組込みアセンブリプログラム解析によるSMTモデル検査

小橋 潤平[†2,a)]　山根 智[†1,b)]　竹下 淳[,c)]

概要：近年の発展を続けている組込みシステムにおいて, ハードウェアに依存する性質を持つソフトウェアが用いられており, 開発期間の短縮や信頼性の向上の観点から, それらの性質に対応した形式的検証手法について需要が高まっている. 我々の研究では組み込み向けのアセンブリコードの静的プログラム解析を基礎として, SMT ソルバを用いた有界モデル検査手法について取り扱う. 本論文では, レジスタの内容, アドレス空間上の変数・定数, 割り込み動作を含めたアセンブリプログラムの振る舞いのモデル化について, 基本ブロックに着目したモデル化手法を採用し, 実験により手法の有効性を示す.

キーワード：形式的検証; プログラム検証; アセンブリ言語; SMT ソルバ; 有界モデル検査;

SMT-Based Model Checking by Analyzing Embedded Assembly Program

Jumpei Kobashi[†2,a)]　Satoshi Yamane[†1,b)]　Atsushi Takeshita[,c)]

Abstract: Recently, embedded assembly programs have properties dependent on hardware at the process of development. Thus, demands about the established formal verifications corresponding to those properties are increasing from the point of view of shorter development and high reliability. Our work aims at developing a SMT-Based Bounded Model Checking method besed on based on static analysis for embedded assembly programs. Our method generates the program model with basic blocks which considered interruptions. In addition, the model has the behavior of registers and an address space. In this paper, we develop a parser and a model converter for assembly programs. Moreover, we show the validity of our method by experiments.

Keywords: Formal Verification; Program Checking; Assembler Language; SMT-based BMC; Bounded Model Checking;

1. はじめに

1.1 研究概要

近年の組込みソフトウェアの多くはハードウェアの操作や, ハードウェアの挙動に依存する動作を行うプログラム (I/O レジスタなどの読み込み/書き込み, 割り込み例外処理による値の更新を見込んだ変数の存在, 割り込みのためのポーリングなど) である. しかし, 既存のプログラム検証ツールはハードウェアに依存する性質を扱えない場合があり, それらに対応した検証ツールについて需要が高まっている. また, ハードウェアの制約が強い低資源環境や厳密なタイミングを求められる環境ではアセンブリ言語により実装する場合がある. しかし, アセンブリ言語がマイコンによって異なることもあり, 対応している検証ツールは少ない. 一方で, 開発期間の短縮や信頼性・製品品質の向上の観点から, アセンブリプログラムに対して, より高速に不具合を検出することができる形式的検証のツールが求められていることは事実である.

我々の研究は組込みソフトウェアのアセンブリコードに対して, プログラムの静的解析により, 命令によるレジスタの値の変化や割り込み処理を含めた振る舞いをモデル化し, SMT ソルバを用いた有界モデル検査 (SMT ベース BMC) を適用することで安全性検証を行う事を目的としている. 当研究では, 事例対象としてルネサス・エレクトロニクス製 H8/3687 マイコン [1][2] を搭載した車輪ロボットのファームウェアのアセンブリコードを扱う.

†1 現在, 金沢大学 理工学域 School of Science and Engineering, Kanazawa University, Japan
†2 現在, 金沢大学大学院 自然科学研究科 Graduate School of Natural Science and Technology, Kanazawa University, Japan
a) jkobashi@csl.ec.t.kanazawa-u.ac.jp
b) syamane@is.t.kanazawa-u.ac.jp
c) atakeshita@csl.ec.t.kanazawa-u.ac.jp

本論文では [11] にて提案された制御フローの基本ブロックに着目したモデル化手法を採用し, 実験により効果を示すことを主眼としている. モデル化手法を自動的に適用するためのツールとして, H8/3687 マイコンのアセンブリコードの構文解析器とモデル変換器を開発した. 実証実験として, 作成したツールによりアセンブリコードのモデルを作成し, バッファオーバーフローの安全性検証を行った.

1.2 関連研究

SMT ソルバとは, 背景理論付き充足可能性問題 (Satisfiability Modulo Theories Problem; SMT Problem)[3] を解決するツールの総称である. SMT 問題は与えられた一階述語論理式について全体が真になるような変数の割り当てが存在するかどうかを判定する問題である. 整数や実数などのデータ型の演算や関数を扱うことができ, 一階述語論理式の表現力により応用分野が多岐に渡る. 応用の一つがモデル検査への応用であり, SMT ベース有界モデル検査 (SMT-Based Bounded Model Checking) と称される [4][5][6]. 組込み向け ANSI-C プログラムに対して SMT ベース BMC を適用した研究も行われている [7][8].

アセンブリプログラムに対する検証では, D.Brylow らは, Z86 マイコンで割り込み駆動するアセンブリプログラムについて, マイコンの構成要素の振る舞いを部分的にモデル化した制御フローを作成し, 状態爆発の発生を抑えた割り込み動作の静的検査を行っている [9]. Broylow らによる研究はスタックサイズの静的解析, スタックされているデータの型検査, 割り込みの遅延時間の解析に留まっており, スタック操作を伴わない多くの命令については扱っていない.

B.Schlich らは ATMEL ATmega16 マイコンのシミュレータを作成し, 同マイコンで動作するアセンブリプログラムのモデル検査を行っている [10]. Schlich らはシミュレータを用いてコードを解析, 状態空間を作成し, 割り込み動作を含めたモデル検査を適用している. Schlich らによる手法ではシュミレータとモデル検査器が独立しており, マイコン毎のシミュレータを用意することで異なるマイコンで用いられるアセンブリコードの検証を可能としている.

我々の研究の前身として, 竹下らによる組込みアセンブリコードに対する SMT ベース BMC が提案されている [11]. 竹下らの研究では, アセンブリコードの制御フローグラフの基本ブロックに着目したモデル化手法を提案している. [11] にて提案されたモデル化手法は割り込み処理を含めたモデルを, 変数や論理式の数を抑えて作成することが出来るとされている. しかし, [11] では実験が不足しており, 手法の効果が十分に確認できたとは言いがたい.

2. 組み込みアセンブリプログラム向け SMT ベース BMC

2.1 SMT ベース BMC によるプログラム検証

SMT ベース BMC とは, 状態遷移システムについて, 有限回数の状態遷移による実行経路を一階述語論理式として記号的に表現し, 検証性質を満たすかを SMT ソルバを用いてチェックする形式的検証手法である [4][5][6]. プログラム検証に SMT ベース BMC を適用する場合, プログラムは制御フローグラフ (Control Flow Graph; CFG) から振る舞いを抽出することで生成される状態遷移システムとしてモデル化される [7][8].

状態遷移システム M に対して, 経路長 k の実行経路における安全性質 $AG(\phi)$ を検査するための検証条件 ψ^k を式 (1) に示す.

$$\psi^k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i) \qquad (1)$$

状態遷移システム $M = (\mathbb{S}, \mathbb{T}, s_0)$ において, $\mathbb{S}$ はシステム状態 s_i の集合, $s_0 \in \mathbb{S}$ はシステムの初期状態, $\mathbb{T} \subseteq \mathbb{S} \times \mathbb{S}$ は状態遷移をモデル化した命題論理式の集合である.

式 (1) において, $s_i \in \mathbb{S}$ はステップ i におけるシステム状態, $I(s_0)$ は初期状態の値を割り当てる論理式, $T(s_i, s_{i+1})$ はステップ i からステップ $i+1$ への状態遷移 $T = (s_i, s_{i+1}) \in \mathbb{T}$ における状態の制約を与える論理式, $P(s_i)$ はステップ i における性質 ϕ の論理式である.

検証条件 ψ^k において, $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ はパス長 k の実行パスおけるシステムの振る舞いを表し, 常に真である. 性質 $P(s_i)$ に否定演算子が付加されていることから, 性質違反が存在する場合に充足可能となり, 性質違反に至る実行パス (=反例) が得られる. 一方, 充足不能であるなら, k までに到達しうる状態において性質が保証される. SMT ソルバは与えられた式 (検証条件) が真となるように値の割り当てを探索するため, 不具合の発見に有効である.

無限ループが存在したり, 大規模なシステムでは実行経路が長大化して状態爆発を起こし, 検証がままならない場合がある. そのため, 実行の途上で発生しうる不具合の発見に焦点を定め, 経路長 k を $k = 0$ から徐々に増やして繰り返し検証を行う. この時, 経路長がシステムの直径に至るまで検証結果が充足不能である場合, 検証性質が保証されると見なすことができる.

プログラム検証においては CFG のノードが 1 状態に相当し, 状態 s_i はプログラムの実行に付随するプログラムカウンタとすべてのプログラム変数の値の集合で構成される. 状態遷移 $\mathbb{T}$ はプログラム記述の実行の集合である. SMT ソルバは一つの変数に対して一つの値を割り当てるため, プログラムの変数にバージョン情報を付けてリネームし, 静的単一代入形式 (Single Static Assignment Form; SSA

Form[12]) へ変換して状態毎のバージョンを与える. ループを含むプログラムでは, ループアンローリングにより各イテレーションを明確にして, ループを取り除いた上で SSA 形式への変換を施す必要がある. プログラム記述はプログラムカウンタの値を遷移条件とし, 等式, 述語, 関数などを用いて振る舞いを表す一階述語論理式として表現され, 制御フローの分岐を選言で表すことで, 制約式 $T(s_i, s_{i+1})$ を構成する.

2.2 アセンブリプログラムに対する SMT ベース BMC

アセンブリプログラムに対する SMT ベース BMC では, 状態は CPU の構成要素 (レジスタ, アドレス空間) への値の割り当てによって決定する. レジスタを固定長ビットベクトルのデータ型の変数として, アドレス空間をアドレスから値を写像する固定長ビットベクトルの関数として見なす. 状態遷移はアセンブリ命令の実行が該当する. 構成要素が持つ値や, 構成要素に対する命令の振る舞いを述語論理式として表現してモデル化する. 命令の構成要素に対する振る舞いとして, 構成要素間のデータの移動や四則演算, ビットレベルでの操作が挙げられる. データの移動は要素間の等号関係で, 演算はビットベクトルの演算関数で, ビットレベルの操作はビットベクトルの抽出・結合関数を用いて表現できる. 命令による構成要素に対する振る舞いについて, PC やステータスレジスタの更新など, 全てがオペランドとして明示されない場合があるため, マイコン毎の構成やアセンブリコードの構文に従ったモデル化が必要となる.

2.3 H8/3687 アセンブリコードのモデル化

当研究で用いる H8/3687 マイコンの構成要素は汎用レジスタ (16bit × 16 本), プログラムカウンタレジスタ (PC, 24bit × 1 本), コンディションコードレジスタ (CCR, 8bit × 1 本), アドレス空間 (64kByte) が存在する [2]. 汎用レジスタの内, 8 本目 (ER7) はスタックポインタとしての機能が割り当てられている. 汎用レジスタは 8bit または 32bit レジスタとしても扱うことができる (図 1). この特徴に対し, 各状態においてレジスタに相当する 8 個の 32bit の固定長ビットベクトルの変数を用意し, より小さいビット幅のレジスタとして扱う際には, 固定長ビットベクトルの背景理論でサポートされる抽出・結合操作を利用することでレジスタに対する処理をモデル化することができる.

アドレス空間はプログラム領域, スタックなどのデータ領域含めて通常は 64kByte までサポートしているため, 16bit の固定長ビットベクトルを 8 ビットの固定長ビットベクトルに写像する関数としてモデル化する. 汎用レジスタをアドレスレジスタとして用いたり, PC を扱う際には抽出操作により下位 16bit のみを用いる. ワードデータ (16bit), ロングワードデータ (32bit) のアクセスは結合操作を用いる.

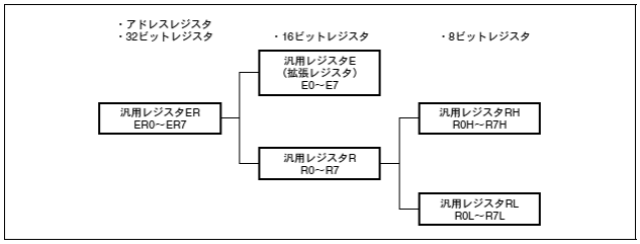

図 1: H8/3687 の汎用レジスタ [2]

また, ポストインクリメント/プレデクリメントレジスタ間接アドレッシングを行う際には, アクセス後/前にアドレスレジスタの更新が行われるため, この処理に相当する論理式を命令処理の論理式と結合し, SSA 形式に変換する必要がある. PC や CCR は通常コード上に現れず, 命令の実行や例外処理時に暗黙的に更新される. そのため, プログラム解析時点で更新される PC, CCR を明示し, 該当命令での CCR の処理に相当する論理式を命令処理の論理式と結合する必要がある.

表 1 にプログラム中のニーモニックに対するマイコンのオペレーション, 及びそのオペレーションを論理式に変換した例を示す. 命令の振る舞いはマイコンのマニュアルでの記載に準じて論理式として変換する. 論理式の表記は SMT-LIB2 形式 [13] に準ずる. 表 1 では論理式には PC や CCR に対する振る舞いを省略している点に注意されたい. 実際には PC や CCR に対する振る舞いも論理式化し, オペレーションの論理式との連言として命令の論理式を構成する.

(a) MOV 命令を始めとする転送命令は前状態の転送元 (ERs) を表す変数 (ERs_i) と次状態の転送先 (ERd) を表す変数 (ERd_{i+1}) との等式によって振る舞いが表現できる.

(b) より小さいビット幅にに対する転送や ADD 命令など, 演算を伴う命令はビットベクトルの背景理論によりサポートされる抽出・結合・演算関数 (concat, extract, bvadd など) を組み合わせた等式で表現する.

(c) 関数として定義されるアドレス空間 (AS) へのアクセスは間接アドレスを引数として指定し, 格納・参照共に等式で表現する. ワードデータ, ロングワードデータは参照はエンディアンに従った順番で引数を指定し, 結合関数を用いて表現する. 格納の際はアドレス空間関数も SSA 変換を施すため, 格納した番地以外の値を次のバージョンにコピーする必要がある. 当研究では全称量化子 (∀, forall) を用いた等式で行う.

3. アセンブリコードブロックを用いるモデル化手法

3.1 アセンブリコードブロックとは

我々の研究で採用するアセンブリコードブロック (ACB)

	ニーモニック	オペレーション	論理式
(a)	MOV.L ERs,ERd	ERs32 → ERd32	$(= ERd_{i+1}ERs_i)$
(b)	ADD.W Rs,Rd	Rd16 + Rs16 → Rd16	$(= ERd_{i+1}(concat((\_extract\ 31\ 16)ERd_i)$ $(bvadd((\_extract\ 15\ 0)ERd_i)((\_extract\ 15\ 0)ERs_i))))$
(c)	MOV.B Rs,@ERd	Rs8 → @ERd	$(and(= (AS_{i+1}((\_extract\ 15\ 0)ERd_i))((\_extract\ 15\ 0)ERs_i)$ $(forall((x(\_BitVec\ 16)))(or(x = (\_extract\ 15\ 0)ERd_i)(= (AS_{i+1})(AS_i)))))$

表 1: H8/3687 アセンブリ命令の論理式表現 (SMT-LIB2 記法)

とは, プログラムの CFG 作成で用いられる基本ブロックを拡張し, 割り込み動作を考慮しつつアセンブリ命令をブロック化するアイデアである [11].

割り込み動作を含めたアセンブリコードの検証を考えた場合, 割り込みは 1 命令毎に発生することが考えられるため, ブロック化を行なわなず, 1 命令を 1 状態に対応させた CFG を作成する必要がある. しかし, この方法では状態数が膨大となり, モデルも巨大化するため検証不可能となる場合がある. これに対して, 検証する割り込み関数と検証性質を限定し, 割り込み処理が発生してもデータの整合性に影響を与えなかったり, 割り込み制御フラグの操作により割り込み処理が発生しないコードはブロック化し, 状態やモデルのサイズを削減することが ACB の目的である. ACB ではコード内の以下の 4 要素を分割点とする.

(1) 分岐命令 (ジャンプ命令, サブルーチンコール, リターン命令を含む)
(2) 分岐命令の遷移先 (ルーチンやルーチン内部のラベル)
(3) 割り込みルーチンで用いられる要素 (レジスタ, アドレス空間上の変数) を指定する命令
(4) 割り込み制御フラグの操作を行う命令

これらの分割点は, 字句・構文解析時に明らかとなるので, ACB をノードとした CFG を作成することができる. 図 2 に ACB をノードとした CFG の例を示す. 割り込みルーチンへの遷移では, 割り込みハンドラの処理をモデル化した論理式により, 状態に制約が与えられる.

この手法では制御フラグの操作により例外処理の発生を制御でき, CCR やアドレス空間上の割り込み制御レジスタで値が参照可能な割り込み要因について取り扱うことができる. 従って, 現段階において当研究では, リセット端子によるシステムのリセット要求やマスク不可能な割り込みは扱わない.

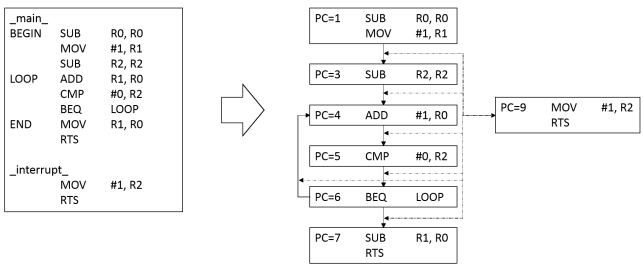

図 2: 割り込み処理を含む ACB をノードとした CFG

3.2 アセンブリコードブロックを用いる CFG の論理式表現

ACB をノードとする CFG から検証条件 ψ^k を構築するため, 検証条件 ψ における状態遷移の論理式 $T(s_i, s_{i+1})$ を式 (2), (3) の様に書き換える. $T(s_i, s_{i+1})$ は ACB 内の命令列による遷移の選言により定義される.

$$T(s_i, s_{i+1}) := \bigvee_{|x|} ACB_x(s_i, s_{i+1}) \tag{2}$$

$$ACB_x(s_i, s_{i+1}) := G_x(s_i) \wedge (s_i = s_{i,0}) \wedge \bigwedge_{j=0}^{|y|-1} INST_y(s_{i,j}, s_{i,j+1}) \wedge (s_{i+1} = s_{i,|y|}) \tag{3}$$

式 (2) において, $ACB_x(s_i, s_{i+1})$ は前状態 s_i から遷移可能な x 個目の ACB による遷移に相当する論理式である. 1 ステップの遷移の論理式 $T(s_i, s_{i+1})$ には $|x|$ 個の ACB による遷移が含まれる. $ACB_x(s_i, s_{i+1})$ は式 3 で示す遷移条件とブロック内の命令処理を表す論理式との連言からなる.

式 (3) において, $G_x(s_i)$ は $ACB_x(s_i, s_{i+1})$ に対する遷移条件であり, 状態を構成する PC 値の等式である. 前状態の PC 値が遷移条件の PC 値と異なる場合はその ACB の式は成立しない. 状態に PC の値は 1 つしか持たないため, 状態遷移は常に 1 つの ACB の実行が有効になる. $INST_y(s_{i,j}, s_{i,j+1})$ は ACB に含まれる y 個目の命令処理に相当する論理式であり, ACB の内部状態 $s_{i,j}$ に制約を与える. $ACB_x(s_i, s_{i+1})$ には $|y|$ 個の命令が含まれる. 式 $(s_i = s_{i,0})$ 及び式 $(s_{i+1} = s_{i,|y|})$ により, 実行された ACB により与えられる制約をそのステップにおける状態の制約として反映させる.

4. 実験

4.1 検証手順

実験に際して作成した検証ツールにおける検証手順を図 3 に示す.

(1) "Parser/Lexer" ではアセンブリコードを解析してプログラムで使用される構成要素や命令を取得し, 各行にプログラムカウンタの値を追加する.
(2) "Make Control Frow Graph" ではループアンローリングや関数のインライン展開を行い, プログラムに従った CFG を作成する. 作成するためのアルゴリズムは,

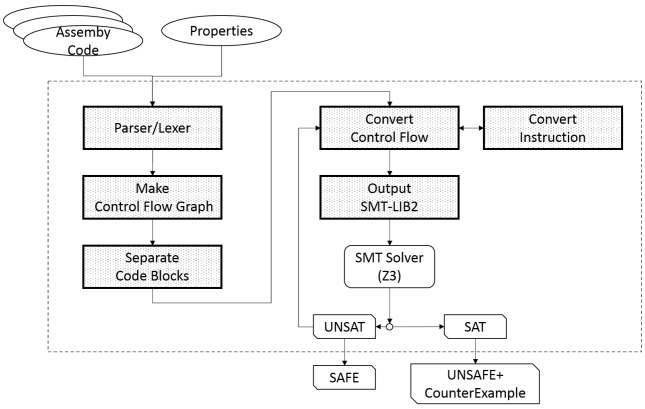

図 3: 適用手順

[11] で提示されたアルゴリズムに準ずる.

(3) "Convert Control Flow" では CFG に従って状態遷移の論理式を構築し, 検証条件に結合する. "Convert Instruction to Logic" では ACB に含まれるアセンブリ命令を, 変数や関数に対して命令処理と等価に振る舞う論理式へと変換する. "Convert Control Flow" と "Convert Instruction to Logic" ステップは交互に実行され, 検証条件を生成する.

(4) "Output SMT-LIB2" では生成した検証条件を SMT ソルバの入力形式として代表的な SMT-LIB2 形式 [13] に出力し, SMT ソルバに入力して結果を確認する. 結果が充足不能である場合, 探索する経路長を増やし, 再び検証条件を構築して検証を繰り返す. 結果が充足可能である場合, 直ちに性質が満たされないことを割り当て結果と共に出力して停止する.

当研究では H8/3687 マイコンで用いられるアセンブリコードに対する構文解析器とモデル生成器を Java プログラムとして実装した. 開発したモデル生成器では, H8/3687 マイコンの命令セットの内, ブロック転送命令, スリープ命令以外の命令について述語論理式に変換することができる. また, 静的解析時点で遷移先が不明であるレジスタ間接ジャンプや, ランタイムライブラリのルーチンへのジャンプは扱うことができない.

本論文におけるモデル生成器の実装では, 割り込み例外処理について, 割り込み状態への遷移, 或いは割り込み状態からの遷移は考慮するが, 割り込まれる関数の内容は考慮しないモデルの作成を行う. 即ち, 割り込みのモデル化は割り込みハンドラのモデル化に留まる.

実験では車輪型ロボット教材「e-nuvo®WHEEL」において, ライントレースロボットのファームウェアで用いられるルーチンのアセンブリコードを対象に, 提案手法によるモデルの削減効果の調査と, 作成したモデルに対する SMT-Based BMC を適用した結果の調査を行った. 表 2 に実験環境を示す.

CPU	Windows 7 Professional 64bit
OS	Core i7-3770 CPU @3.40GHz
メモリ	16GB
SMT ソルバ	Microsoft Z3 v4.3.0[14]
java	Ver. 1.7.0_45
プログラム	約 8,400 行

表 2: 実験環境

4.2 状態数削減実験

状態数削減実験で対象とした関数はファームウェアのメイン関数 (_main) と CPLD にエンコーダの値を書き込む関数 (_cpld_write) である. _main はハード・ソフトの初期化を行った後, 無限ループに入る関数である. _cpld_write は CPLD が接続されている IO ポートのデータレジスタにスタートビット, 命令, アドレス, データを順に書き込んでいく関数である. この 2 つの関数について, 1 命令に対し 1 状態を生成する手法によるモデル (lazy) と, ACB に対し状態を生成する提案手法によるモデル (ACB) での CFG のノード数を比較した. 表 3 に比較した結果を示す.

対象関数	行数	ノード数 (lazy)	ノード数 (ACB)	削減率	作成時間 [ms]
_main	67	311	228	0.69	442
_cpld_write	48	911	657	0.72	678

表 3: 状態数削減結果

表 3 の結果により, 提案手法により状態数は 3 割程度削減されていることがわかる. この結果は変数の総数やコピーを行う論理式の数も削減されていることも示している. モデルについて, _cpld_write が _main と比較して行数に対する状態数が多い理由として, _cpld_write でのデータ書き込み時のループがアンローリングにより展開されたためと考えられる.

4.3 BMC 適用実験

BMC 適用実験は前述の実験で作成した_cpld_write 関数のモデルを対象に, スタックオーバーフロー/アンダーフローの発生について安全性検証を行った. スタックオーバーフロー/アンダーフローの発生はプログラムにおいてスタックにアクセスする命令が実行される際に, スタックポインタの内容がハードウェアで定義されているスタック領域の範囲内であるかを調べることで検証する. ここでは検証結果を得ることのできる遷移回数とその検証結果, モデル作成を含めた検証時間, その遷移回数までの累積実行時間を測定した. なお, 検証器には制限時間として 3600 秒 (1 時間) を設定し, 累計検証時間がこれを超えた場合は結果にかかわらず停止させる. 表 4 に結果を, 図 4 に遷移回数に対するソルバでの検証時間のグラフを示す.

	実行行数	検証結果	検証時間 [sec]	累計時間 [sec]
lazy	3162	UNSAT	2.12	3601
ACB	20447	UNSAT	3.94	3602

表 4: SMT-Based BMC 適用結果

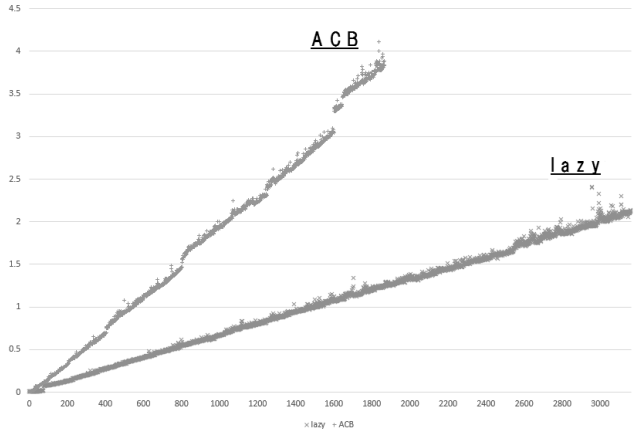

図 4: 遷移回数-検証時間グラフ

lazy モデル, ACB モデルの検証は, 共に結果が累計検証時間が制限を超えて停止した. 停止時点での結果は共に UNSAT であり, 共に到達した遷移回数まではスタックオーバーフロー/アンダーフローは発生しないことが確認できた. 表 4 において, 累計検証時間に対する検証可能なコード行数は lazy モデルが 0.88 行であるのに対し, ACB モデルでは 5.68 行となるため, 検証効率は ACB モデルの方が良好であることが確認できた. 一方, 図 4 から, 1 回の遷移についてのソルバでの検証時間は lazy モデルの方が短いことが確認できた.

検証の結果について, それぞれの遷移回数時点でスタックオーバーフロー/アンダーフローに対する安全性は保証されているが, それ以後の遷移において違反が確認できていないため, プログラムの検証には更に長時間の検証が必要となる.

5. まとめ

本稿では組込み向けのアセンブリプログラムの SMT-Based BMC に関して, 割り込みを考慮した検証を行うために, ACB を用いて状態数の削減を試みるモデリング手法を採用し, 実験を通じてその手法によるモデルの削減と検証時間を示した.

今回の実験では具体的な割り込み動作を含まないため, 今後は割り込まれるルーチンの動作を含めた性質検証を行なっていく. また, プログラム検証としては十分な遷移回数の検証ができておらず, 明確な結果も得られていないため, 他の抽象化手法の導入により状態爆発を軽減出来るモデル生成器の開発や, タイムアウトした遷移回数以降も検証を続ける等, モデル検査器の改良も進めていく.

参考文献

[1] H8S H8/300 シリーズ C/C++コンパイラパッケージ Ver.7.00 ユーザーズマニュアル, RJJ10J2552-0100, ルネサス・エレクトロニクス (株) (2009)
[2] H8/3687 グループハードウェアマニュアル, RJJ09B0151-0500, ルネサス・エレクトロニクス (株) (2005)
[3] Clark Barrett et al. Satisfiability Modulo Theories, Handbook of Satisfiability, Chapter12, pp.737-795 (2008)
[4] Armin Biere et al.: Symbolic model checking without BDDs, LNCS 1579, pp.193-207, Springer (1999)
[5] A.Armando et al.: Bounded model checking of software using SMT solvers instead of SAT solvers, International Journal on Software Tools for Technology Transfer (STTT) archive, Vol.11 Num.01, pp.69-83 (2009)
[6] 梅村 晃広: SAT ソルバ・SMT ソルバの技術と応用, コンピュータソフトウェア, Vol.27, No.03, pp.24-35 日本ソフトウェア科学会 (2010).
[7] L.Cordeiro, B.Fischer, J.Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE/ACM International Conference on Automated Software Engineering, pp.137-148, 2009.
[8] Lucas Cordeiro: SMT-based bounded model checking for multi-threaded software in embedded systems, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol.02, pp.373-376, ACM (2010)
[9] Dennis Brylow et al.: Static Checking of Interrupt driven Software, International Conference on Software Engineering, pp.47-56, IEEE Computer Society (2001)
[10] Bastian Schlich et al.: Model checking of software for microcontrollers, ACM Transactions on Embedded Computing Systems, Article 36, Vol.09, No.04, pp.1-27, ACM(2010)
[11] 竹下淳, 小橋潤平, 山根智: CISC 型組込みアセンブリプログラムの SMT ベースの有界モデル検査 (システム数理と応用), 信学技報, Vol.113, Num.421, pp. 65-70, 電子情報通信学会 (2014)
[12] Steven S. Muchnick: Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., (1997)
[13] Clark Barrett et al: The SMT-LIB Standard: Version 2.0, Department of Computer Science, The University of Iowa, Available at www.SMT-LIB.org (2010)
[14] Leonardo De Moura, Nikolaj Bjørner: Z3: An efficient SMT solver, LNCS 4963, pp. 337-340, Springer (2008)
[15] Christoph Wintersteiger, Youssef Hamadi, Leonardo De Moura: Efficiently solving quantified, Formal Methods in System Design, Vol.42, Num.01, pp.3-23, Springer (2013)
[16] 車輪型ロボット教材 nuvo®WHEEL ver.1.1, 株式会社 ZMP.