ra4+ , cr4+) ∨ **AF**(rr4+ ∧ ra4+ , cr3-) ∨ **AF**(ra4+ ∧ cr3- , rr4-) ∨ **AF**(cr4+ , ra4-) ∨ **AX**(ra4- , cr4-) ∨ **AF**(rr4-

∧ cr4- , cr3+)]

この記述は, 図.2 における ctrl モジュールの全ての振る舞いを記述した結果となっている. それは, 入力-出力関係だけでなく, (*sr+,ra1+*) のような出力-入力関係も考慮した記述となっているためである. 次に, 導出法を用いた時相論理式を以下で示していく.

[導出法を用いた場合]

[ctrl1] **AG**[**AX**(ur+ ∧ ra1- , sr+) ∨ **AX**(ur+ ∧ ra1+ , cr1+) ∨ **AX**(ur- , sr-) ∨ **AX**(ra1- , cr1-)]

[ctrl2] **AG**[**AX**(cr1+ ∧ ra2- , rr2+) ∨ **AX**(cr1+ ∧ ra2+ , cr2+) ∨ **AX**(cr1- , rr2-) ∨ **AX**(ra2- , cr2-)]

[ctrl3] **AG**[**AX**(cr2+ ∧ ra3- , rr3+) ∨ **AX**(cr2+ ∧ ra3+ , cr3+) ∨ **AX**(cr2- , rr3-) ∨ **AX**(ra3- , cr3-)]

[ctrl4] **AG**[**AX**(cr3+ ∧ ra4- , rr4+) ∨ **AX**(cr3+ ∧ ra4+ , cr4+) ∨ **AX**(cr3- , rr4-) ∨ **AX**(ra4- , cr4-)]

これらの時相論理式は, 提案する手法を用いることで, 入力-出力関係のみを考慮した記述となっている. さらに, 将来においていつかは真になるという時相オペレータ **AF** が省かれた記述を行うことができた. これにより, 未来における曖昧な時点の動作を考慮する必要がなく, 検証を行うことができる.

5 検証結果

表.1 において, いくつかの非同期ベンチマーク回路の検証結果を示す. これら全ての回路は, Intel Pentium-II 450Mhz processor, 388Mb RAM, Vine Linux 2.1, SMV 検証ツール [14][15] を用い検証を行った. 表.1 において, ("BDD vars") は回路をモデリングするのに必要なブール変数の数であり, Memory と Time は, それぞれ検証に要する容量と時間を示している. また, これらの回路いくつかは, NuSMV 検証ツール [14] とともに配布されている. ここで表中の, pipeline4 は図.2 の非同期式パイプラインをしめしており, pipeline♯ のように ♯ は, そのパイプラインの段数を表している. 例えば, pipeline10 は 10 段から成る非同期パイプラインを意味する. また,

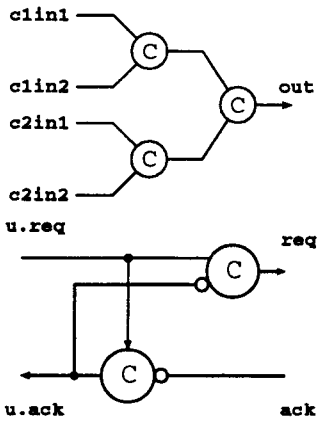

図 3: 検証した回路: (upper) C-element4, (lower) queue モジュール.

C-element4 は, 図.3(upper) のように 3 つの Muller C 素子が接続された 4 入力 1 出力の回路で, queue4 は, 図. 3(lower) の queue モジュールを 4 段, 数珠繋ぎしたものである. 検証結果より, C-element4, queue4 や pipeline4 などの小規模な回路においては, 提案する手法を用いても検証に必要なメモリ容量と時間は大きな差は見られなかったが, 回路の規模が大きくなるにつれてその効果が現われてくることがわかる. 特に, パイプラインの段数を増やすことでその効果が著しく現われる.

6 まとめ

本研究では, モデル・チェッキングの仕様記述をする際の時相論理式を帰納的に導出する手法を提案した. 回路またはシステムが複雑になりすぎると, それに伴い仕様記述も複雑度が増し, 設計者は時相論理式の記述を熟知しなければならないが, 今回, 提案する手法を用いることで, 信号遷移のイベントの入力-出力関係のみを考慮するだけで, 時相論理式を記述することが可能となる. その際, 出力-入力関係のイベントを考慮しないことに特徴がある. そしてさらに, 時相論理式を導出する過程で strong/weak temporal order relations の概念を考慮することで, 全ての信号イベントを記述することなく, 入力信号イベント間の生起順序関係 (weak temporal order relation), 出力-入力信号イベント間の生起順序関係 (Strong temporal order relation) を表現することが可能となった. またそれに伴い, 必要のない信号イベントを省略した結果, 検証に要するメモリ容量と処理時間の減少が計れた. 今後は, 他の検証ツールとの比較検討を行い, 今回提案した手法の有効性を調べていく予定で

表 1: 検証結果

		with our method		without our method	
Circuit name	BDD vars	Memory(KB)	Time(secs)	Memory(KB)	Time(secs)
C-element4	19	4355	0.11	4355	0.1
C-element16	217	5283	0.32	5283	0.39
queue4	55	4430	0.11	4430	0.12
pipeline4	69	4457	0.14	4507	0.19
pipeline10	141	5424	0.51	5551	0.82
pipeline20	261	15207	34.45	15358	125.42
dme1	359	6397	0.59	6430	0.68
dme2	369	8408	1.29	8445	1.35
abp4	67	5184	0.57	5811	1.12
pci	129	6634	0.98	6727	1.08

ある.

7 謝辞

本研究に対して多大なご支援をいただきました高柳記念電子科学技術振興財団に深謝致します. また, 本研究を共に推進した本研究室の博士後期課程:山田親稔院生に感謝致します.

参考文献

[1] E.M. Clarke, O. Grumberg, and D. A. Peled: *Model Checking*, MIT Press, 2001.

[2] M. Huth and M. Ryan: *Logic in Computer Science*, Cambridge University Press, 2001.

[3] M. Dwyer: *Model Checking Software*, Springer, 2001.

[4] T. Kropf: *Introduction to Formal Hardware Verification*, Springer, 1999.

[5] Kenneth L. McMillan, "Symbolic Model Checking," Kluwer Academic Publishers.

[6] B.Berard et al., "*Systems and Software Verification - Model-Checking Techniques and Tools*," Springer.

[7] Chikatoshi Yamada, Yasunori Nagata and Zensho Nakao: "Efficient Verification of Asynchronous Circuits Exploiting Temporal Order-Based Reduced-STG," *Proc. of 2001 International Technical Conference on Circuit/Systems, Computers and Communications* Vol. II, ITC-CSCC2001, pp.965-968, July 2001.

[8] Tam-Anh Chu, "Synthesis of self-timed VLSI circuits from graph-theoric specifications," In *Proc. International Conf. Computer Design (ICCD)*, pp.220-223. IEEE Computer Society Press, 1987.

[9] Gerard J. Holzmann, "The Model Checker SPIN," *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp.279-295.

[10] 吉川 宜史, 米田 友洋, "非同期式回路の検証における liveness クラスに関する考察," 電子情報通信学会誌 A Vol. J81-A No.4 pp.1-12 1998 年 4 月.

[11] 熊谷 貞俊, 鷹田 憲久, "ペトリネットによる離散事象システム," コロナ社.

[12] 村田 忠夫, "ペトリネットの解析と応用," 近代科学社.

[13] 今井 雅 他, "非同期式 VLSI 設計用 CAD ソフトウェア技術の研究," 第 19 回 IPA 技術発表会 2000 年 10 月.

[14] `http://sra.itc.it/tools/nusmv/` .

[15] `http://www-2.cs.cmu.edu/~modelcheck/` .