組込みアセンブラのSMT検証手法の提案と実装

竹下　淳[†]　小橋　潤平[††]　山根　智[††]

本稿では，組込みアセンブリプログラムを対象としたSMT有界モデル検査手法を以下のように提案する．(1) 検証の前処理として，アセンブリプログラムをSSA形式に変換処理する．(2) アセンブリプログラムのSSA形式からSMT-LIBに変換してSMT有界モデル検査を行う．以上の検証手法をJavaで実装して実証実験を行った．

Verification of embedded software in Assembly code by SMT prover

Atsushi Takeshita ,[†] Junpei Kobashi [††] and Satoshi Yamane [††]

We propose the technique of Bounded Model Checking(BMC) for embedded assembly program. We use SMT solver for BMC. We also propose improvement of SSA conversion algorithm for assembly program and implement the Algorithm using Java language. Finally, we verify the simple assembly program's property of a real embedded robot system.

1. はじめに

昨今の技術発展から，組込みシステムは高速データ通信などの多機能化が進み，機能実現のためハード・ソフト両面の複雑化が進んでいる．これらを対象とした検証は重要な課題であり，プログラムのバグ発見に優れる検証方式として有界モデル検査がある．近年はモデル検査の検証部にSMT(Satisfiabbility Modulo Theory) solverを使う手法が多く見られ，その例としてL.Cordeiroらは組込みシステムのANCI-Cコードのコード変換及び検証[3]を効率的に行っている．SMTを検証部に用いることで，検証性質を簡単な表現に落とす事が出来るため，検査式の大規模，複雑化により検証が不可能となる状態爆発問題の発生を抑えられる．しかしANSI-Cコードの場合，スタックオーバーフローや割り込み処理等のハードウェア依存の性質検証を扱えない問題点があった．これに対して，B.Sclich[6]の研究では組込みシステムのアセンブリコードをモデル検査対象として検証している．これにより，スタックポインタを含めたレジスタへのアクセスや割り込み処理の痕跡を把握でき，より多くの問題を検証できる．ただし，アセンブリコードは高級言語と比べるとコード数が多く，各レジスタ情報も必要になるため，状態爆発が発生する危険度が高くなってしまう．

本研究ではSMT solverを用いた有界モデル検査により，アセンブリプログラムを用いたハードウェア依存の性質を含む検証を行う事を目的とする．検証時に各変数の値を一意に決定するため，コードのSSA(Single Static Assignment)形式への変換を行う．土屋はコード変換方式を含めた，SMTを用いた有界モデル検査方式について解説[10]しており，佐々によるCOINSコンパイラの仕様書[11]などでは各種SSA変換方式について解説がなされている．しかし，これらは一般的なコードに対する変換手順を述べており，組込みアセンブリプログラム中で生じる独自の表記法や命令には対応できていない．本稿では，有界モデル検査で必要となるループ展開処理のアセンブリプログラムへの適用法と，アセンブリ命令中に登場するレジスタ，アドレス空間表現に対し，SSA変換を容易にするための新しい表記法を提案し実装を行った．表記法については節4．1にて述べる．対象システムにはRenesas社製マイコンチップ「H8/3687F」[13](CPU:H8/300 tiny)を搭載した2/4輪駆動ロボットのアセンブリプログラムを用いる．ここで利用するアセンブリプログラムは，システムの動作を記述したC言語を専用ワークショップ上でコンパイルした際に出力される物を利用しており，アセンブリ命令内容を含めたコードの形式はコンパイラの設定依存となる．SSA変換の各処理

† 金沢大学大学院自然科学研究科
Graduate School of Natural Science and Technology ,Kanazawa University
†† 金沢大学理工学域電子情報学類
School of Science and Engineering, Kanazawa University

ⓒ 2012 Information Processing Society of Japan

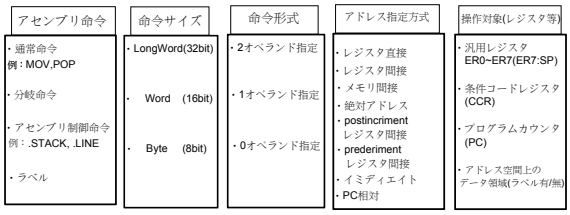

図 1 アセンブリプログラム特性

には，Java 及び構文解析ツール BYacc[15]，JFlex[16] を用いて実装した．次節では，本稿で検証対象とするシステムの概要を述べる．

2. 対象システム

検証対象として取り上げるシステムのアセンブリプログラムは**図 1** のような特徴を持つ．

このロボットは，8 本の 32bit 汎用レジスタを持ち，各アセンブリ命令で利用される．その中の ER7 はスタックポインタとして扱われる．通常命令は四則演算等のデータを扱う命令集合であり，命令サイズにより処理するビット長を指定できる．各汎用レジスタは以下のように特定ビットの使い分けが行われる．

- E(上位 16bit)， R(下位 16bit)
  R0H(R の上位 8bit), R0L(R の下位 8bit) -

このほか，ラベルは関数のトップや，処理の区切りに登場し，分岐命令と共に SSA 変換の CFG 作成時にノード分割点として利用する．また，特定のデータ領域にはラベルが割り当てられ，演算時などに用いられる．分岐命令は各処理後のフラグを保持する条件コードレジスタ (CCR) の値を基に分岐先を決定する．また，アセンブリ制御命令[14] はスタックサイズの設定などのシステム制御を行う．実際のデータ転送の対象は，上記の汎用レジスタ，アドレス空間上のデータ領域の他，PC,CCR に限定される．次節では，検証の流れとそれに用いる各理論の説明を行う．

3. 検 証 手 法

3.1 検証の概要

この節では，対象システムのプログラムに対して有界モデル検査を行う際の手順と，その行程で用いる各理論の説明を行う．プログラムへの有界モデル検査までの流れは大きく次の処理に分けられる．

1. コードのループの展開処理
2. コードの SSA 形式への変換処理
3. SMT-LIB 形式への変換処理
4. SMT solver による検証

このように 3 度のコード変換を経て検証を行うのが今回の手法の特徴である．SMT-LIB は，SMT 上で用いる各理論のフレームワークをまとめたライブラリであり，この形式のコードに対して検証を行う．SMT solver をモデル検査に用いる利点として，状態遷移などを充足可能性問題に落とし込むことで，全ての性質が true，false の 2 値で結果を表現出来ること，その表現中に線形演算などの理論が扱えるという 2 点が挙げられる．ループの展開処理では，展開上限回数 k を決定し，その範囲内でプログラムの展開を行う．有界モデル検査では, 展開したプログラムが指定されたステップ回数内で性質を満たすかどうかを判定できる．プログラム中の各ループ回数が導出可能ならば，適切なステップ数を指定してプログラム全体を検証できる．

本稿では，検証部の SMT solver に，Microsoft Research が作成した Z3[12] を利用する．この SMT solver は多くの理論に対応し，実装例[3] の結果などから，その優秀さが示されている．以降の副節では，SMT solver を用いた有界モデル検査の方法について述べる．

3.2 有界モデル検査

有界モデル検査 (Bounded Model Checking) とは SAT(SMT の前景理論)/SMT solver を使用したモデル検査の一種で, 検証対象である状態遷移システムの有限長の実行パスの範囲において，満たすべき検証性質を検証する手法である[7]．SMT solver を用いる場合，システムの状態遷移を一階述語論理式で, 検証性質は時相論理式で与えられたものを同じく述語論理式で表現してシステムの挙動を SMT 問題へモデル化する．最大実行ステップ回数 k の内, i 回目の実行ステップにおけるシステムの状態を s_i, 状態遷移を $T(s_i, s_{i+1})$ と表現する．初期状態 s_0 を決定する論理式の集合を $I(s_0)$, 性質 ϕ について状態 s_i 時の論理式を $P(s_i)$ と表現して安全性検証 (AGϕ) を行うとすると SMT solver に入力する論理式は次式 (1) の様に構成される．

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

性質 $P(s_i)$ はその否定が式に組み込まれ, k 回以下の状態遷移でいずれかの状態で性質違反 ($P(s_i) = False$) が存在する場合 $\psi^k = True$ となり, 各自由定数に値が割り当てられ，特性に対する反例となる動作モデルが得られる．一方で特性違反が存在しない場合 $P(s_i) = False$ を満たす自由定数の割り当てが存在しないため $\psi^k = False$ となり, 性質が保たれたまま検査が終了する．近年ではプログラム検証の分野においても研究が進められており[3], システムの 1 状態をプログラムに出現する全てのプログラム変数の集合, プ

ⓒ 2012 Information Processing Society of Japan

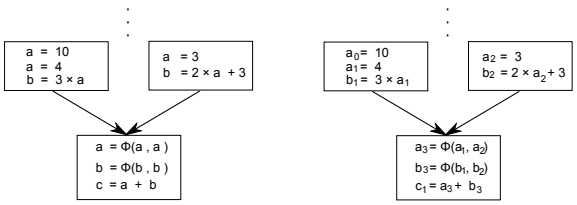

図 2 SSA 変換例 (minimal SSA 方式の適用)

ログラム命令による変数の演算と代入を状態遷移として動作をモデル化する.

アセンブリコードの検証ではレジスタとメモリ上の静的変数の集合が状態を表し, アセンブリ命令による演算が状態遷移である. レジスタやメモリは固定長ビットベクトルの変数, 或いはビットベクトルの引数からビットベクトルの写像を返す関数として見なせば, アセンブリ命令によるレジスタや静的変数に対する演算はビットベクトルの演算や述語としてモデル化が可能である. SMT solver は論理式に含まれる各自由定数に対して値を 1 つ割り当てて原子論理式の真偽を定めて充足可能性を判定するため, プログラム内で複数回の演算・代入処理が行われる変数に対しては SSA 変換によるリネームを必要とする. アセンブリコードにおいては演算に用いられる各レジスタとメモリ上の静的変数のリネームが必要となる. また, アセンブリ命令による CPU の状態変化を述語論理式で表現し, リネーム後のアセンブリコードを SMT 問題として記述するための SMT-LIB 言語変換器の作成が必要である. 以降では, 有界モデル検査の前処理となる SSA 変換の流れを説明する.

3.3 SSA 変換

SSA 形式は, コンパイラのコード最適化等に用いる中間言語方式であり, そのコードは以下の特徴を持つ.

- 1 つの変数に対して代入は 1 度に限定
- ϕ 関数の導入

1 変数 1 代入に限定するために, 複数の代入が存在する変数では各々の変数に値番号を付けることで各代入を個別に扱う. また, ϕ 関数は, **図 2** のようなプログラムの各分岐処理で値が更新されうる変数に対して, 分岐合流地点での値を一意に決定するための疑似処理の事を言う. 本稿では SSA 変換を行うアルゴリズムとして, 文献 1) の minimal SSA 方式を適用している.

有界モデル検査の前処理としての SSA 変換アルゴリズムは, 文献 1) を参考として大まかに次の行程に分けられる.

1. コードのループ展開
2. CFG(Control Flow Graph) 作成
3. 支配辺境ノードの導出
4. ϕ 関数の導入
5. 変数への値番号付け

上記工程に加えてループ展開処理では, 前処理として各ループの展開回数を導出が行われ, 支配辺境ノードの導出処理では, 前処理に支配木の作成が行われる. 各工程の詳細な処理について以下で述べる. なお SSA 変換で必要とされる各処理のアルゴリズムについては, 文献 1),11) などで解説されている.

3.3.1 コードのループ展開

ここでは, コードの有界化のためのループ展開処理を行う. 本研究では, 元プログラムに対してループ展開回数を指定して, その回数分ループを展開した際のパスを作成する事を展開処理として扱う. 図 3 を例に, ループを含むコードの展開処理を示す. なお, 図 3 は配列の初期化処理である. 図 4 は展開結果例である. 実際には, ループボディと, 条件判定部をセットに, 指定回数だけコードの追加を行う. この処理中では, 後の CFG 作成部のノード判定部に混乱を招かないように, ループ中のラベルには添え字を付けて差別化を図る. また, 図 4 の 20 行目の命令などは, 展開時点で分岐先が一意に決まるので, 分岐の役割を失ったという情報を付加する. これにより, 次行程でのサクセッサ, プレデセッサ情報追加先のノードを明確化がなされる.

3.3.2 CFG 作成

ここでは, プログラムの処理を分岐処理などで分割したひとまとまりのノード毎に取り扱い, 処理の流れを明確化する. 処理中で作成される各ノードのサクセッサ・プレデセッサの関係は, 後述の支配木作成や Rename 処理に用いる. また, この行程では実装にて導入するコードの変換処理と, プログラム中に登場する変数の列挙を同時に行う. これらの変数は全て Rename 処理の対象となる. また CFG 作成と同時に, ループ展開処理で添え字付けされたラベルを元のラベル表記に戻す作業を行う.

3.3.3 支配辺境ノードの導出

この処理では文献 1) のアルゴリズムから, ϕ 関数導入の指標となる支配辺境及び Rename 処理に利用する支配木を導出する. このアルゴリズムでは, 直接支配関係による支配木を作成し, そのリーフノードからボトムアップに各ノードの支配辺境を導出する. 図 5 左の例では, ノード X の支配辺境ノードは, Z であり, X の支配辺境ノードとは次のルールに従うノードが該当する. なお, 図 6 は図 5 の支配木である.

· 支配木中のノード X について,

```
1     BRA       L86:8
2  L85:
3     MOV.W     @(-6:16,ER6),R0
4     SHLL.W    R0
5     MOV.W     R6,R1
6     ADD.W     #-12,R1
7     ADD.W     R0,R1
8     SUB.W     R0,R0
9     MOV.W     R0,@ER1
10    MOV.W     @(-6:16,ER6),R0
11    INC.W     #1,R0
12    MOV.W     R0,@(-6:16,ER6)
13 L86:
14    MOV.W     @(-6:16,ER6),R0
15    CMP.W     #3,R0
16    BLT       L85:8
```

図 3 ループを含む処理

```
1     BRA       L86:8
2  L86:
3     MOV.W     @(-6:16,ER6),R0
4     CMP.W     #3,R0
5     BLT       L85:8
6  L85:
7     MOV.W     @(-6:16,ER6),R0
8     SHLL.W    R0
9     MOV.W     R6,R1
10    ADD.W     #-12,R1
11    ADD.W     R0,R1
12    SUB.W     R0,R0
13    MOV.W     R0,@ER1
14    MOV.W     @(-6:16,ER6),R0
15    INC.W     #1,R0
16    MOV.W     R0,@(-6:16,ER6)
17 L86_1:
18    MOV.W     @(-6:16,ER6),R0
19    CMP.W     #3,R0
20    BLT       L85_1:8
21 L85_1:↵
22    MOV.W     @(-6:16,ER6),R0
23    SHLL.W    R0
24    MOV.W     R6,R1
25    ADD.W     #-12,R1
26                ·↵
27                ·↵
28                ·
```

図 4 ループ展開例

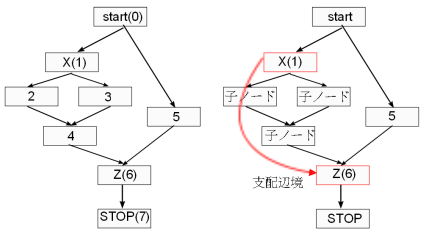

図 5 左：CFG 例，右:ノード X の直接支配関係及び支配辺境

- ノード X のサクセッサノード ”または”
  ノード X の支配木の子ノードの支配辺境ノード
  ”かつ” X に直接支配されていない

支配木作成には Lengauler[2] の，深さ優先探索を用いた CFG の新しいノード関係を利用する方式を採用した．

3.3.4 ϕ 関数の導入

ϕ 関数は，複数のパスが合流する地点での変数の値

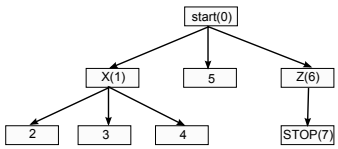

図 6 図 5 の支配木

を決定する為に用いる SSA 変換の特徴的な処理である．しかし，本稿の方式では最終的に 1 本ずつパスを作成し，それらを検証していく方針であるため，この処理および ϕ 関数は飾りの処理である側面がある．このため，実装の簡便さの観点から本研究では minimal SSA 方式を採用している．

3.3.5 変数の Rename 処理

ここでは，コード中の各変数の値を一意にするための Rename 処理を行う．Rename 処理は具体的に，コード中の変数に値番号を付けを行う．論文[1] では，プログラムの先頭ノードから各変数の番号付けを行う．対象となる変数は，各命令中で定義・参照される変数全てであり，各ノードが直接支配するノードのコードを順に変換処理する．定義側の変数には，定義毎に新しい値番号を変数に付与し，参照側は，直前までの命令フロー中で付けられた最新の値番号が付与される．ϕ 関数の参照側には特別処理として，現在のノードのプレデセッサノード処理終了時点で最新の番号を持つ変数が値の候補として随時追加される．

この値番号付けまでの一連の処理を行う事で SSA 変換は完了し，SMT-LIB 形式への変換が続けて行われる．次節では，アセンブリコードに対する SSA 変換処理を実装する際に行ったコードの変換処理及び，SMT solver を用いた検証例とその結果について述べる．

4. 計算機実験

4.1 実　　装

アセンブリコードへの SSA 変換を実装するに際し，以下の 3 つの作業を追加で行った．

- 明示化されていないレジスタ情報の追加
- 汎用レジスタ表記法の変更
- メモリ参照を扱う疑似関数 memory の導入

まず，本稿で取り扱うアセンブリコードは，スタック操作中の ER7(=SP) や，各演算操作後のフラグ管理をする CCR について特に値の参照･定義が行われた事を命令中で明記していないため，これらの情報をコード中に追加し，SSA 変換時に値番号付けを行う．汎用レジスタの表記法は，コード上に登場する ER,E,R などと特定ビット長を個別のレジスタ名で取り扱う方式から，R0→ER0[15:0] といった形の，取り扱うレジ

ⓒ 2012 Information Processing Society of Japan

スタ名及び処理範囲のビットを明記する方式に変換する．これにより，汎用レジスタのデータ更新が発生した場合に，E0〜R0L までの複数のレジスタ情報を更新する手間を省く利点がある．

また，実行時まで参照先のアドレス番地が分からない間接参照処理に対して，memory 関数はコード中のレジスタ表記を引数として対象の番地データに対するデータ転送を疑似的に行う．この memory を用いる方式では，個別のアドレス空間へのデータアクセスが全て memory へのアクセスに集約されることになり，値番号の添付処理が簡便になる事，前術のように実行時まで分からない参照先アドレスについて考慮する手間が要らないという利点があるが，各データの整合性が必要な場合には，各処理におけるデータ参照先アドレスが必要となる欠点がある．なお，ここで新しく示した表記法への変換処理は，CFG 作成処理中で行っている．

CFG 作成時のノード作成の目安には各関数，処理の区切りとなるラベルおよび分岐命令を用い，これらを読み込んだ際に，直前までの処理ひとかたまりを新しいノードとして扱っている．例えば以下のアセンブリ命令 MOV によるデータ転送処理は一度 (1) 式のように変換され，その後 SSA 変換が行われる．

MOV.W @R2 ER0
→ MOV.W memory(@ER2[15:0]), ER0[31:0] (1)

本稿では上記追加処理を含めた SSA 変換の各アルゴリズムを Java 及び構文解析ツールの BYacc,JFlex を用いて実装し，実際の組込みロボットシステムに対し変換を行った．今回処理対象としたプログラムでは，他の関数呼び出し等を含まない独立した関数を対象として，関数内部には条件分岐，ループ変数の増減値が一定であるループ処理以外には通常処理しか含まない事を前提としており，その他のループ展開については実装中である．また変換対象のコードは前記の関数中に登場する者に限定しており，値番号を付ける対象変数として，各アセンブリ命令に登場する 8 本の汎用レジスタ，疑似関数 memory,CCR を取り扱っており，関数外に存在するアセンブリ制御命令は処理対象から外している．

また，H8/3687F CPU のアセンブリコードを SMT-LIB2 形式に変換し，特性を入力せずに SMT ソルバで解決し動作モデルを確認した際の実行環境及び結果を表 1 に示す．特性を入力しないため SMT ソルバの出力は SAT となり 1 つの動作モデルが得られた．

4.2 実　　験

実際の H8/Tiny アセンブリプログラムの安全性検証を行い，SMT ソルバでの結果と実行時間を計測する．対象となるアセンブリプログラムは 10 行程度のごく簡単な例 (**図 7**) で，ANSI-C 言語ソースでは簡単な条件分岐を含む静的変数の代入処理を行う関数である．コメント行で挿入されている `CHECK AG(@_test_var1 > 0)` は検証したい特性 (安全性) を意味する．

検証手順は検証対象となるアセンブリコードで分岐命令がある場合，分岐した際の CFG と分岐しなかった際の CFG を作成し，それぞれに対し SMT 検証を行うものである．次に具体的な流れを示す．

step1.アセンブリコードから CFG 作成，SSA 変換．
step2.アセンブリ命令を同等の述語論理に変換し，SMT-LIB2 入力言語形式で SMT ソルバへの入力ファイルに記述．
step3.検証特性を SSA 変換に準じて述語論理で記述し，その否定の選言をファイルに記述．
step4.ファイルを SMT ソルバに入力して結果を確認．
step5.結果が SAT::特性違反であるため割り当てモデルを出力して終了．
　UNSAT::step1 に戻って他の CFG を作成．

表 1　変換器による変換結果 (上) SMT ソルバでの出力結果 (下)

CPU	Intel Core2 Duo CPU 200GHz
RAM	3.00GB
OS	Windows vista

アセンブリコードの命令数	38
変換器の実行時間 [s]	0.057
SMT ソルバの結果 (SAT/UNSAT/UNKNOWN)	SAT
SMT ソルバのメモリ使用量 [MB]	3.07
SMT ソルバの実行時間 [s]	0.17

```
_test_check:            ; function: test_check
        .STACK    _test_check=4
        PUSH.W    R6
        MOV.W     R7,R6
        MOV.W     @_test_var1:16,R0
        MOV.W     @_test_var2:16,E0
        CMP.W     E0,R0
        BGE       L217:8
        MOV.W     @_test_var2:16,R0
        MOV.W     R0,@_test_var1:16
L217:
        POP.W     R6
        RTS
;       CHECK     AG(@test_var1 > 0)
        .SECTION  B,DATA,ALIGN=2
_test_var1:             ; static: test_var1
        .RES.W    1
_test_var2:             ; static: test_var2
        .RES.W    1
        .END
```

図 7　アセンブリコード例

step6.全ての CFG が UNSAT :: 特性違反が存在しないため終了.

アセンブリ命令から述語論理への変換では汎用レジスタとメモリの振る舞いのみを抽出した. 分岐によって複数の CFG が作成される場合, 全ての CFG を作成・保存するのではなく検査する分岐の CFG をその都度作成することで, 特性違反発見時の手順全体の実行時間に関して CFG 作成にかかる時間を削減できる.

実験結果を表 2 に示す. 対象としたアセンブリコードでは分岐の発生により 2 つの CFG が作成された. このコードでは静的変数の初期化を考慮していないために分岐の発生に関わらず特性違反が発生しうることが予想され, 実際に提案した検証手順に従って実験を行った所, 作成した 2 つの CFG のどちらでも特性違反を検出した. また, 確認のために特性違反を回避する前提条件を加えた際にはどちらの CFG でも step5 で UNSAT を出力した. 短いコードだったため命令の変換に要する時間とソルバでの解答時間は短く, 結果の SAT/UNSAT に関わらず 1 回の検査に要する時間に差がでなかった. 手順全体の所要時間で見ると特性違反が発生する場合には 1 つめの CFG の検査の結果により step5 で終了するのに対して, 発生しない場合には 2 つの CFG の作成・変換・検査のすべての合計時間がかかるため特性違反が発生しないほうが時間がかかるのは明らかである.

表 2 SMT ソルバでの出力結果と所要時間

CPU	Intel Core2 Duo CPU 200GHz	
RAM	3.00GB	
OS	Windows vista	
CFG1(分岐なし) の結果	SAT	0.03 [sec]
CFG2(分岐あり) の結果	SAT	0.02 [sec]
前提条件あり (CFG1)	UNSAT	0.03 [sec]
前提条件あり (CFG2)	UNSAT	0.02 [sec]
前提条件有効時の合計時間		0.05 [sec]

5. ま と め

本稿では，SMT solver を用いた有界モデル検査の優位性及び，組込みシステムのアセンブリコードをその検証対象とする事の意味について述べた．また，SMT 検証部の前処理として，アセンブリコードを対象としたループ展開処理から SSA 変換までの手法を提案し，他の関数呼び出しを含まない，ループ変数の増減値を一意にするという条件の下で，SSA 変換の実装を行い，有界モデル検査を行った．今後は，実際にスタックオーバーフロー等のハードウェア依存する特性の検証，関数呼び出しへの対応を行う事，変換対象外となっている乗算等の通常命令・アセンブリ制御命令を含めた，システム中に登場する全ての命令に対応できるように拡張を行う予定である．

参 考 文 献

1) Cytron, R. et al: Efficiently Computing Static Single Assignment Form and Control Dependence Graph,ACM Trans. Prog. Lang. Syst., Vol.13 , No.4, pp.451-490, 1991
2) Lengauer, T. et al: A fast algorithm for finding dominators in a flowgraph, ACM Trans. Program. Lang. Syst., pp.121–141, 1979
3) Cordeiro,L. et al: SMT-Based Bounded Model Checking for Embedded ANSI-C Software,Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp.137-148, 2009
4) Clark, B. et al: Satisfiability Modulo Theories, Handbook of Satisfiability, Chapter12, pp.737-795, 2008
5) Leonardo de Moura: SMT Solvers: Introduction & Applications, Microsoft Research, 2007
6) Schlich, B. : Model Checking of Software for Microcontrollers, ACM Transactions on Embedded Computing Systems, Article 36, Vol. 9, No. 4, pp.1-27, 2010
7) 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.3, pp.24-35, 2010
8) 岩沼 宏治 他:SMT:個別理論を取り扱う SAT 技術, 人工知能学会誌，25 巻，1 号, pp.86-95,2010
9) 土屋 達弘:記号モデル検査の並行ソフトウェアシステムへの応用について, 回路とシステムワークショップ, pp.303-308, 2004
10) 土屋　達弘:充足可能性判定を利用したモデル検査, 第 7 回プログラミングおよびプログラミング言語 サマースクール, 2009
11) 佐々政孝:　静的単一代入形式最適化システム及び関連ユーティリティ，http://www.is.titech.ac.jp/∼sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf，pp.1-15，東京工大，2010
12) Z3:Theorem Prover,http://research.microsoft.com/en-us/um/redmond/projects/z3/, Microsoft Research, 2012
13) H8/3687 グループハードウェアマニュアル，ルネサス　エレクトロニクス，Rev.5.00，2005
14) H8S,H8/300 シリーズ　クロスアセンブラユーザーズマニュアル，Renesas Electronics ，Rev.6.00，1999
15) BYacc/J HomePage, http://byaccj.sourceforge.net/
16) JFlex - The Fast Scanner Generator for Java , http://jflex.de/

ⓒ 2012 Information Processing Society of Japan