Vol.2017-SE-195 No.25
2017/3/13

組込み制御Cプログラムのための リソース制約の簡易モデル検査手法の提案

鶴見 昂希[1]　上田 賀一[1]

概要：組込みシステムに要求される仕様として，一定時間内での処理の完了，限られたメモリ容量・メモリ性能での動作の確保があり，時間制約，メモリ制約を満たすことが重要である．また，ソフトウェアの潜在的な不具合のもたらすリスクの増大により，プログラムの信頼性向上が求められ，信頼性向上の手段としてモデル検査が有用である．本研究では組込み制御 C プログラムに対して検証の前処理として C プログラムを拡張することで制約の追加を行い，有界モデル検査ツール CBMC によってモデル検査を行うことで，厳しいリソース制約を満たしながら，プログラムの高い信頼性を実現する開発手法を提案する．

キーワード：モデル検査，CBMC，リソース制約

A Simple Model Checking Method of Resource Constraints for Embedded C Programs

Koki Tsurumi[1]　Yoshikazu Ueda[1]

Abstract: As specifications required for embedded systems, it is important to satisfy time constraints and memory constraints, such as completion of processing within a certain period of time, operation with limited memory capacity and memory performance. Also, due to the increased risk of potential problems of software, improvement of program reliability is required. Model checking is useful as a means of improving reliability. In this study, we add constraints to C program as preprocessing of verification for embedded control C program and perform model checking by CBMC bounded model checking tool. We propose a development method that realizes high reliability while satisfying stringent resource constraints.

Keywords: model checking,CBMC,resource constraint

1. はじめに

組込みシステムは急激な高機能化により，システムの大規模化・複雑化が進み，ソフトウェアの潜在的な不具合のもたらすリスクは増大している．組込みシステム開発ではシステムの安全性・信頼性の確保，厳しいリソース制約を満たすことが求められる．組込みシステムに要求される仕様として，一定時間内での処理の完了，限られたメモリ容量・メモリ性能での動作の確保がある．

一定時間での処理の完了は，組込みシステムのリアルタイム性として知られる．組込みシステムにおけるリアルタイム性とは，ソフトウェアなどの組込みシステムがある入力を受けて，対応した出力を行う際に，出力内容だけでなく，入力から出力までに時間的制約を求められるという性質を指す．例えば，電車の自動列車停止装置のような車載安全装置の場合，規定時間内に停止処理が完了しなければ，前の列車への衝突や脱線が起こり得る．その他にもロボットの関節制御など，応答の遅れが故障に結びつくようなケースは多く存在していて，リアルタイム性の実現は重要である．

限られたメモリ容量・メモリ性能で動作が求められる理由として，組込みシステムに用いられるマイクロコン

[1] 茨城大学
Ibaraki University

ピュータの選択に強い関連があることが挙げられる．メモリ容量やメモリ性能を抑えることは，利用するマイクロコンピュータの性能を抑えることである．性能の低い，つまり安価で消費電力の低い適したマイクロコンピュータの選択は，組込みシステム開発のコスト削減につながる．本研究では，対象をメモリの動的な利用に限定し，RAM の容量決定の支援を目指す．このように時間制約，メモリ制約といったリソース制約を満たす組込みシステムを開発するためにはリソース制約の分析が必要となる．

プログラムの信頼性向上の手段として，モデル検査が用いられている．組込みシステムのソフトウェア開発言語としては，C 言語を用いられることが多く，C プログラムのモデル検査は，有界モデル検査ツール CBMC[1] を用いることが有効である．

本研究の目的は厳しいリソース制約を満たしながら，プログラムの高い信頼性を実現する開発手法の確立である．提案手法では，組込み制御 C プログラムに対して，検証の前処理として制約の追加を行う．時間制約はコードを静的解析し，取得した情報から実行に要する時間を制約として追加する．メモリ制約は，動的メモリ割り当て処理を制約として置き換える．その後，CBMC によってモデル検査を行い，リソース制約を満たしているか検証する．さらに，ある C プログラムに対し適用実験を行い，提案手法の妥当性を検討する．

2. 関連知識

2.1 モデル検査

モデル検査とは，システムが正しく動くことを検証する方法である．設計段階での適用が可能なので，モデル検査法を用いることによって設計のバグを早期に発見することができ，開発コストを削減することができる．また，システムを厳密に記述して検証するので，出来上がるシステムの信頼性を向上させることができる．一般にモデル検査法では，システムを状態遷移系（オートマトン）として記述，システムに要求する性質，主に動作仕様や検査項目を論理式で記述，状態遷移系が論理式を満たすことを示すことを通して検証を行う．モデル検査法では，状態遷移系の動きを計算機の上で模倣して，あらゆる場合を網羅的に調べることによって状態遷移系が論理式を満たすことを示す．

2.2 モデル検査ツール CBMC

CBMC は C および C++プログラム用の有界モデル検査ツールである．配列境界の検査（バッファオーバーフロー），ポインタの安全性の検証，ユーザ定義のアサーションの検証を行うことができる．プログラムを入力した場合，各変数が一度のみ代入される静的単一代入形式に変換，有界な有限状態線形に変換されるので，範囲外の不具合は検証できないが，範囲内の検査を完全に自動で行うことができる．CBMC は組込みソフトウェアを対象としていて，malloc や new を使用した動的メモリ割り当てもサポートしている．

2.3 CPU 実行時間とクロックサイクル

コンピュータのほとんどの演算やデータ転送は，クロックに同期して行われる．したがって，クロック 1 周期分の時間が実行時間の基本単位となる．クロック信号の周期を，クロックサイクル時間とよぶ．

$$\text{クロックサイクル時間 [sec]} = 1/\text{クロック周波数 [Hz]} \quad (1)$$

ある命令を実行するのに要したクロック数をクロックサイクル数とし，クロックサイクル数を N とおくと，プログラムの実行時間は N とクロックサイクル時間の積で表せる．

2.4 クロックサイクル数

プログラムおよび命令の実行時間にクロックサイクル数は比例の関係にある．あるアセンブリ言語のソースコードのクロックサイクル数の値は，各命令のクロックサイクル数の和となる．

$$\text{クロックサイクル数} = \sum \text{命令のクロックサイクル数} \quad (2)$$

命令のクロックサイクル数は，クロックサイクル時間やクロック周波数に依存するため，マイクロコントローラやマイクロプロセッサによって違いがある．例として，PIC(Peripheral Interface Controller) の命令を**表 1** に示す．表には本研究でも扱う代表的な命令のみを示す．これらのクロックサイクル数の値は最小値が示してあり，キャッシュミス，ミスアライメント等によってクロックカウントが大幅に増加する可能性がある．

表 1 PIC の命令一覧

命令	機能	クロックサイクル数
add	加算	1
and	論理積	1
mov	移動 格納	1
sub	減算	1
call	サブルーチンへジャンプ	2
ret	サブルーチンから戻る	2

3. 提案手法

リソース制約の簡易モデル検査の手順を**図 1** に示す．本手法は 5 つの手順から構成される．

(1) 検証対象のソースコードを得る．
(2) 検証対象プログラムのアセンブルを行う．アセンブリ言語のソースコードから命令数を取得し，クロックサイクル数を得る．

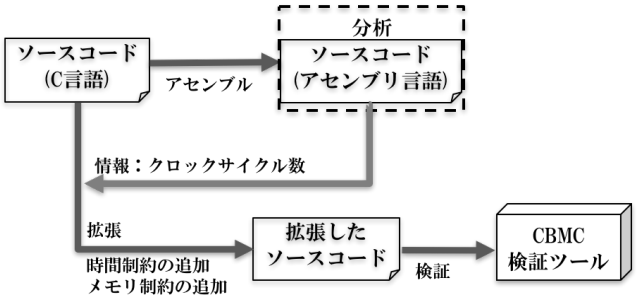

図 1 リソース制約の簡易モデル検査の手順

(3) クロックサイクル数に応じた処理時間計測コードを C ソースコードに拡張する．
(4) malloc，calloc，realloc，free をメモリ割り当て容量計測コードに変換，メモリ割り当て容量計測関数の拡張を行う．
(5) CBMC で検証を行う．

3.1 アセンブル

検証対象の C 言語のソースコードをアセンブリ言語に変換する．C 言語のソースコードを入力とし，アセンブリ言語のソースコードを出力とするツールをアセンブラと呼び，一連の変換処理をアセンブルと呼ぶ．アセンブラは GNU Compiler Collection のコンパイラオプション -S を用いることによって実現した．

3.2 アセンブリ言語のソースコードの分析

アセンブリ言語のソースコードは，add や mov のような CPU が直接実行する命令と，.byte や.set のような CPU が直接実行するわけではないけれども，プログラムの実行に必要となるデータの用意や，メモリ中におけるプログラムの配置をコントロールする命令で構成されている．

本提案では，プログラム実行時の処理時間は，add や mov などの CPU が直接実行するアセンブリ命令の処理時間と解釈する．各命令には，処理時間を計算する上で必要となる要素であるクロックサイクル数があり，クロックサイクル数は処理時間に比例するため，アセンブリ言語のソースコードからクロックサイクル数を得る必要がある．クロックサイクル数はマイクロコントローラおよびマイクロプロセッサに依存し，値が異なるため，本提案では表 1 に示すクロックサイクル数とする．各命令のクロックサイクル数の和を main 関数，外部関数，ループごとに計算する．

3.3 処理時間計測の拡張

C 言語のソースコードにクロックサイクル数を処理時間に見立て，計測する拡張を行う．拡張は 4 つの手順から構成される．

(1) クロックサイクル数を計算する変数をグローバル変数として宣言する
(2) 処理に応じたクロックサイクル数を実行ごとにグローバル変数に加算する
(3) assert によってクロックサイクル数を判定する
(4) 1 周期の終わりにグローバル変数を初期化する

3.4 動的メモリ割り当て

動的メモリ割り当ては，プログラムの実行中に並行して必要なメモリ領域を確保する，または開放するメモリ管理である．メモリの利用状況は，プログラムの実行状況に応じて常に変動するため，それらの処理に支障をきたさぬように必要なメモリ領域を適切なアドレスに対して確保および開放する必要がある．

組込みシステムでは，固定サイズブロック・アロケーションが有用であることが知られている．固定サイズブロック・アロケーションは，単純なアルゴリズムとして未使用メモリ領域をサイズごとに分類し，線形リストに繋いでスタックとして使用する手法がある．要求されたサイズと同じかひとまわり大きいブロックをデータに割り当てることで使用する．このような方法が単純な組込みシステムと好相性である．

C 言語の動的メモリ割り当ての方法として用いられる関数に，malloc 関数，calloc 関数，realloc 関数，free 関数がある．これらの関数に対して検証のための変換を行う．

3.4.1 データ型

動的メモリ割り当てで確保するメモリの大きさは確保したい変数のデータ型に依存する．データ型によって必要なビット幅，つまりバイト数が異なるため，データ型に応じて，検証のための変換を変える必要がある．**表 2** に，主要なデータ型とその型の使用に必要なバイト数を示す．

表 2 データ型のバイト数

型名	バイト数
char	1
short int	2
int	4
long int	4
float	4
double	8
long double	8

3.4.2 関数の変換

変換は 5 つの手順から構成される．

(1) 変数の宣言
動的メモリ割り当てで使用するメモリ量を計測した値を格納する変数の宣言を行う．動的メモリ割り当ての対象となる変数はポインタで宣言されているが，動的メモリ割り当てで使用するメモリ量を計測した値を格納する変数は整数型が好ましいため，変換を行う．
(2) 関数の変換

malloc，calloc，realloc はデータ型のバイト数と引数の積の分だけメモリを確保するため，型名と確保するメモリ量を把握する必要がある．そのため，確保するメモリ量を求める関数へと記述を変換する．確保するメモリ量を求める関数は，引数として char 型の型名と，int 型の確保するメモリの数をとる．

(3) 確保するメモリ量を求める関数の作成
引数を char 型の変数と，int 型の変数で，返り値を確保するメモリ量とする int 型の関数を作成する．関数内では，型名に応じた返り値を返す処理を記述する．型名を strcmp 関数で比較を行うため，ヘッダファイルの追加も同時に行う．

(4) assert によってメモリ確保量を検査する
CBMC で検証を行う際に，動的メモリ割り当てを行うメモリ量を検証するために，assert 関数を用いて判定する記述を加える．また，assert 関数を利用するためにヘッダファイルの追加を行う．

(5) メモリ解法のタイミングで，開放したメモリ分だけ変数の値を引く
free 関数を，開放したメモリの分だけメモリ確保量を表す変数から引く記述に変換する．

4. 適用実験

提案手法の妥当性を検証するために実験を行う．

4.1 実験環境

実験を行う環境を**表 3** に示す．

表 3 実験環境

実験環境	バージョン
OS	OS X Yosemite 10.10.5
gcc	4.9.3
CBMC	5.4

4.2 実験の目的

本研究の目的は厳しいリソース制約を満たしながら，プログラムの高い信頼性を実現する開発手法の確立である．本実験では，提案手法の手順に従い C 言語のソースコードに対してリソース制約を分析するための拡張を行い，CBMC を用いて検証することで，リソース制約とプログラムの信頼性を確保する．

本研究で扱うリソース制約は時間制約とメモリ制約である．組込みシステムにおける時間制約は，一定時間内に規定した処理が完了すること，つまりリアルタイム性があることである．リアルタイム性を実現する上での課題に，割り込み処理がある．割り込み処理とは，コンピュータが通常の流れでプログラムを実行している最中に，別のプログラムが割り込んで強制的に別のプログラムが実行されるような処理を指す．割り込み処理が行われるのは，主にリアルタイム処理の中で，実行時間の長いプログラムの実行中に他の即時処理が必要な場面である．このような場面では，優先度の低い長時間処理を一時停止させて，優先度の高い処理を先に実行させる．

4.3 実験 1

4.3.1 実験の概要

実験では，リアルタイム性および割り込み処理を実現するために，優先度の低い処理を for 文を用いて，優先度の高い割り込み処理を if 文を用いて記述する．for 文では 500 サイクルだけ処理が実行されるように記述し，毎サイクルで割り込み処理の実行の有無を判定，特定の条件で割り込み処理として簡単な演算関数が呼ばれる記述をした．また，メモリ制約として演算に用いる変数に対して，動的メモリ割り当てとして calloc 関数を用いて，演算回数だけ，つまりサイクル数である 500 個分のメモリを確保する記述をした．ソースコードに対して，手法による拡張を行い検証を行う．ソースコードをアセンブルすることで得られたクロックサイクル数を拡張に反映させて実験を行った．

4.3.2 実験の結果

拡張を行ったソースコードを CBMC で検証した結果，検証は正常値を示した．

4.4 実験 2

4.4.1 実験の拡張

次に，リソース制約，つまり assert 関数で検証する時間制約とメモリ制約の値を変えずにソースコードを拡張して実験を行う．ソースコードの拡張は，新たに動的メモリを確保して，簡単な演算関数を追加して行う．動的メモリの確保に realloc 関数を用いた記述を追加した．また，外部関数，およびそれを扱う割り込み処理を追加した．この拡張によって，各処理にかかるクロックサイクル数に変化が起きたため，クロックサイクル数の変化を反映させた．

4.4.2 実験の結果

拡張を行ったソースコードを CBMC で検証した結果，検証から不具合が存在することがわかった．出力された項目に Counterexample : 反例と Violated propertiy : 違反箇所がある．出力結果の一部を**図 2** に示す．

CBMC の出力結果から，違反は memory の値，つまり動的メモリ確保時の使用メモリ量が想定より多いことによる反例であるとわかる．この結果からメモリ使用量について再検討し，メモリ容量を上げる対処を行うべきであることがわかる．

4.5 実験 3

4.5.1 メモリ制約の緩和

結果をもとに，対処としてメモリの容量を上げる．つま

```
:
Counterexample:
:
State 220 file expeiment2-2.c line 22 function main thread 0
------------------------------------------------------
  memory=3200 (00000000000000000000110010000000)

Violated property:
  file expeiment2-2.c line 24 function main
  assertion memory < 3000
  !((signed long int)!(memory < 3000) != 0l)
```

図 2 CBMC による出力（一部）

り assert 関数で検証する値を設定し直して検証を行う．出力結果の一部を**図 3** に示す．結果として不具合が存在することがわかり，その不具合は systime，つまりクロックサイクル数の値であることがわかった．

```
:
Counterexample:
:
State 230 file expeiment3.c line 47 function twice thread 0
------------------------------------------------------
  systime=89 (00000000000000000000000001011001)
:
State 240 file expeiment3.c line 53 function square thread 0
------------------------------------------------------
  systime=99 (00000000000000000000000001100011)
:
Violated property:
  file expeiment3.c line 41 function main
  assertion systime < 50
  !((signed long int)!(systime < 50) != 0l)

VERIFICATION FAILED
```

図 3 メモリ制約を緩和したソースコードの CBMC による出力（一部）

4.5.2 時間制約の緩和

メモリ制約を緩和して検証を行った際の反例から，時間制約，つまりクロックサイクル数が違反していることがわかった．時間制約についても許容するサイクルクロック数，つまり systime 変数の assert による条件の緩和を行うことで反例に対処する．

4.5.3 考察

実験 2 において，CBMC による出力結果からは memory の値の違反，つまり動的メモリを確保する際に必要なメモリに関する assert 文のみに違反が検知され，反例も memory についてのみ出力された．実験 3 において，メモリ制約に関する assert 文のみ修正を行い，CBMC による検証を行ったところ，systime の値で違反が起きている結果が出力された．このことから，実験 2 の際潜在的に systime の assert 文は違反を起こしていたと考えられるが，通常の CBMC の検証では違反が起きた時点で検証を終了し結果を出力するため，潜在的な違反まではわからない．

このような場面では，CBMC のオプションを用いることが有効である．実験 2 における拡張を行ったソースコードに対して，CBMC のオプション–all-properties を加えて検証を行った．出力結果の一部を**図 4** に示す．

```
:
** Results:
:
[main.assertion.10] assertion memory < 3000: FAILED
[main.assertion.11] assertion systime < 50: FAILED

** 2 of 12 failed (2 iterations)
```

図 4 –all-properties オプションを加えた CBMC の検証による出力（一部）

出力結果から，assertion memory < 3000 と assertion systime < 50 の箇所で FAILED，不具合が存在することがわかった．このように–all-properties オプションを用いることで，全ての assert に対して検証結果を得ることができる．

4.6 実験 4

提案手法を用いて，時間制約，メモリ制約の確認についての実験を行い，期待した動作を得ることができた．実験 4 では，複数の規模の異なるソースコードを用意し，規模の異なるソースコードを検証した際の出力結果を比較する．実験に用いるソースコードは，外部関数を複数用意し，外部関数を使用する数の違いで規模の差を表現する．実験には 3 通りのコードを用いる．規模は 1 つ目のコードでは外部関数による割り込みを 3 種類，2 つ目のコードでは外部関数による割り込みを 6 種類，3 つ目のコードでは外部関数による割り込みを 12 種類記述した．また，外部関数についてはそれぞれ処理のクロックサイクル数が異なっていて，割り込み処理の発生頻度が低いものほど，処理が重いものとなっている．

それぞれのコードに対して，制約に違反せず正常終了した場合と，制約が違反し反例が出た場合の検証結果を得た．例として，割り込み数 3 の規模の小さいソースコードの検証成功時を**図 5**，検証失敗時の出力を**図 6** に示す．

4.6.1 分析

検証の結果を分析する．図 5 と図 6 を比較すると，反例の有無によって出力結果に違いが見受けられる．最初に違いが顕著に現れているのは，remaining after simplification の項目である．これは，Generated 1005 VCC(s) つまり 1005 個の検証条件が生成され，単純化後も残る検証条件の数を表している．図 5 の検証成功時の出力では，1 remaining after simplification となっているが，図 6 の検証失敗時の出力では，34 remaining after simplification が出力されている．検証時に反例がある場合，検証条件の単純化を行っても単純化されない検証条件が，反例が無い場合に比べて多いことが伺える．

```
:
size of program expression: 32197 steps
simple slicing removed 31726 assignments
Generated 1005 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
22345 variables, 2712 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 0.032s
VERIFICATION SUCCESSFUL
```

図 5 規模の小さいソースコードの CBMC の検証による検証成功時の出力（一部）

```
:
size of program expression: 32230 steps
simple slicing removed 327 assignments
Generated 1005 VCC(s), 34 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
135145 variables, 2723 clauses
SAT checker: instance is SATISFIABLE
Solving with MiniSAT 2.2.1 with simplifier
135145 variables, 0 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 0.658s
```

図 6 規模の小さいソースコードの CBMC の検証による検証失敗時の出力（一部）

次に出力結果に大きな違いが見られるのは，variables と clauses が出力されている部分であり，図 5 の検証成功時の出力では，22345 variables, 2712 clauses となっているが，図 6 の検証失敗時の出力では，135145 variables, 2723 clauses が出力されている．variables，つまり変数が検証失敗時の出力では約 6 倍となっている．この出力の差は前行の Solving with MiniSAT 2.2.1 with simplifier，つまり MiniSAT によって簡略化し解く際に，上記の様な差が生じたと考えられる．検証失敗時のほうが，変数の数が多いのは，前述した検証条件と相関が見られる可能性がある．この相関に関しては，規模の異なるソースコードに対する実験による結果から考察する．

SAT checker の出力について検証失敗時に違いが見られ，かつ，SAT checker と SAT checker inconsistent が存在する．SAT checker は instance が満たされていて，その後再度 MiniSAT による簡略化を通して，clauses が 0 となる．SAT checker inconsistent については，出力内容は同じである．

Runtime decision procedure でも大きな差が生じている．こちらは，検証成功時の出力では 0.032s，検証失敗時の出力では 0.658s と，20 倍程度の差が生じている．Runtime decision procedure についても，相関の有無を確認するために，規模の異なるソースコードに対する実験を行う．

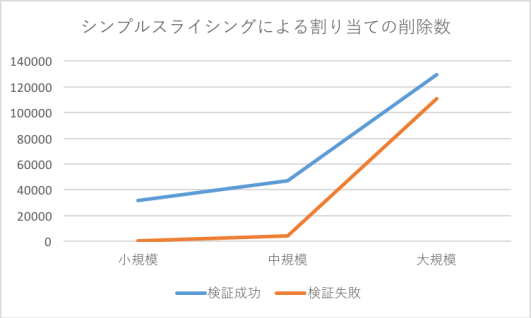

図 7 シンプルスライシングによる割り当ての削除数

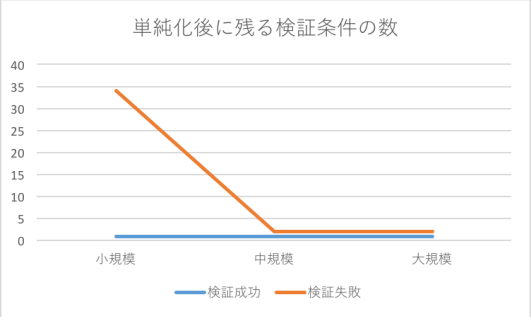

図 8 単純化後に残る検証条件の数

4.6.2 ソースコードの規模を変更した際の検証結果の比較

複数の規模の異なるソースコードに対する検証結果を検証成功および検証失敗についての検証を行った．それぞれの出力結果の値について比較を行う．比較を行うのは，シンプルスライシングによる割り当ての削除数，単純化後に残る検証条件の数，MiniSAT 簡略化後の変数の数，実行時決定手順に要する時間とする．シンプルスライシングによる割り当ての削除数は，検証対象を処理プログラムとして表現した場合のステップ数に対して，シンプルスライシングを行い，省略できる検証対象を削除する．削除後の処理プログラムは検証条件のみとなり，単純化できるものとできないものに分けられる．単純化できないものは単純化後に残る検証条件の数として示す．単純化できる検証条件に関して MiniSAT による簡略化を行い，そのときの変数の数，実行時決定手順に要する時間を示す．シンプルスライシングによる割り当ての削除数を**図 7** に，単純化後に残る検証条件の数を**図 8** に，MiniSAT 簡略化後の変数の数を**図 9** に，実行時決定手順に要する時間を**図 10** に示す．

4.6.3 考察

検証したソースコードの規模に相関が見られる出力結果は，図 7 に示したシンプルスライシングによる割り当ての削除数である．中規模に関しては比例関係とは言えない値を示しているが，規模との相関を考えると高い相関が見られる．その他の単純化後に残る検証条件の数，MiniSAT 簡略化後の変数の数，実行時決定手順に要する時間について

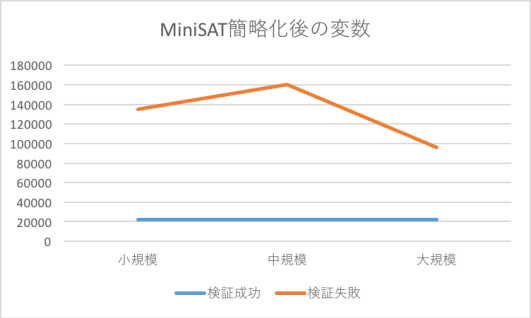

図 9　MiniSAT 簡略化後の変数

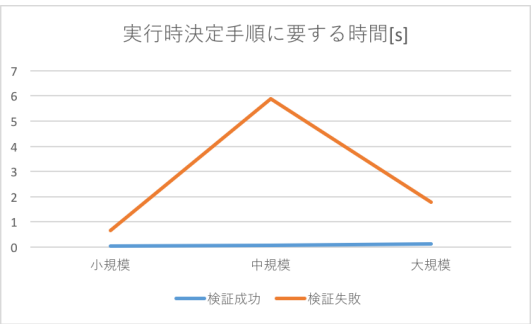

図 10　実行時決定手順に要する時間 [s]

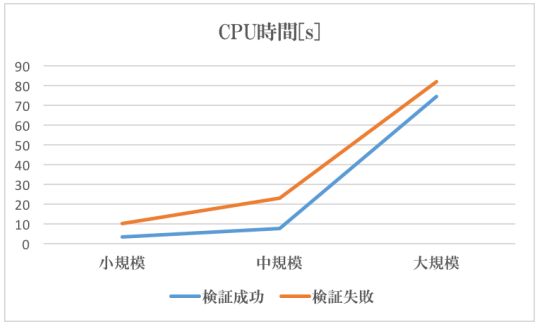

図 11　CPU 時間 [s]

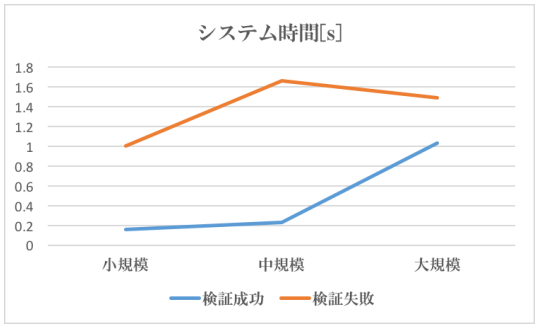

図 12　システム時間 [s]

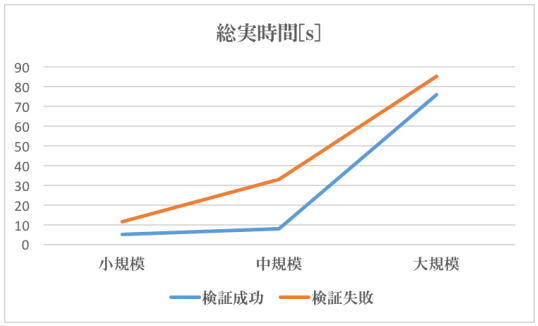

図 13　総実時間 [s]

は，強い相関は見られなかった．単純化後に残る検証条件の数に関しては，弱い負の相関が見られるが，中規模及び大規模の 2 つに注目したとき，検証の成功，失敗に関わらず差が見受けられないので，規模に依存した数値差ではないと考えられる．また，MiniSAT 簡略化後の変数の数，実行時決定手順に要する時間についても規模に依存した数値差は見受けられず，検証の成功，失敗の差であると考えられる．このことから，CBMC による検証は入力するソースコードの規模に関わる処理は少ない．

4.7　規模の差による検証時間の比較

ここでは検証に要する時間について示す．検証時間は time コマンドを用いて取得する．time コマンドはユーザ CPU 時間と呼ばれるコマンド自体の処理時間，システム時間と呼ばれるコマンドを処理するために OS が処理をした時間，コマンドによる CPU 占有率，プログラムの呼び出しから終了までにかかった実時間を出力するコマンドである．

実験 4 で行った検証それぞれに要した処理時間をグラフで示す．**図 11** に CPU 時間を，**図 12** にシステム時間を，**図 13** に総実時間を示す．

CPU 時間はソースコードの規模に対して強い相関が伺えるが，システム時間は相関がやや弱く見受けられる．しかし，システム時間自体が CPU 時間に比べて小さいものであるため，総実時間に大きな影響はなく，総実時間も CPU 時間と強い相関を示している．このことから，本検証はソースコードの規模を増やすことで，検証に要する時間は比例することがわかった．検証成功失敗の差としては，全ての規模において，全ての処理時間において検証失敗時のほうが所要時間が多かった．これは検証出力の比較において，検証失敗時は検証の簡略化が検証成功時より容易でなかったことが要因で，検証条件や検証手順が多かったのが原因と考えられる．

5.　関連研究

C 言語プログラムを対象としたモデル検査の研究として，橋本ら [2] による有界モデル検査法を用いた C プログラムのモジュラー検証，古川ら [3] によるモデル検査の反例を用いた C 言語プログラムの不具合修正方法の提案と実装がある．橋本ら [2] の方法では，SAT に基づく有界モデル検査法に Design by Contract の考え方を取り入れ，Design by Contract に基づくモジュラ検証を実現することにより，モデル検査が直面するスケーラビリティの問題を解決する．C プログラムにコントラクトを付与することでプログラム

の信頼性の向上を図っている．本研究では CBMC による検証で信頼性の向上を図る．モデル検査の反例を用いた C 言語プログラムの不具合修正方法の提案と実装では，反例から具体的な修正案を出力するツールを開発している．本研究では，対象とする C 言語プログラムに不具合がないものとしていて，制約追加による不具合は修正案が必要となる内容ではない．また，ANSI-C で書かれたプログラムを対象とした Cordeiro らの研究 [4] がある．これは ANSI-C で書かれた組込みソフトウェアへの SMT ソルバの検証を適用するもので，CVC3，Boolector，Z3 のソルバを CBMC に統合する研究である．本研究では，対象をリソース制約に限定し，ソースコードの拡張を行っている．

組込みシステムの制約に関する研究として，山本ら [5] による静的実行時間予測ツール構築のためのフレームワーク，鷲見ら [6] による組込みシステムにおける外部環境の分析がある．山本ら [5] のフレームワークでは，各タスクの最悪実行時間を把握するツールの構築およびフレームワークの提案を行っている．本研究では，時間制約として C プログラムを拡張しモデル検査することで，違反の可能性を検証でき，信頼性を示すことができる．鷲見ら [6] の研究では，組込みシステムが対応しなければならない実世界の状態（外部環境）を定義し，影響を分析する方法を提案している．本研究では，外部環境ではなく，メモリ制約と時間制約に焦点を当てている．

6. おわりに

6.1 まとめ

本研究では，リソース制約を満たしながら，プログラムの高い信頼性を実現するための開発手法の提案を行った．提案手法では，リソース制約として時間制約，メモリ制約に焦点を当てた．時間制約に関しては検証対象プログラムをアセンブルし，アセンブリコードから命令を読み取ることで，検証対象プログラムの処理に対して重み付けを行い，時間制約として検証できるように拡張を行った．メモリ制約に関しては動的メモリ割り当て関数を変換する形で，ソースコードの拡張を行った．拡張を行ったソースコードを CBMC によって検証することでソースコードの信頼性を確認した．適用実験では，手法による検証結果の確認やソースコード規模による検証結果の変化について比較を行った．その結果，検証対象のソースコードの規模に応じて検証に要する時間が大きく，検証失敗時のほうが，検証処理の負荷が大きいことがわかった．

6.2 今後の課題

今後の研究課題として以下の点が挙げられる．

6.2.1 対応する制約の拡張

今回対象とした制約は，リソース制約のうち時間制約とメモリ制約である．メモリ制約については，動的メモリ割り当てのみを対象としていた．組込み分野でリソース制約を考える場合直接的な影響が大きいのはマイコンの選定である．マイコン選定には処理能力，動的メモリ，静的メモリのバランスを考える必要があるため，検証できる制約の拡張や精度向上が課題となる．

6.2.2 手法の自動化

開発手法の単純化は重要な要素である．本手法で示したアセンブル，時間制約の重み付け，コードの拡張，検証は操作が独立しているため，各々の手順を自動化し組み合わせることが今後の課題である．

参考文献

[1] The CBMC Homepage http://www.cprover.org/cbmc/ .
[2] 橋本祐介，中島震．有界モデル検査法を用いた C プログラムのモジュラー検証．情報処理学会論文誌 Vol.52 No.8 2422-2430(Aug.2011)
[3] 古川直樹，深海悟．モデル検査の反例を用いた C 言語プログラムの不具合修正方法の提案と実装．情報処理学会研究報告 Vol.2013-SE-179 No.11 2013/3/11
[4] Lucas Cordeiro,Bernd Fischer,Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. cs.SE 13 Jul.2009
[5] 山本啓二，石川裕，松井俊浩．静的実行時間予測ツール構築のためのフレームワーク．spa.jssst.or.jp/2005/papers/papers/spa2005-04-2.pdf .
[6] 鷲見毅, 平山雅之, 鵜林尚靖．組込みシステムにおける外部環境の分析．社団法人 情報処理学会 研究報告 2004/11/26
[7] GCC,the GNU Compiler Collection https://gcc.gnu.org/ .
[8] 情報通信工学実験 マイクロコンピュータ 実験 7 命令サイクル数と実効速度の確認 https://www.mlab.im.dendai.ac.jp/∼assist/PIC/experiment/7/ .
[9] CompSys2012-06.pdf http://www-tlab.math.ryukoku.ac.jp/∼takataka/course/CompSysII/CompSysII2012-06.pdf .
[10] 産業技術総合研究所システム検証研究センター．4 日で学ぶモデル検査初級編．NTS，2006