組込みアセンブリプログラムのモデル構築によるモデル検査

公下 亮佑[1,a)]　山根 智[1]　櫻井 孝平[1]

概要：組込みシステムは，広く利用されており，徐々に複雑化している．複雑化している組込みシステムを安全に運用するためには，システムの安全性を保証することが重要である．複雑なシステムに対する検証手法として，モデル検査が有効である．我々は，組込みシステムのアセンブリプログラムの全振る舞いから，モデルを自動的に生成するツールを開発した．ツールによって生成されたモデルを，モデル検査器に入力することで，システムの安全性を保証する．また，状態爆発に対応するため，Delayed NonDeterminism をベースとした，不定値と呼ぶ抽象化手法を導入する．

キーワード：モデル検査，組込みシステム，アセンブリプログラム，検証

Model Checking by Modeling for Embedded Assembly Programs

Konoshita Ryosuke[1,a)]　Yamane Satoshi[1]　Sakurai Kohei[1]

Abstract: Embedded systems have been widely used. In addition, embedded systems have been gradually complicated. It is important to ensure the safety for complex systems. Model checking is effective to ensure the safety for complex systems. We have developed Behavior Extractor to automatically model all behaviors of embedded assembly programs. Then, we ensure the safety by model checkers. In addition, we have introduced the undefined value to reduce the number of states. The undefined value is based on the Delayed NonDeterminism.

Keywords: Model Checking, Embedded System, Assembly Program, Verification

1. はじめに

組込みシステムは，広く利用されており，徐々に複雑化している．複雑化された組込みシステムを安全に運用するためには，安全性を保証することが重要である．しかし，複雑なシステムの安全性を，テストによって保証するには，多大なコストと時間を要する．複雑なシステムに対する検証手法として，モデル検査 [1] が有効である．モデル検査は，システムやプログラムから導出されるモデルが，満たすべき性質を充足するかどうかを検証する．モデル検査は，自動的かつ網羅的にモデルを探索するため，人の手では発見しにくいエラーも発見できる．しかし，既存のモデル検査器は，完全なモデルの構築が実現されておらず，組込みシステムを完全に検証することはできない．それゆえに，組込みシステムを検証することができる，新しいモデル検査手法が必要とされている．

この問題に対し，B.Schlich らによる，マイコンのモデル検査を行う，[mc]square[7][8] が開発された．このモデル検査器は，アセンブリプログラムを対象とした検証を行っている．

我々は，上記の研究を基礎とし，動的プログラム解析により，組込みシステムのアセンブリプログラムの全振る舞いを，自動的にモデル化する，Behavior Extractor（BEx）を開発した．これは，アセンブリプログラムを網羅的にシミュレーションし，その結果からモデルを生成する．アセンブリプログラムを検証対象としている理由は 2 つある．1 つは，レジスタレベルの解析を目的としているためである．これにより，スタックオーバーフローやスタックアンダーフローなどの性質を容易に解析できる．もう 1 つは，時間に依存するエラーを発見するためである．例えば，組込みシステムにおいて重要な性質である，リアルタイム性などを指す．しかし，アセンブリコードは，C 言語などの高級言語と比べて，コード数の増加を招いてしまう．そのため，状態爆発 [2] が起こりやすくなるという問題点も存在する．我々は，この問題に対処するため，ビット単位で

[1] 金沢大学
Kanazawa University
[a)] rkonoshita@csl.ec.t.kanazawa-u.ac.jp

の抽象化を行う不定値を導入し，状態爆発を抑制する．また，本研究では，Renesas 社のマイコン H8/3687[3] を対象としている．内部装置として，複数のタイマや，A/D コンバータなどを搭載している．

本論文の構成は以下の通りである．まず 2 章では，モデル検査と，モデルを表現する Kripke 構造について，簡潔に述べる．3 章は，本研究の目的となる，我々が開発した BEx の内容について述べる．また，実装する上で問題となる，割込み処理の扱いについて述べる，加えて，状態爆発問題への対処として，BEx に実装された抽象化手法である，不定値についての概要と，実際に扱う際の例を述べる．4 章にて，実装を行った BEx による実験を行い，5 章で本論文のまとめを行う．

1.1 関連研究

2009 年に B.Schlich らにより，C 言語に対応した既存のモデル検査器（e.g. BLAST[4]，BOOP[5]）では，組込みシステムを検証できないことが報告されている [6]．理由として，既存の C 言語に対応したモデル検査器が，ANSI C をターゲットとしているためである．そのため，独自の拡張などを行った組込みシステムの記述を，検査器が解析できなかった．この結果を受け，2010 年に B.Schlich らにより，組込みシステムのアセンブリプログラムを対象とした，モデル検査器 [MC]SQUARE[7][8] が開発された．[MC]SQUARE は，アセンブリプログラムをシミュレートして，モデルを構築し，モデルの構築と並行して，モデルに対する検証を行っている．この生成されたモデルは，マイコンのクロックサイクルを考慮しておらず，間接的に考慮すると述べられている．また，抽象化手法として，Delayed NonDeterminism(DND)[9]，Dead Variable Reduction(DVR)[10][11]，Path Reduction(PR)[11] などが用いられている．DND は，非決定的な値（e.g. 外部入力）を，できるだけ非決定的な値のまま扱い，非決定的な値のまま扱えなくなった場合は，できるだけ分割が少なくなるように具体化するという手法である．

本研究との違いとして，モデルへクロックサイクルを組み込んでいることが挙げられる．これにより，クロックサイクル依存の割込みのタイミングを一意に決定できるため，余分な経路を削減できる．さらに，時間的な要素を考慮していることになるため，デッドラインなどのリアルタイム性検証を可能にできる．また，抽象化手法として，DND の考え方をベースとした，不定値という方法を用いている．不定値は，ビット列を対象とした抽象化手法で，DND 同様に，具体化が必要な場合にのみ具体化される．

2. モデル検査と Kripke 構造

本研究で用いるモデル検査と，モデルである Kripke 構造 [12] について，簡潔に述べる．

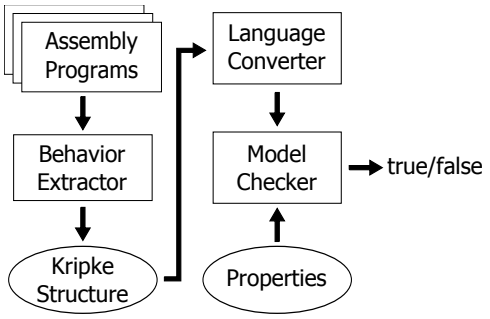

図 1 検証システムの構成
Fig. 1 The Construction of our Verification System

2.1 モデル検査

モデル検査は，形式的検証手法の 1 つであり，システムを表現しているモデル（状態遷移系）の状態空間を網羅的に探索する．Kripke 構造 M と，満たすべき性質 ϕ を入力し，M における状態 $s \in S$ が，ϕ を満たすかどうかを，全ての状態に対して調べることで，性質の充足を検証する．性質は，LTL や CTL[13] などの，時相論理式で記述される．

2.2 Kripke 構造

Kripke 構造は，非決定性有限オートマトンの一種である．Kripke 構造 $M = (S, R, L)$ は，以下のように構成される．

- S：状態の有限集合
- $R \subseteq S \times S$：状態間の 2 項関係
- $L : S \to 2^{AP}$：各状態で真となる命題の集合（AP) を割り当てるラベリング関数

3. 検証システム

3.1 検証システムの全体図

本研究の全体図は，**図 1** のようになる．まず，複数のアセンブリプログラムをモデル生成器 Behavior Extractor へ入力することで，アセンブリプログラムの全振る舞いから，Kripke Structure（Kripke 構造）を生成する．この Kripke Structure は，各状態集合 S，状態間の遷移関係 R，外部環境からの入力・出力など，その状態で発生した事象や，各種レジスタの情報などで表現される命題を保持している．Kripke Structure と，Properties（性質）を Model Checker（モデル検査器）へ入力することで，検証結果や反例を得る．しかし，Behavior Extractor の出力である Kripke Structure は，モデル検査器の Input Format（モデル検査器に対応した入力形式）に整形されていない．そのため，Language Converter を実装し，任意の Model Checker（e.g. NuSMV[14]）の Input Format へ変換する必要がある．

3.2 Behavior Extractor

BEx は，組込みシステムのアセンブリプログラムの全振る舞いを，自動的にモデル化するツールである．モデル

は，アセンブリプログラムの網羅的探索によって生成され，Kripke 構造として表現される．この Kripke 構造には，命題としてレジスタやメモリの値を利用できる．そのため，スタックオーバーフロー，スタックアンダーフローなどの違反を検出できる．また，マイコンの内蔵モジュールのような，ハードウェアの影響も，Kripke 構造に含んでいる．網羅的探索は，抽象化手法の不定値によって実現されている．

本節では，まず，アセンブリプログラムから Kripke 構造を構築する際のアルゴリズムを示す．このアルゴリズムは，割込みの有無や，種類によって挙動を変える．そのため，割込み処理の考え方・アルゴリズム上での扱い方に関しても述べる．また，BEx に実装されている抽象化手法である不定値について，例を用いて説明する．

3.2.1 モデル生成のアルゴリズム

Argorithm 1 から，Kripke 構造を生成する．まず，現状態 s において，実行可能な割込みを全て実行し，対応する状態を生成する（line 7,16）．次に，s が持つプログラムカウンタ（PC）に指定される命令を実行し，対応する状態を生成する（line 9,29）．ただし，割込み処理の段階で，タイミングを一意に決定できる割込みを実行した場合は，PC に指定される命令は実行できない（line 8）．これら 2 つのフェーズで生成された状態は，AddNewState でモデルに追加される．

InterruptHandling（line 16）では，割込み処理を行っている．割込み実行可能な割込み i に対応する PC_i を，割込みベクタテーブルから取得し，PC を更新する（line 20）．その後，フラグのマスク（line 22）や，解放（line 23）を行ない，割込み処理を実行させる．

AddNewState（line 35）では，新しく生成した状態を，モデルに追加する処理を行う．まず，新しく生成された状態 s' と同じ情報を持つ状態 s'' が，モデル上に既に存在するかどうかを調べる（line 36）．もし存在しなければ，s と，s' 間で遷移を構築し（line 39,40），$list$ に s' を追加する（line 41）．もし存在するならば，s' の生成を取り消し，s と s'' 間で遷移を構築する（line 37）．以上を繰り返し，$list$ が空になれば，Kripke 構造の生成を終了する（line 5）．

3.2.2 割込み処理の実行判定

組込みシステムには，割込み処理が存在する．割込みは，現在の処理を一時的に中断し，重要度の高い処理を先に行う．割込みは，システムによって様々な種類が存在しており，ユーザ独自に定義することも可能である．Algorithm 1 の 18 行目では，割込みが可能かどうかをチェックしている．この節では，割込みの実行の可否判定などについて説明する．

いくつかの割込み処理を実行できる場合，どの割込み処理を行うかは，以下の要素から判定する．

Algorithm 1 Model generation algorithm

```
 1: S := {s_0}                                          ▷ set of states
 2: R := ∅                                  ▷ set of relations between states
 3: list = [s_0]        ▷ states which don't search successor states
 4: function Main
 5:     while list.length ≠ 0 do
 6:         s ← head(or tail) of list                  ▷ current state s
 7:         InterruptHandling(s)
 8:         if decidable interrupts don't exist then
 9:             ExecuteInstruction(s)
10:         end if
11:         remove s from list
12:     end while
13: return (S, R)
14: end function
15:
16: function InterruptHandling(s)
17:     for all i ∈ Interrupts do
18:         if i is interruptible then
19:             s' ← s                                 ▷ copy
20:             PC_i = VectorTable[i]          ▷ get vector address
21:             s'.PC = PC_i                   ▷ set PC_i to PC of s'
22:             GlobalMaskBit_s' ← true                ▷ mask
23:             InterruptFlag_s'_i ← false             ▷ flag clear
24:             ExecuteInstruction(s')  ▷ interrupt is executed
25:         end if
26:     end for
27: end function
28:
29: function ExecuteInstruction(s)
30:     operation ← memory[s.PC]                   ▷ get operation
31:     s' ← execute(s, operation)
32:     AddNewState(s, s')
33: end function
34:
35: function AddNewState(s, s')
36:     if ∃_{s''∈S}(s' = s'') then
37:         R := R ∪ (s, s'')              ▷ add new relation to R
38:     else
39:         S := S ∪ s'                        ▷ add new state to S
40:         R := R ∪ (s, s')
41:         add s' at the tail of list
42:     end if
43: end function
```

A 割込み要求の有無
B 割込みイネーブル（マスク）ビットの状態
C 割込みのタイミングが決定可能かどうか
D 割込みの優先度

タイマのオーバーフローをトリガーとした，タイマ割込みを例として考える．このタイマ割込みは，タイマがオーバーフローすると，指定されたビットを真にすることで，割込みの要求を出す．この要求がなければ，割込み処理は行われない（A）．また，割込みの処理の要求を出しても，Interrupt Enable bit（IE）や，Global Mask bit（GM）によって，許可されていなければ実行できない（B）．さらに，タイミングを決定できる割込みが存在することにより，その割込み以下の優先度を持つ割込みは，実行できない（C）（D）．これに関しては，3.2.4 で例を用いて説明する．

割込みのタイミングは次のように決定できる．先程と同様に，タイマオーバーフローをトリガーとした，タイマ割込みを例にあげる．

$$if_s = (tc_s + c_i > lim_s) \vee if_{s'} \quad (1)$$

$$ti = if_s \wedge \neg gm_s \wedge ie_s \quad (2)$$

式 (1) は，タイマオーバーフローを計算している．if_s は，現状態 s における Interrupt Flag（IF）で，右辺を計算した結果，オーバーフローが発生していれば真になる．右辺は，タイマのカウントを示す tc_s と，現在の命令 i を処理した場合にかかるクロックサイクル数 c_i を足し，タイマの上限値である lim_s を超えているかどうかを判定している．また，前状態 s' において，タイマ割込みが処理されず，$if_{s'}$ が真のままになってる場合は，if_s も真となる．if_s が真になれば，割込みの要求を出す．式 (2) は，割込みの実行条件である．gim_s は，GM であり，このビットが真の場合は，割込みを実行できない．ie_s は，各割込みに存在する IE で，このビットが偽である場合は，割込みを実行できない．式 (2) を計算し，真になれば，割込みを実行する．この 2 つの式から，割込みのタイミングを決定することができる．

割込みは，タイミングを決定できるものに関しては，決定した方が良い．タイミングを決定することが望ましい理由は，タイミングを決定しない場合，偽反例を導出してしまう可能性があるためである．例えば，**図 2**，**図 3** のようなモデルを考える．図 2 は，モデルにクロックサイクルが組み込まれており，図 3 では組み込まれていない．モデルの形は変わっているが，どちらも同じプログラムをモデル化したものである．発生しうる割込みは，タイマオーバーフローをトリガーとした割込み 1 つのみならば，割込みの競合が発生しない．そのため，クロックサイクルを考慮することで明白になるタイマカウントや，割込みを制御するマスクビットなどから，割込み処理を実行するタイミングを一意に決定できる．図 2 では，タイマオーバーフローが，最初のノードで発生している．タイマオーバーフローが起こっており，タイミングを決定できるため，PC の指す命令を実行した経路（Instruction at PC）は生成されず，タイマ割込みの経路（Timer Interrupt）のみが生成される．次のノードでは，先程タイマオーバーフローが発生しているため，（タイマの上限値が限りなく低い場合を除いて）タイマオーバーフローは起こらない．よって，Instruction at PC のみを生成するため，エラーロケーションへ到達しない．一方，図 3 では，クロックサイクルを考慮していないため，最初のノードでタイマオーバーフローを検知できない．当然，次のノードでも検知出来ないため，どちらも可能性があると考え，Timer Interrupt と，Instruction at PC の両方が生成される．この場合，エラーロケーションへ到達する．タイマを組み込んだ場合の方が，モデルとしては正確であるため，図 3 におけるエラーロケーションは偽反例である．このような，偽反例を取り除くためにも，割込みのタイミングを把握することは重要である．

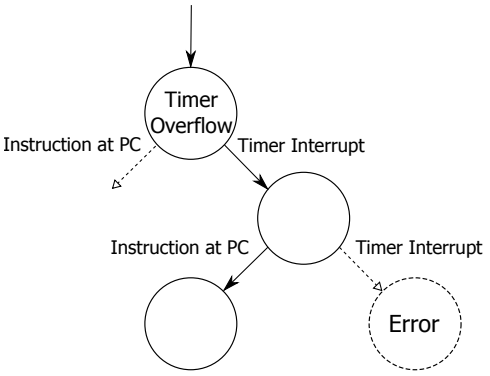

図 2 クロックサイクルを含むモデル
Fig. 2 A Model with Clocks

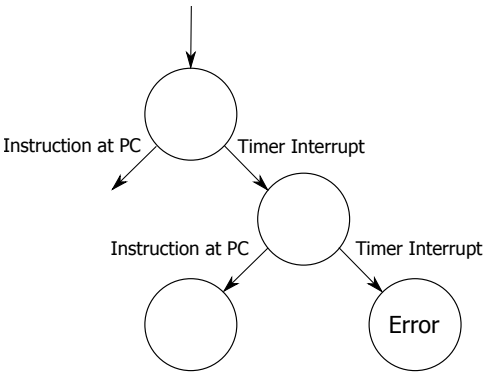

図 3 クロックサイクルを含まないモデル
Fig. 3 A Model without Clocks

Example 1. 割込み実行の条件を満たす割込みが存在する場合における，実行する割込みの選択方法を，**表 1** から説明する．ここでは，4 種類の割込み $\{I1, I2, I3, I4\}$ を仮定する．$\{I2, I3\}$ は，様々な要素からタイミングを一意に決定できるとする．また，$\{I1, I4\}$ は，外部信号をトリ

表 1　解析時の各割込みの実行判定
Table 1　The Judgment of Interrupts in the Analysis

	enable interrupts	$I1$	$I2$	$I3$	$I4$	PC
case 1	$I1$	T	F	F	F	T
case 2	$I1, I4$	T	F	F	T	T
case 3	$I1, I3, I4$	T	F	T	F	F
case 4	$I1, I2, I3, I4$	T	T	F	F	F

ガーとし，タイミングを決定できないとする．さらに，割込みの優先順位は，$I1 > I2 > I3 > I4$ であるとする．PC は，PC が指す命令である．

例えば，case 1 の場合を見る．case 1 では，$I1$ が，実行可能な割込み (enable interrupts) となっている．$I1$ は，タイミングを決定できない割込みであるため，「$I1$ が実行される可能性がある．$I1$ が実行されなければ，PC に従って処理を継続する.」と判断される．そのため，$I1$ を実行した際の経路と，PC に従った場合の経路を生成しなければならない（T）．それ以外は，実行するための条件が満たされていない（enable interrupts に記述されていない）ため，経路は生成されない（F）．

case 2 では，タイミングを決定できない割込み $I1, I4$ が，実行可能になっている．これは，「$I1$ が実行される可能性がある．$I1$ が実行されないならば，$I4$ が実行される可能性が出てくる．$I4$ も実行されないならば，PC に従って処理を継続する.」と判断される．よって，$I1, I4, PC$ はどれも T であり，他は F となる．

case 3 は，case 2 に，タイミングを決定可能な I3 が，加わっている．この場合は，「$I1$ が実行される可能性がある．$I1$ が実行されなければ，$I3$ が必ず実行される．$I1, I3$ のどちらかで確定するため，$I4$ は実行できず，PC に従った処理も行わない.」と判断される．このように，タイミングを決定できる割込みがある場合は，これより優先度の低い割込みと，PC に従う処理は，実行されない．よって，$I1, I3$ は T となり，他は F となる．

case 4 に関しては，case3 と同様に考えればよく，$I1, I2$ のみに可能性があるため，$I1, I2$ は T で，他は F となる．

このようにして，実行する割込みと，実行しない割込みを決定する．

3.2.3　不定値

不定値は，DND をベースとした，抽象化手法の 1 つである．ある任意のビットについて，0 か 1 のどちらの可能性もあることを表現する．1 つのビットに着目した場合，2 つの状態を 1 つにすることができるため，状態数の削減に貢献できる．例えば，**図 4** のように使われる．R0 は 16 ビットレジスタで，下位 8 ビットが不明であることを示している．@FF21 は，メモリ上のアドレス H'FF21 の全ビットが不明であることを示す．特に後者のように，全ビットを不定にすると，未知の値を表現できる．この表現は，レジスタの初期値や，外部環境からの入力などに使われる．また，不定値は，可能な限り不定なままで扱う．もし，命令によって参照され，不定なままで扱えなくなった場合には，0 と 1 に分割される．その際，具体化は，必要な部分に限定して行われる．具体化は，主に Algorithm 1 の，31 行目にて，命令の実行と共に行われる．

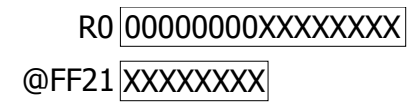

図 4　不定値の表現
Fig. 4　The Representation of the Undefined Value

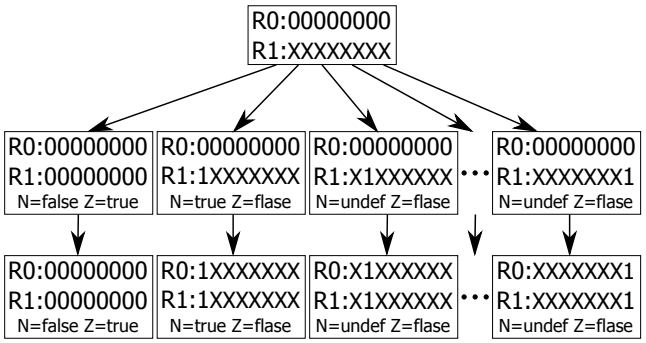

図 5　具体化の手順
Fig. 5　The Procedure of the Concrete

BEx の網羅的探索は，初期値を不定値とし，徐々に具体化していくことによって，実現している．特に，組込みシステムは，初期値を明示していない限りは，電源投入時の値は不明であるため，全てを探索するために，このような表現は必須となる．

Example 2. **図 5** を用いて，具体化の手順について説明する．図 5 は，レジスタ R1 からレジスタ R0 へ，8 ビットデータを転送する際の，不定値の扱い方を表している．R1 は全ビットが不定値であるため，転送する際は，必要な部分を具体化しなければならない．ここで必要な部分とは，データ転送によって影響を受ける，コンディションコードレジスタ（CCR) である．よって，CCR と転送データ間で，矛盾を起こさないように具体化する．

CCR は，計算結果の情報を格納している．H8/3687 における CCR は，8 ビットで構成されており，それぞれが計算結果に応じて変化する．データ転送によって変化するのは，データの正負に影響を受けるネガティブフラグ（N フラグ）と，データが，ゼロか非ゼロかによって影響を受けるゼロフラグ（Z フラグ）である．N フラグは，最上位ビット（MSB）を見れば判断できる．Z フラグは，任意のビットに 1 つでも 1 が入っていれば，0 ではないことが分かる．最初に，Z フラグに対しての矛盾を解消していく．この例では，全ビットが 0 である状態と，任意のビットに 1 を含む状態の，9 つに分けることができる．その後，N フラグの評価を行ない，さらなる具体化が必要であれば，具体化

を行う．MSB が 1 の場合は true，0 の場合は false，不定値の場合は不定値によって，矛盾なく表現できるため，N フラグに対して具体化をする必要はない．矛盾は全て取り除かれたため，R1 から R0 へデータを転送する．最後は，具体化前と具体化後，つまり，根と葉のみを取り出し，結果とする．

不定値を使わずにモデル化する際，図 5 の例では，（8 ビットデータの転送であるため）2^8 の経路を生成しなければ，モデルを表現できない．しかし，不定値を使った場合は，9 通りの経路生成で全てを表現できている．このことから，状態数削減にある程度効果を持つことが分かる．一方で，命令によっては効果がほぼない場合もある．例えば，加算・減算命令などの場合である．加算・減算命令の場合は，部分的な具体化によって，CCR との矛盾を解消することは困難となっている．そのため，これらの命令に対して，効果的な方法を模索する必要がある．

4. 実験

本章では，クロックサイクルを考慮している BEx によるモデル構築と，クロックサイクルを考慮しない場合の両方の実装を行い，両者の差を比較する．共に不定値は実装されており，クロックサイクルを考慮することによる，割込みの扱いのみが異なる．

不定値を実装せずモデル構築を行う場合は，考え得る初期値の組み合わせの数だけモデルを作る必要がある．そのため，不定値を実装した場合と比較すると，状態数が増えるのは明白であるため，実験には含めない．

4.1 単純なプログラムを対象とした実験

対象となるアセンブリプログラムは，**表 2** の関数からなり，**図 6** のような流れになる．_startup から開始し，スタックポインタを設定する．_startup から，メモリ初期化関数__INITSEC を呼び出し，メモリ初期化後に_startup へ戻る．_startup から，_main を呼び出し，さらにタイマ初期化関数_tim_b1_init を呼び出す．_tim_b1_init にて，タイマがオーバーフローする間隔や，割込み制御ビットの操作を行い，割込み実行可能な状態にした後，_main へと戻る．その後，_main の処理を行うが，_main の処理が終わる前に，タイマオーバーフローが発生した場合，割込み処理（_int_tim_b1）が実行される．割込み処理が終わると_main へと戻る．

このプログラムを，以下の実験環境においてモデル化する．

- Windows7 64bit Home Premium
- Intel(R) Core(TM) i3-2120 CPU @3.30GHz
- JVM ヒープ領域 約 2GB

また，BEx の実装は，Java と Scala にて行った．

- Java 1.7.0_45 約 15000 行
- Scala 2.10.3 約 5000 行
- 使用ツール：JFlex[15] Jacc[16]

表 2 単純なプログラムの構成
Table 2 The Construction of the Simple Program

function	C code(line)	Assembly Code(line)
_startup	-	3
_main	5	11
__INITSEC	5	41
_tim_b1_init	5	11
_int_tim_b1	7	17

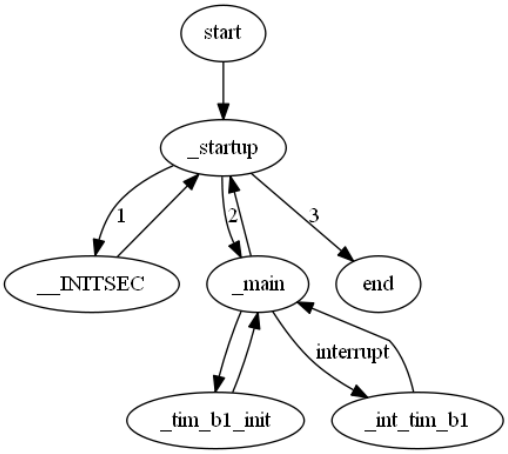

図 6 単純なプログラムのフロー
Fig. 6 The Flow of the Simple Program

表 3 単純なプログラムにおける実験結果
Table 3 The Result of the Experiment in the Simple Program

	Not considering cycles	Considering cycles
states	1044	37
edges	1106	36
time(msec)	32405	359

クロックサイクルを考慮した場合と，しない場合のモデルは，**表 3** のような結果となる．この例においては，状態数などで，約 30 倍の差が出た．クロックサイクルを考慮しない方は，マスク・イネーブルビットの条件が整えば，無秩序に割込みを行うため，単純なプログラムを対象にした場合でも，**図 7**（一部簡略化）のようなモデルを構築してしまう．一方，クロックサイクルを考慮した場合は，分岐が一切存在しない．図 6 のプログラムは短いプログラムであるため，実際には，タイマオーバーフローが発生しない．つまり，タイマ割込みが起こらないことを示している．

4.2 実システムを対象とした実験

先ほどの実験では，非常に極端な例を用いたが，ここでは，実際のハードウェア制御も含めた実験を行うことで，実システムに対するモデル構築を目指す．

対象となるシステムは，3 つの LED と 3 つセンサを持っている．センサは白黒を識別することができ，色によっ

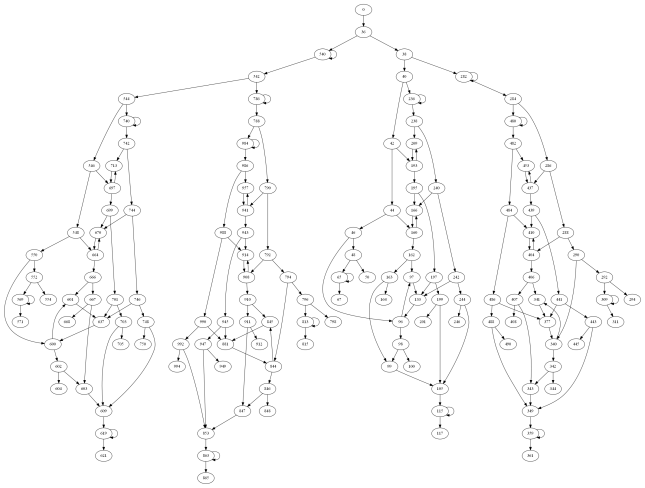

図 7 クロックサイクルを含まずモデル化
Fig. 7 Modeling without Considering Cycles

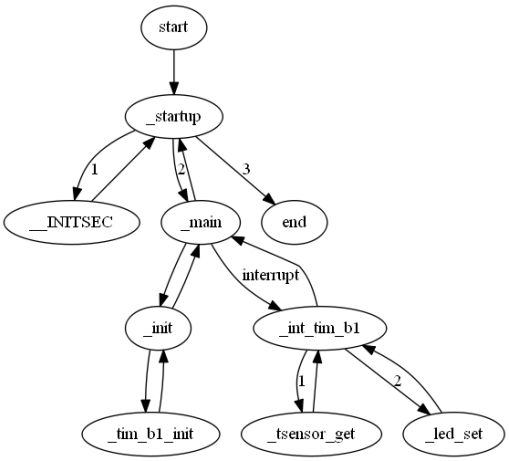

図 8 実システムにおけるプログラムのフロー
Fig. 8 The Flow of the Program in the Actual System

表 4 実システムにおける構成
Table 4 The Construction of the Actual System

function	C code(line)	Assembly Code(line)
_startup	-	3
_main	5	8
_INITSEC	5	24
_init	8	11
_tim_b1_init	5	7
_int_tim_b1	19	47
_tsensor_get	3	5
_led_set	3	3

て 0 か 1 を出力する．また，ソフトウェアによって，センサの出力の組み合わせを取得し，対応する LED を点灯させる．例えば，センサ 1 とセンサ 2 が黒を検出した場合，LED1 と LED2 が制御によって点灯する．センサから値を取得し，対応する LED を点灯させる処理は，タイマオーバーフロー割込みとして記述されている．トリガーとなるタイマーオーバーフローは，約 100μsec 毎に発生する．

表 4，**図 8** にソフトウェアの詳細を示す．4.1 節の実験と同様に，スタックポインタの設定から，メモリの初期化までを行い，_main を呼び出す．その後，_main からハードウェアの初期化関数_init を呼び出し，タイマの初期化やセンサ，LED などの設定を行う．最後に，割込みを実行可能にして_main へ戻る．_main に戻ると，カウンタを持つループに入る．カウンタは，タイマ割込み関数（_int_tim_b1）が実行されるたびにカウントアップし，規定回数の割込みを実行すると，ループを抜けた後に終了する．_int_tim_b1 では，センサ値の取得や LED への出力を行い，_main へと戻る．

先ほどの実験と同様の方法でモデル化を行う．環境は以下のとおりである．

- Gentoo 3.10.7
- Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz 8 コア
- JVM ヒープ領域 約 8GB

クロックサイクルを考慮した際の結果は，**表 5** である．n は割込みの実行回数を指しており，割込みを実行する回数を増やした時の状態数，時間などの増加を見ている．

図 9 から，n に対する実行時間は，大きく増加していることがわかる．その理由として，不定値と外部環境からの値による，数多くの分岐が発生しているためであると考えられる．

この対象プログラムでは，マスクビットの操作が明確に行われている．よって，クロックサイクルを考慮し，タイマカウンタを明白にすると，タイマ割込みは決定的に扱える．つまり，割込み処理による分岐は発生しない．しかし，センサが読み取る色は，マイコンに依存しない外部環境であるため，センサが読み取る値は未知である．したがって，センサからの値取得が行われるたびに不定な値を取り入れており，不定な値を LED への出力として扱うために具体化しなければならない．3 つのセンサから得られる組み合わせは 8 通りであるため，割込みが発生するたびに，新たに 8 通りの経路が生成される．それによって，状態数や実行時間の大きな増加を引き起こしている．

また，クロックサイクルを考慮せずに実装したもので実験を行った場合，$n = 2$ の時点でタイムアウトとなった．この時の実行時間は約 140 時間で，状態数は 513883 である．このような $n = 2$ の小規模な例の場合であっても，状態爆発によりタイムアウトとなった理由として，クロックサイクルを考慮しないがために，無秩序に割込みが実行されたと考えられる．具体的には，以下の 2 点である．

- **図 10** は，割込み待ちのループ（アセンブリプログラムで 3 行）をモデル化した場合の構造を表している．左側はクロックサイクルを考慮した場合，右側は考慮しない場合のモデルである．左側のモデルは，タイマカウントの情報を保持しているため，同じ情報を持つ状態と認識されない．そのため，ループを構成するこ

表 5　実システムにおける実験結果
Table 5　The Result of the Experiment in the Actual System

n	state	edge	time(msec)
2	2616	2668	40866
4	12440	12724	208953
8	55317	56645	971307
16	163997	168013	3791905
32	381357	390749	13012868

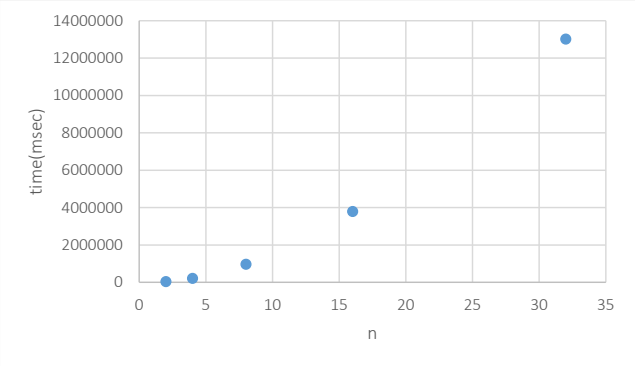

図 9　n に対する実行時間の増加
Fig. 9　Increasing the execution time for n

とができず，1 つの長いパスを生成する．しかし，クロックサイクルの考慮によって，タイマカウントが明白となり，割込みのタイミングは一意に決定できるため，割込みのパスも 1 つ生成されるに留まる．一方，右側のモデルは，クロックサイクルを考慮しないため，タイマカウントを無視せざるを得ない．よって，同じ情報を持つ状態が生成され，ループ構造を持つ．しかし，ループ構造内で無秩序に割込むため，ループを形成している 3 つの状態全てにおいて割込みが行われている．前述のとおり，割込みが発生した場合，8 通りの分岐が生成されるため，クロックサイクルを考慮しないモデル（右側）の方が，状態数が爆発的に増加すると予想される．

- 今回の例の場合，割込みルーチンを抜ける前に，Global Mask bit 以外の割込みに関するビットを許可に設定している．さらに，割込み実行時に退避されている Global Mask bit は，割込み終了時に元に戻る．つまり，割込みが終了し，割込み待機のループに戻った時点で，割込みが実行可能な状況となっている．クロックサイクルを考慮している場合は，タイマのカウンタによって，再度割込むことを抑制できる．しかし，考慮していない場合は，割込みから戻った直後に，再び割り込み処理を実行できてしまう．そのため，状況によっては，状態数を爆発的に増加させてしまう要因となる可能性がある．

このように，時間を考慮するかどうかで，実行時間・状態数などに大きな差が生まれる．なおかつ，3.2 節で述べ

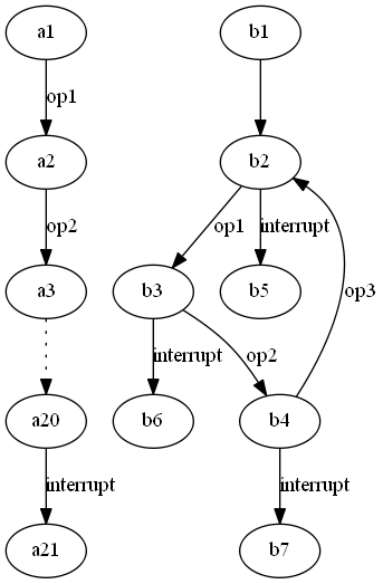

図 10　割込み待機ループのモデル化
Fig. 10　Modeling the Loop Structure to Wait the Interrupt

たように，クロックサイクルを考慮しなかった図 7 は，不正確な経路を含んでいる可能性がある．このことから，タイマなどを使う場合の検証においては，クロックサイクルを考慮すべきであると言える．

5.　まとめ

本論文では，組込みアセンブリプログラムに対するモデル検査，特に，Kripke 構造を生成する BEx について述べた．BEx は，動的プログラム解析により，組込みシステムのアセンブリプログラムの全ての振る舞いや，マイコンの内蔵モジュールのような，ハードウェアの影響をモデル化するためのツールである．また，BEx を用いてモデル化する際に問題となる，割込み処理の扱い方について述べた．我々は，モデルに時間（クロックサイクル）を考慮し，より正確なモデルを得ることで，必要のない命令を実行させず，経路を削減し，偽反例を生成しないようにしている．また，状態爆発問題が起こりうるという問題があった．この問題に対し，不定値を導入し，実際の例から，ある程度の状態削減効果があることを示した．さらに，不定値により，初期値や外部環境の値を表現できるようになり，網羅的探索を実現できた．

6.　展望

今後は，3.1 節で述べた，Language Converter を実装し，Kripke 構造を記号モデル検査器 NuSMV への入力へ変換することにより，実際にシステムの検証を行うことを目指す．

また，モデル構築器い関しては，記号的実行 [17][18][19] への切り替えや，組み合わせによってさらなる状態数の削減を目指す．特に，加算・減算命令時における，状態数の

増加を可能な限り抑制する．また，現在は，モデル生成と検証の工程が完全に切り離されている．この 2 つの工程を同時に並行して実施し，エラーが存在する場合の，検証時間短縮を目指す．

参考文献

[1] Clarke, Jr., E. M., Grumberg, O. and Peled, D. A.: *Model Checking*, MIT Press (1999).

[2] Clarke, E. M., Emerson, E. A. and Sifakis, J.: Model Checking: Algorithmic Verification and Debugging, *Commun. ACM*, Vol. 52, No. 11, pp. 74–84 (2009).

[3] Corporation, R. E.: Renesas Electronics, Renesas Electronics Corporation (online), available from ⟨http://japan.renesas.com/⟩ (accessed 2014-6-17).

[4] Team, B. .: MTC (Models and Theory of Computation): BLAST Project, EPFL (online), available from ⟨http://mtc.epfl.ch/software-tools/blast/index-epfl.php⟩ (accessed 2014-6-17).

[5] Weissenbacher, G.: The BOOP Toolkit v0.42, Graz University of Technology (online), available from ⟨http://boop.sourceforge.net/⟩ (accessed 2014-6-17).

[6] Schlich, B. and Kowalewski, S.: Model Checking C Source Code for Embedded Systems, *Int. J. Softw. Tools Technol. Transf.*, Vol. 11, No. 3, pp. 187–202 (2009).

[7] Schlich, B.: Model Checking of Software for Microcontrollers, *ACM Trans. Embed. Comput. Syst.*, Vol. 9, No. 4, pp. 36:1–36:27 (2010).

[8] Schlich, B., Brauer, J. and Kowalewski, S.: Application of Static Analyses for State-space Reduction to the Microcontroller Binary Code, *Sci. Comput. Program.*, Vol. 76, No. 2, pp. 100–118 (2011).

[9] Noll, T. and Schlich, B.: Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code, *LNCS 4899*, Springer-Verlag, pp. 185–201 (2008).

[10] Holzmann, G. J.: The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited, *LNCS 1680*, Springer-Verlag, pp. 232–244 (1999).

[11] Yorav, K. and Grumberg, O.: Static Analysis for State-Space Reductions Preserving Temporal Logics, *Form. Methods Syst. Des.*, Vol. 25, No. 1, pp. 67–96 (2004).

[12] Browne, M. C., Clarke, E. M. and Grumberg, O.: Characterizing Kripke Structures in Temporal Logic, *LNCS 249*, TAPSOFT '87/CAAP '87, Springer-Verlag, pp. 256–270 (1987).

[13] Quemener, Y.-M.: Model-Checking of Infinite Kripke Structures Defined by Simple Graph Grammars, *Segragra'95, ENTCS 2*, pp. 67–74 (1995).

[14] Team, N. D.: NuSMV home page, ITC-IRST, Carnegie Mellon University, the University of Genoa and the University of Trento (online), available from ⟨http://nusmv.fbk.eu/index.html⟩ (accessed 2014-6-17).

[15] Klein, G.: JFlex - The Fast Scanner Generator for Java, CSE UNSW (online), available from ⟨http://jflex.de/⟩ (accessed 2014-6-27).

[16] Jones, M. P.: Jacc: just another compiler compiler for Java, Department of Computer Science and Engineering at the OGI School of Science & Engineering at OHSU (online), available from ⟨http://jflex.de/⟩ (accessed 2014-6-27).

[17] King, J. C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[18] Păsăreanu, C. S. and Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, ACM, pp. 179–180 (2010).

[19] Khurshid, S., Păsăreanu, C. S. and Visser, W.: Generalized Symbolic Execution for Model Checking and Testing, *LNCS 2619*, TACAS'03, Springer-Verlag, pp. 553–568 (2003).