Vol.2012-CDS-5 No.27
Vol.2012-DCC-2 No.27
2012/10/18

車載 C プログラムの割込み競合の静的検出手法

稲森豊[†1]　　山田信幸[†2]

車載制御ソフトウェアの実装において高速応答性の確保等のために利用される割込み処理は，割込み競合（割込み処理に起因するデータ競合）を発生させる可能性があるため，その未然防止対策は必須である．そこで我々は，割込み競合の発生位置を漏れなく検出することを最重要な要件として掲げ，割込み競合の自動検出手法を開発した．検出漏れを防ぐ方策として，各処理の変数アクセス情報を基に割込み競合の発生可能性のある位置を網羅的に列挙した後，各位置について静的コード解析を用いて割込み競合の発生可能性を判定する．抽象解釈を利用した割込み禁止状態の判定，モデル検査器 SPIN を利用した実行パスの存在性の判定等，観点の異なる多段の判定により誤検出を抑制する．

Static Interrupt-Race Detection Method for C Program in Vehicle

YUTAKA INAMORI[†1]　NOBUYUKI YAMADA[†2]

Interrupt handlers are used in vehicle control programs for high responsiveness but are a possible cause of data races. The present paper describes a detection method for interrupt race conditions that produces no false negatives and a smaller number of false positives. The proposed method is characterized by a mechanism whereby the masses of false positives are sifted through using five types of static code analysis methods that are free from false negatives. One of these methods identifies possible interrupt states for every statement using abstract interpretation and determines the possibility of interruption in accessing a shared memory. Another of these methods uses the model checker SPIN to verify the non-existence of an execution path creating a race condition.

1. はじめに

車両運動を制御する車載ソフトウェアの実装では高速応答性の確保等のために割込み処理が用いられるが，割込み処理は割込み競合（割込み処理に起因するデータ競合）の原因となりうるため，高信頼性確保の観点から割込み競合の未然防止は不可欠である．割込み競合は特定の割込みが限られたタイミングで発生した場合にのみ起こる稀な挙動であり，テストで網羅的に確認するのは困難である．そのためデザインレビューやコードインスペクション，テストといった品質保証手段を多重的に配置し，開発プロセス全体で高い信頼性を確保している．

しかし近年, 車載ソフトウェアの大規模化や複雑化, ECU (Electronic Control Unit) 数の増加に伴い，品質保証に要する工数は増大しており，品質保証作業の効率化が求められている．ここで我々が着目したのは，プログラム実装後に行われるコードインスペクションである．同作業は割込み競合の発生有無を目で見て検証するものであるが，制御フローやアクセスメモリを細かく追う作業は難易度が高く，長時間を要する．

そこで，我々は割込み競合に関する検証作業を大幅に軽減するために，割込み競合を自動で検出手法を開発した．手法開発時に掲げた要件は以下の通りである．

- **割込み競合を漏れなく検出する**：割込み競合の原因をテスト工程以降に残さない
- **割込み競合の誤検出**[a]**を低減する**：誤検出位置の人手確認作業を軽減する
- **人手の介在を最小限とする**: 人手作業を軽減すると共に手作業によるミスを防ぐ

上記要件を満たすため，検出漏れのない静的コード解析を多段に組み，割込み競合の発生する可能性のある位置を徐々に絞り込む手法を開発した．解析手法には制御フロー解析，ポインタ解析，抽象解釈，モデル検査を用い，解析時間の短い手法から適用することにより，全体の解析時間短縮を図っている．

本論文の構成を示す．2 章でデータ競合検出の関連研究について静的解析手法を中心に述べる．3 章で本研究が扱う問題を定義し，4 章で解析手法の概要，5 章で解析手法の詳細を説明した後，6 章では開発したシステムを紹介するとともに，製品向けプログラムを用いて評価した結果を示し，実務上の有用性を確認する．

2. 関連研究

割込み競合を含め，データ競合の検出に関する研究は，動的解析（プログラム実行やシミュレーションを伴う手法）と静的解析（静的コード解析を用いる手法）とに分類される．動的解析（例えば[1], [2]）はプログラム中の欠陥をで

† (株)豊田中央研究所
TOYOTA CENTRAL R&D LABS., INC.
†† アイシン精機(株)
AISIN SEIKI CO., LTD.

a) 誤検出は割込み競合の起こり得ない箇所を誤って検出することであり，品質に影響を及ぼすものではない．

きる限り多く発見することに適している反面，検出漏れを完全に防ぐことは技術的に難しい．逆に，静的解析にはデータ競合を漏れなく検出する手法が多い反面，誤検出の多さに課題がある．以下に静的解析の研究例を挙げる．拡張 C 言語 nesC [3]では，共有メモリへのアクセスが atomic 文の指定する領域内に無い場合，コンパイル時に割込み競合の可能性のある位置として検出する．LOCKSMITH [4]は，マルチスレッドプログラムを対象に共有メモリへのアクセスがロックによって整合的に保護されているかを型推論により検証する．両者とも atomic 文やロックの有無を検証するが，別の手段による保護（例えば，モード変数によるアクセス制御）については解析を行わないため，誤検出が生じる．

3. 問題の定義

割込み競合の検出について，対象の範囲を示し，問題定義および用語説明を行う．

3.1 解析の対象

解析の対象は，単一プロセッサ上で OS 無しにシングルスレッドで動作する C プログラムである．

プログラムは処理の集合 P（1 つのメイン処理および任意個の割込み処理）で構成される．処理 p（$\in P$）は優先度 i（0 以上の整数，大きいほど優先度が高い）を持つ．便宜上，メイン処理の優先度を 0 とし，割込み処理の優先度は自然数とする．割込み処理は，自身より優先度の低い処理が動作中，かつ，割込み許可状態の時にのみ起動可能である．割込み禁止状態／許可状態（以降，割込み状態と呼ぶ）を決定する命令には割込み禁止命令，割込み許可命令，割込み許可レベル設定命令（引数として与えられた許可レベルより高い優先度の割込み処理のみに割込みを許可する命令）があるものとする．以降，これらの命令を割込み関連命令と呼ぶ．

処理間のデータ授受手段は大域変数を介した共有メモリへのアクセスのみとする．

また，C プログラムは goto 文および関数の再帰呼出しを含まないものとする．共に MISRA C [5]のルールに準拠しており，実用上問題はない．

3.2 割込み競合

割込み競合は，割込み処理とメイン処理，または，割込み処理同士によるデータ競合である．以下の**発生条件 A**，**B** が共に成立する挙動を割込み競合と定義する．

発生条件A　低優先関数が短時間の間に共有メモリに対して 2 回アクセスする間に，割込みが発生し，高優先関数が同一のメモリに対してアクセスする

発生条件B　上記のアクセスのうち少なくとも 1 つは write アクセスである

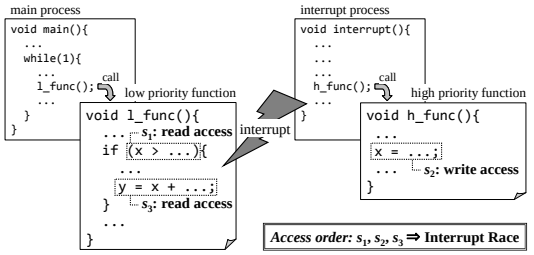

図 1　割込み競合の例

Figure 1　An example of interrupt race condition.

発生条件 A において，低優先関数は割り込まれる側の処理に割り当てられた関数，または，その関数から直接または間接的に呼び出される関数を指す．高優先関数は割り込む側の処理に割り当てられた関数，または，その関数から直接または間接的に呼び出される関数を指す．また，短時間の定義は「低優先関数の実行開始から終了までの時間」としている．同一のメモリの定義は，ビットフィールドの使用を前提とするため，「ビット単位で同一のメモリ」である．

図 1は割込み競合の例を示している．条件式$\boldsymbol{s_1}$で読み込んだ**x**の値が高優先関数の文$\boldsymbol{s_2}$で上書きされ，文$\boldsymbol{s_3}$では$\boldsymbol{s_1}$と異なる値を読み込むことになる．これが意図しない挙動であれば，割込み禁止命令でアクセス保護する等のプログラム修正が必要である．

3.3 割込み競合の事象系列

割込み競合の事象系列，または，割込み競合の可能性のある事象系列等と呼ぶ場合の「事象系列」を以下の 3 つ組で表現する．

- 低優先関数の一番目のアクセス事象 $\boldsymbol{e_1}$
- 高優先関数のアクセス事象 $\boldsymbol{e_2}$
- 低優先関数の二番目のアクセス事象 $\boldsymbol{e_3}$

アクセス事象 $\boldsymbol{e}$ は，アクセス位置 $\boldsymbol{s}$，アクセス変数 v，および，アクセス種類 a（READ または WRITE）の組で定義する．アクセス位置 $\boldsymbol{s}$ は，共有メモリへのアクセスを行う文または式（if 文や while 文の条件式）の位置を表し，その文または式が含まれる関数 f とその行番号 l，および，関数 f を呼び出す処理 p の組で定義する．この定義は[2]を参考にしている．

3.4 割込み競合の検出

本研究における割込み競合の検出とは，与えられた C プログラムが発生させる割込み競合の事象系列 $(\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3})$ を抜け漏れなく抽出することである．すなわち，割込み競合を引き起こす C プログラムを対象に解析した場合には，その割込み競合を発生させる事象系列が検出結果に必ず含まれること（検出漏れがないこと）が求められる．また，誤検出（割込み競合とならない事象系列が誤って検出され

ること）をできる限り少なくすることも求められる．

4. 割込み競合検出手法の概要

割込み競合を漏れなく検出するため，以下の 2 ステップから成る手法を開発した．

第 1 ステップでは，割込み競合の可能性のある事象系列を網羅的に列挙する．まず，各処理について大域変数にアクセスするアクセス事象を全て抽出する．次に，共通の大域変数にアクセスする処理の組 (p_1, p_2) を探し，割込み競合の可能性のある事象系列 $(\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3})$ を求める（$\boldsymbol{e_1}.v = \boldsymbol{e_2}.v = \boldsymbol{e_3}.v$, $\boldsymbol{e_1}.\boldsymbol{s}.p = \boldsymbol{e_3}.\boldsymbol{s}.p = p_1$, $\boldsymbol{e_2}.\boldsymbol{s}.p = p_2$）．その際，割込み競合の**発生条件A**に従い$\boldsymbol{e_1}$, $\boldsymbol{e_3}$は同一の低優先関数における事象でなければならない（$\boldsymbol{e_1}.\boldsymbol{s}.f = \boldsymbol{e_3}.\boldsymbol{s}.f$, $\boldsymbol{e_1}.l \leq \boldsymbol{e_3}.l$）．低優先関数におけるアクセス事象の列挙において，アクセス位置がループ文中にある可能性を想定し，$\boldsymbol{e_1} = \boldsymbol{e_3}$ の場合についても列挙する．ここで求めた事象系列のことを判定項目と呼ぶ．判定項目の網羅的な列挙により，最終的に判定項目の集合（割込み競合の判定項目リスト）が得られる（図 2）．

Check item No.	Shared variable	Low-Priority Function							High-Priority Function					Result (possibly race / never race)
		Function name	1st Access		3rd Access		Process		Function Name	2nd Access		Process		
			Line No.	R/W	Line No.	R/W	Name	Priority Level		Line No.	R/W	Name	Priority Level	
1	x	fun1	110	read	110	read	main	0	sub1	15	write	int1	3	?
2	x	fun1	110	read	113	read	main	0	sub1	15	write	int1	3	?
3	x	fun1	113	read	113	read	main	0	sub1	15	write	int1	3	?
4	y	sub1	53	write	53	write	int1	3	sub2	22	read	int2	5	?
...	...	...	...		...		...	...	...	...		...	...	...

図 2　割込み競合の判定項目リスト

Figure 2　A checklist of interrupt race conditions.

第 2 ステップで，各判定項目について複数の観点から割込み競合の発生可能性を判定し，割込み競合の発生し得ない判定項目を除いていく．そして，最終判定後に残った判定項目が割込み競合の可能性のある位置として検出される．具体的には，以下に示す**判定 1**～**判定 5** を順に行う．検出漏れを防ぐため，全ての判定は「割込み競合は発生し得ない」か「割込み競合の可能性がある」かのいずれかを判定し，「割込み競合の可能性がある」と判定された判定項目のみについて次の判定を行う．

判定1. **アクセスパターン判定**：
　アクセス種類についての条件を満たすか

判定2. **アクセス同時性判定**：
　低優先関数のアクセス位置 $\boldsymbol{s_1}, \boldsymbol{s_3}$ は同一の制御フロー上に在り得るか

判定3. **割込み禁止状態判定**：
　低優先関数のアクセス位置 $\boldsymbol{s_1}, \boldsymbol{s_3}$ は割込み禁止命令で保護されていないか

判定4. **メモリ同一性判定**：
　アクセス位置$\boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{s_3}$における共有変数へのアクセスはビット単位で同一か

判定5. **実行パス存在性判定**：
　割込み競合を発生させる実行パス（$\boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{s_3}$ の順の通過）は存在するか

全体の判定時間を短くするため，解析時間の短い判定から順に配置しており，ポインタ解析を用いる**判定 4**，および，モデル検査を用いる**判定 5** を後段に配置している．次章では，各判定の詳細を説明する．

5. 割込み競合検出手法の詳細

5.1 判定 1. アクセスパターン判定

割込み競合の発生条件のうちアクセス種類に関する条件（**発生条件 B**）の成立を判定する．すなわち，3 つのアクセス事象 $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}$ のうち少なくとも 1 つが write アクセスであれば（$\boldsymbol{e_1}.a = \text{WRITE} \vee \boldsymbol{e_2}.a = \text{WRITE} \vee \boldsymbol{e_3}.a = \text{WRITE}$），割込み競合の可能性があると判定する．同判定は判定項目の情報のみから判定できる．

5.2 判定 2. アクセス同時性判定

割込み競合の**発生条件 A** のうち，低優先関数における共有変数への 2 回のアクセスが短時間に発生する可能性，すなわち，低優先関数の 1 回の実行で 2 つのアクセス事象 $\boldsymbol{e_1}$ と $\boldsymbol{e_3}$ が共に発生する可能性を判定する．例えば，2 つのアクセス位置 $\boldsymbol{e_1}.\boldsymbol{s}$ と $\boldsymbol{e_3}.\boldsymbol{s}$ が分岐文の if 節と else 節に分かれる場合，「割込み競合は発生し得ない」と判定する．

具体的には，2 つのアクセス位置$\boldsymbol{e_1}.\boldsymbol{s}$と$\boldsymbol{e_3}.\boldsymbol{s}$を結ぶ実行パスの存在性を判定する．判定には制御フローグラフを用い，グラフ上での可到達性を判定する[b]．可到達性の判定に先立ち，各文についてその文に可到達な文の集合Rを求めておく[c]．そして，$\boldsymbol{e_1}.\boldsymbol{s}$から$\boldsymbol{e_3}.\boldsymbol{s}$への可到達性を判定する場合には，$\boldsymbol{e_3}.\boldsymbol{s}$に可到達な文の集合に$\boldsymbol{e_1}.\boldsymbol{s}$が含まれるか（$\boldsymbol{e_1}.\boldsymbol{s} \in \boldsymbol{e_3}.\boldsymbol{s}.R$）を調べればよい．以降，可到達性を以下のように表現する．

$$\boldsymbol{e_i}.\boldsymbol{s} \rightsquigarrow \boldsymbol{e_j}.\boldsymbol{s} \Leftrightarrow \boldsymbol{e_i}.\boldsymbol{s} \in \boldsymbol{e_j}.\boldsymbol{s}.R \qquad (1)$$

ループ文の存在を想定し，2 つのアクセス位置が同一の場合（$\boldsymbol{e_1}.\boldsymbol{s} = \boldsymbol{e_3}.\boldsymbol{s}$），同一のアクセス位置に戻る実行パスの存在性（$\boldsymbol{e_1}.\boldsymbol{s} \rightsquigarrow \boldsymbol{e_1}.\boldsymbol{s}$）を判定する．また，2 つのアクセス位置 $\boldsymbol{e_1}.\boldsymbol{s}, \boldsymbol{e_3}.\boldsymbol{s}$ が異なる（$\boldsymbol{e_1}.\boldsymbol{s} \neq \boldsymbol{e_3}.\boldsymbol{s}$）場合，$\boldsymbol{e_1}.\boldsymbol{s}$ から $\boldsymbol{e_3}.\boldsymbol{s}$ への実行パス（$\boldsymbol{e_1}.\boldsymbol{s} \rightsquigarrow \boldsymbol{e_3}.\boldsymbol{s}$），$\boldsymbol{e_3}.\boldsymbol{s}$ から $\boldsymbol{e_1}.\boldsymbol{s}$ への実行パス（$\boldsymbol{e_3}.\boldsymbol{s} \rightsquigarrow \boldsymbol{e_1}.\boldsymbol{s}$）のいずれかが存在することを判定する．

5.3 判定 3. 割込み禁止状態判定

5.3.1 考え方

割込み禁止状態判定は，低優先関数における 2 つのアクセス位置 $\boldsymbol{e_1}.\boldsymbol{s}$ と $\boldsymbol{e_3}.\boldsymbol{s}$ の間で常に割込み禁止状態であるかを判定する．判定結果が真であれば，$\boldsymbol{e_1}.\boldsymbol{s}$ と $\boldsymbol{e_3}.\boldsymbol{s}$ の間で割込みが発生しないため，割込み競合も発生し得ない．

同判定の実現に際し考慮すべき点を以下に挙げる．

b) 同判定では分岐文等の条件式の内容までは考慮していない．**判定 5** の実行パス存在性判定で分岐条件まで考慮した判定を行う．

c) 可到達な文の算出方法には様々なものあるが，我々は抽象解釈を用いている．可到達な文の集合を抽象領域として定義すれば，容易に算出できる（抽象解釈については 5.3 節を参照）．

a. 分岐文内に割込み関連命令が在る場合，実行パスによって割込み状態が変化する
b. ループ文内の割込み状態は，ループ文に入る直前の割込み状態のみならず，ループ文内の割込み関連命令に依存する
c. 割込み状態を規定する割込み関連命令は，実行中の関数内に在るとは限らず，関数呼出し元や呼出し先に在る場合がある

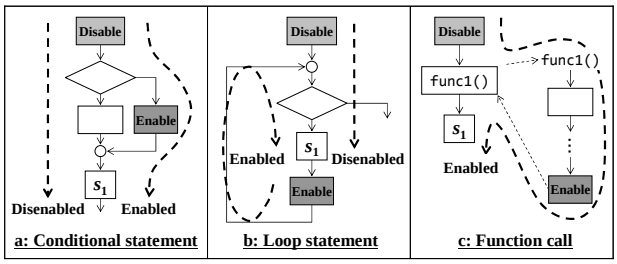

図 3 制御フローと割込み状態の関係

Figure 3 Interrupt states on various patterns of control flows

これらを考慮すると，実行パスの上流にある割込み関連命令を探索する方法では解析が複雑になることが予想されたため，本研究では抽象解釈を応用し，プログラム上の全ての文における割込み状態を特定する手法を開発した．抽象解釈[6]は静的コード解析の一種で，プログラムの実行状態を表す値と演算とを抽象化して実行し，実行結果から有益な情報を引き出す手法である．ここでは，各文の実行前と実行後に取り得る割込み状態に関して，抽象領域および抽象的な演算を定義する．

また c により，関数を跨いだプログラム解析が必要であるが，複数の位置からの関数呼出しや，関数ポインタによる呼出し関数の変化等，複雑なコールグラフに対応する必要がある．同様の研究が[7]に見られるが，本研究では複雑な解析を避けるため，関数単位で抽象解釈を行う方法を考案した．これらの方法を以下に示す．

5.3.2 割込み状態を表す抽象領域と演算

ある位置の割込み状態を表わす抽象領域 *ifg* を，

$$ifg = (low, high, inh) \tag{2}$$

と定義する．*low*と*high*は割込み状態の取り得る範囲を表し，それぞれ‘e’（enabled:割込み許可状態），‘d’（disenabled: 割込み禁止状態），nil（未定義値）のいずれかを取る．例えば，*low* = ‘e’, *high* = ‘d’ の場合，禁止状態と許可状態のいずれも取り得ることを表している．*inh*は割込み状態が関数呼出し元の状態に依存するか（inherited: 継承状態）を真偽値で表す. 例えば, 関数の先頭文の実行前は必ず*inh* = true である．*inh* = trueの場合，*low* ，*high*は実際の範囲を示さず，後述する適用演算(4)により関数呼出し元の割込み状態を加味する必要がある．抽象領域*ifg*は完備束を形成しており（図 4），有限の要素で構成されるため，有限ステップで解析できる．

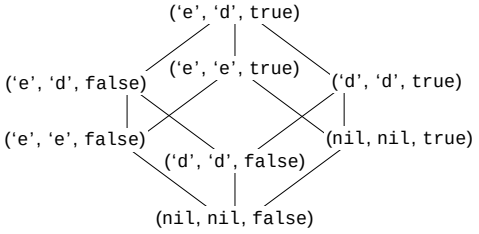

図 4 割込み状態の抽象領域

Figure 4 Abstract region of interrupt state.

まず，文の実行前の割込み状態（以降，実行前状態と呼ぶ）を特定する方法を示す．文の実行前状態は，以下の場合を除き，前文の実行後の割込み状態（以降，実行後状態と呼ぶ）に等しい．関数の先頭文の場合，上述のように実行前状態は (nil, nil, true) である．また，分岐文やループ文の合流位置にある文の場合の実行前状態は，合流直前の複数文の実行後状態についての以下の OR 演算で求める．

$$\begin{aligned}&(low_1, high_1, inh_1) OR_{ifg} (low_2, high_2, inh_2)\\ &= (\min_{ifg}(low_1, low_2), \max_{ifg}(high_1, high_2), inh_1 \vee inh_2)\end{aligned} \tag{3}$$

$\min_{ifg}$ と $\max_{ifg}$ は，‘e’＜‘d’ の関係を前提にした最小値および最大値の演算である．

次に，文の実行前状態から実行後状態を特定する方法を示す．まず，割込み禁止命令（許可命令）の実行後状態は実行前状態に関係なく禁止状態（許可状態）である．関数呼出しを含まない代入文または条件式の実行後状態は実行前状態に等しい．

関数呼出し文の場合，呼出し先関数の後尾文の実行後状態が自身の実行後状態であるため，呼出し先関数の解析結果を参照する．参照した割込み状態が *inh* = true の場合には，自身（関数呼出し文）の実行前状態を継承していることを意味するため，次の適用演算を行う．

$$\begin{aligned}&(low_1, high_1, true) \triangleleft_{ifg} (low_2, high_2, inh_2)\\ &= (low_1, high_1, false)\ OR_{ifg}\ (low_2, high_2, inh_2)\end{aligned} \tag{4}$$

左辺第 1 項は呼出し先関数の後尾文の割込み状態，第 2 項は関数呼出し文自身の実行前状態である．1 文内に複数の関数呼出しがある場合や関数ポインタにより複数の関数を呼び出す場合は，複数の関数に分岐し合流するものと見なし[d]，各関数の後尾文の実行後状態についてOR演算(3)を行った後，適用演算(4) を行う．

割込み許可レベル設定命令を解析する場合は，割込み許可レベルの取り得る範囲を *low* と *high* とし，$\min_{ipl}$ と $\max_{ipl}$

d) 例えば $z = f(x) + g(y)$; の場合，$f(x)$ と $g(y)$ の評価順序は不定のため，どの関数が最後に評価されるか不明であることを忠実に表している．関数ポインタの場合も同様である．

を割込み許可レベルの最大値および最小値の演算として定義すれば，同様に割込み許可レベルの取り得る範囲を特定することができる．

5.3.3 割込み禁止状態の判定

以上の方法により関数単位で各文の実行前状態および実行後状態を演算した後，判定項目の 2 つのアクセス事象 e_1 と e_3 の間の割込み状態を解析する．以後，アクセス位置の実行前状態を $e.s.ifg_{pre}$，実行後状態を $e.s.ifg_{post}$ と表現する．

アクセス事象 e_1 から e_3 までの間で常に割込み禁止であることを以下の条件で判定する．

- アクセス位置 $e_1.s$ の実行前状態が常に割込み禁止状態（$e_1.s.ifg_{pre}$ = (‘d’, ‘d’, `false`)），かつ，アクセス位置 $e_1.s$ と $e_3.s$ の間に割込み許可命令が無い

ただし，$e_1.s$の実行前状態について$e_1.s.ifg_{pre}.inh$ = `true`の場合, $e_1.s$の実行前状態は関数呼出し元の割込み状態に依存することを意味するため, 以下の算出により$e_1.s$の実行前状態を特定する．まず，$e_1.s$の属する関数$e_1.s.f$の関数呼出し元の集合を特定する．その際，e_1 は割り込まれる側の処理$e_1.s.p$の実行時に発生するアクセス事象であるため，ここで求める関数呼出し元の集合は$e_1.s.p$から実行可能な文とする．関数呼出し元の集合の特定は，関数コールグラフを用いて容易に行うことができる（図 5左）．そして，$e_1.s.f$の実行前の割込み状態を求める．先に得られた関数呼出し元が 1 つの場合はその実行前状態に等しく，複数の場合はそれらの実行前状態に対してOR演算(3)を行った結果に等しい．最後に，適用演算(4)により実際の割込み状態が特定できる（図 5右）．

また，条件「アクセス位置 $e_1.s$ と $e_3.s$ の間に割込み許可命令が無い」の判定は，$e_1.s \rightsquigarrow s \wedge s \rightsquigarrow e_3.s$ が成立する文 s に割込み許可命令が無く，かつ，$e_1.s \rightsquigarrow c \wedge c \rightsquigarrow e_3.s$ が成立する文 c に関数呼出し文がある場合は呼出し先関数（間接的な呼出し先を含め）に割込み許可命令がないことを判定すればよい．

アクセス事象 e_1 と e_3 の間に $e_1.s \rightsquigarrow e_3.s$ が成立する場合は前述の条件で判定し，$e_3.s \rightsquigarrow e_1.s$ が成立する場合は，前述の条件の e_1 と e_3 を入れ替えた条件で判定する．

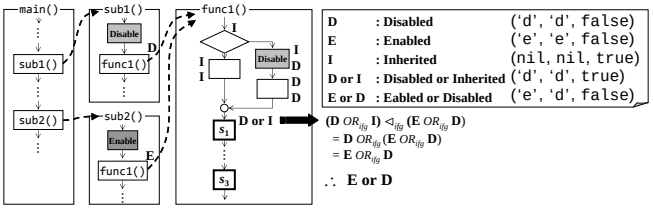

図 5　第 1 アクセス位置の実行前状態の算出
（継承状態が `true` の場合）

Figure 5　Calculation of the pre-executional interrupt state of the first access position (in case that ‘inh’ is `true`).

5.4 判定 4. メモリ同一性判定

低優先関数および高優先関数における 3 つのアクセス事象 e_1, e_2, e_3 がビット単位で同一のメモリにアクセスしているかを判定する．あらゆる式（ポインタ変数の加減算式，ポインタ型のキャスト等）について，アクセスする可能性のあるメモリ（大域変数名）とビット単位のオフセットを求めるため，ビット単位の情報を保持したメモリモデルを構築し，ビット単位でアクセス位置を特定可能な解析手法を開発している[e]．紙幅の関係上，詳細は割愛する．

5.5 判定 5. 実行パス存在性判定

割込み競合を発生させる実行パスの存在性を判定する．割込み競合の発生する実行パスは 3 つのアクセス位置 $e_1.s$, $e_2.s$, $e_3.s$ をこの順に通過するパスである．実行パスには高優先度の割込み処理の発生とそれに伴う低優先度の処理の中断が含まれる．

本研究では，あらゆるタイミングで割込みが発生した場合の割込み競合の有無を検証するために，並行プロセスのモデル化が容易なモデル検査器 SPIN [8]を用いて割込み処理の網羅的な振る舞いをモデル化する．スケーラビリティ対応のためにプログラムスライシングを行った後，スライスされた C プログラムを promela（SPIN のモデル記述言語）コードに自動で変換する．そして，割込み競合の実行パス検査用の assert 文を埋め込み，モデル検査を実行する．

以下の項では，多重割込みのモデル化とプログラムスライシングについて説明する．

5.5.1 多重割込みのモデル化

promela コードによる多重割込みのモデル化方法の概要を示す．

割込み処理を SPIN の並行プロセスとしてモデル化する．その際，割込み処理の以下の振舞いを正確に模擬する．

- 【メイン処理の起動条件】メイン処理は，プログラムの開始時に起動される
- 【メイン処理の動作条件】メイン処理は，割込み処理が実行されていない時に動作する
- 【割込み処理の起動条件】割込み処理は，自身より優先度の低い割込み処理（メイン処理を含む）が動作中で，かつ，割込み許可状態である時に起動することができる
- 【割込み処理の動作条件】割込み処理は，自身が動作中で，かつ，自身より優先度の高い割込み処理が起動していない時に動作する

promela では，並行プロセスの動作条件をプロセス定義文の provided 節に設定できるため，上記の起動条件および動作条件を provided 節で記述する．その際，割込み状態，

e) ただし，ポインタの再代入などのポインタ操作には対応していない．車載制御ソフトウェアでは複雑なポインタ操作の使用頻度が低いからである．

プロセスの実行状態は promela の大域変数（`interrupt_state` および `exec_[`割込み処理名`]`）で表す．割込み禁止命令（許可命令）は `interrupt_state = disenabled`（`interrupt_state = enabled`）で表す．割込み処理の内容は無限ループ内に埋め込み，ループの先頭でプロセスの状態を実行中（`exec_[`割込み処理名`] = true`）とし，ループの後尾で実行終了状態（`exec_[`割込み処理名`] = false`）とする．これにより，割込み処理が動作途中に割り込まれた側の処理に戻ることを防ぐ．また，多重割込みを模擬するためにスタックを設け，ループの先頭で自身の（割り込んだ）プロセスの情報をスタックに積み，後尾でスタックから下ろすことにより，割込み処理終了後に中断中の処理に制御を戻すことができる．

5.5.2 実行パス解析のためのプログラムスライシング

数万行規模の C プログラムを全て promela コードに変換した場合，状態空間の爆発により SPIN はメモリ不足で異常終了するため，実行パス解析の観点でプログラムスライシングを行い，コードを縮減する．

プログラムスライシングに求められるのは，アクセス位置の通過条件の保存に無関係なコードの削除である．以下に開発した基本アルゴリズムを示す（図 6）．

Step1 「未探索必要コード」および「探索済み必要コード」の初期集合を空とする

Step2 3 つのアクセス位置（$e_1.s$，$e_2.s$，$e_3.s$）および割込み関連命令文を必要コードとして「未探索必要コード」集合に加える

Step3 「未探索必要コード」集合から必要コードを 1 つ選び「未探索必要コード」から除く

Step4 関数先頭から必要コードに到達する際に必ず通る分岐文の条件式を求める

Step5 Step4 で得られた条件式に含まれる全ての変数について，変数への代入文を全て探索し，「探索済み必要コード」集合に含まれていない文を「未探索必要コード」に加える

Step6 Step5 で得られた代入文の右辺に現れる全ての変数について，変数への代入文を全て探索し，「探索済み必要コード」集合に含まれていない文を「未探索必要コード」に加える

Step7 Step4 で得られた条件式を「探索済み必要コード」の集合に加える

Step8 「探索済み必要コード」集合が空の場合 Step9 に進み，それ以外は Step3 に戻る

Step9 「探索済み必要コード」集合以外のコードを C プログラムから削除する

基本アルゴリズムは Step5 および Step6 で必要コードの連鎖を生じ易く，コードの縮減効果が低いため，基本アルゴリズムを改良している．改良アルゴリズムは，基本アルゴリズムの Step4 を以下の Step4'に変更したものである．

Step4' 関数先頭から必要コードに到達する際に必ず通る分岐文の条件式のうち，特定条件を満たすもののみ求める

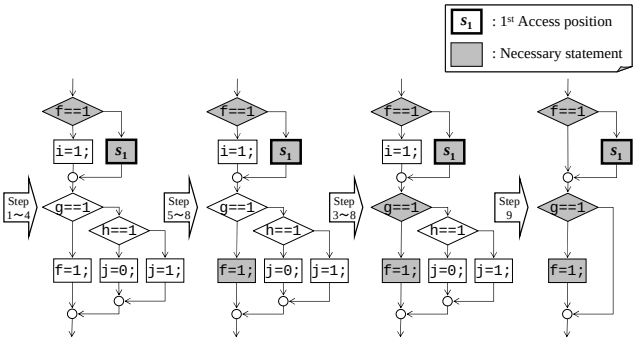

図 6　実行パス解析用プログラムのスライシング

Figure 6　Program slicing algorithm for execution path verification.

一部の条件式を必要コードから除外することにより，Step5 および Step6 における必要コードの連鎖を抑制することができる．ただし，除外された条件式は非決定的な分岐としてモデル化するため，本来はあり得ない実行パスが生じ（オーバー近似），誤検出の原因となる．現在採用している特定条件は「条件式に現れる変数がビット変数のみ」である．ビット変数は定数（0 または 1）の代入がほとんどであり，Step6 おける必要コードの連鎖を抑制できると同時に, promela コードに現れる変数がビット変数のみとなり，モデル検査の状態数を縮減できるからである．

6. システムの開発と評価

開発した割込み競合検出システムの構成を図 7に示す．解析の前処理部である構文木や制御フローグラフ，メモリモデルの生成，判定部の各種静的コード解析をRubyで実装した．モデル検査を行う部分についても，コードの生成，および，結果の解析をRubyで実装し，解析の本体部からSPINを起動する．Cプログラムの構文解析にはScientific Toolworks社のツールUnderstandが生成するデータベース（字句やシンボル，参照の種類や関係を格納）を利用している．

当システムは，開発当初から実務適用を目指して開発が進められ，製品向けCプログラムの 1 事例を代表的な評価事例として設定した. 事例の規模はソースコード数万行で，メイン処理および４つの割込み処理から構成される．開発目標として，検出漏れなし，自動判定率（当システムによ

る自動判定によって「割込み競合は発生し得ない」と判定された項目数が判定項目数全体に占める率）90% 以上[f], を掲げた.

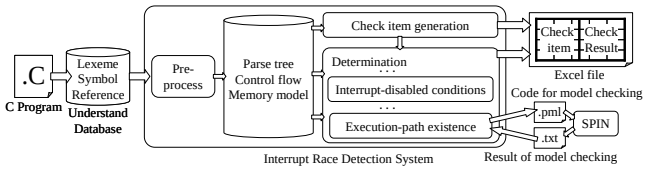

図 7　割込み競合検出システムの構成

Figure 7　Structure of Interrupt Race Detection System

開発当初から技術課題として挙がったのは，実行パス存在性判定（モデル検査を含む）のスケーラビリティ対応である. 数万行のプログラム全体をSPINにかけることは不可能で, 如何に状態数を抑えるかが鍵であった. そこで, 5.5.2項で示したスライシング手法の他，本論文では触れなかったが，実行パス探索に無関係な割込み処理および割込み関連命令を除去するアルゴリズムを実装している. その結果, コード行数を 20〜50 分の 1 に縮減でき, メモリ不足による異常終了を全体の約 15% にまで抑えることができた.

また，モデル検査による判定の回数を減らすため，その前段の判定, 特に割込み禁止状態判定の精度向上に努めた. 例えば，初期の割込み禁止状態判定アルゴリズムは以下の通りであった.

- アクセス位置 $e_1.s$ の実行前状態が常に割込み禁止状態, かつ，アクセス位置 $e_1.s$ と $e_3.s$ の間の全ての文の実行後状態も割込み禁止状態である

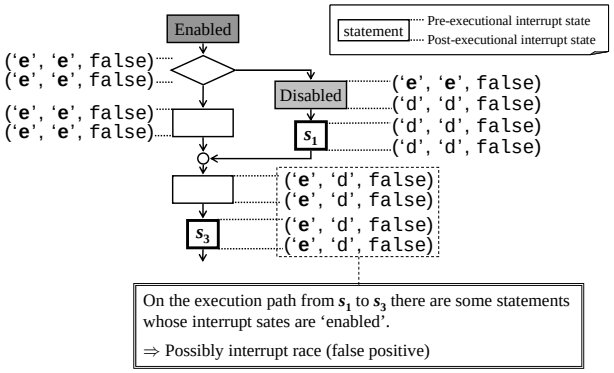

図 8　改良前の割込み禁止状態判定の誤検出例

Figure 8　An example of false positives in the determination of interrupt-disabled conditions (before improvement)

5.3.3項のアルゴリズムと同様，上記アルゴリズムも直感的には正しく，両者の精度は等価と思われた．しかし，事例の評価結果を詳細に分析した結果，に例示される誤検出の存在が判明した．抽象解釈で特定される割込み状態は，全実行パスを加味した場合に取り得る割込み状態であり，部分的なパス（図 8のs_1 からs_3 へ至るパス）の割込み状態を調べる場合にはs_1 とs_3 の間の割込み許可命令の有無で判定する5.3.3項のアルゴリズムの方が高精度であるとの結論に至っている. 以上のように精度向上に取り組んだ結果, 割込み関連命令を用いてアクセス保護を施した位置については誤検出がほぼ 0 となり[g], モデル検査の回数を大幅に削減できた.

評価事例による評価結果を以下に示す.

- 検出漏れ　　: なし （目標通り）
- 自動判定率　　: 94.3%（目標値以上）
- 人手の介在　　: 各処理の優先度等の情報設定，マイコン依存コードの修正のみ

自動判定率に関して,「割込み競合は発生し得ない」と正しく判定できなかった（誤検出の）原因の約 9 割はモデル検査のオーバー近似，または，メモリ不足による異常終了であった[h]. オーバー近似の回避にはスライシングにおける条件式の選定方法（5.5.2項のStep4'）の改良やCEGAR（Counter Example-Guided Abstraction Refinement; 反例による抽象化精錬）[9]の導入が考えられるが，モデルに組み込む変数の増加に伴い状態数が爆発的に増加してしまう．また，同事例よりも規模の大きいプログラムへの適用時には異常終了の率が上がることが予想されるため，モデル改良による更なる状態数の縮減や，問題の分割による状態爆発の回避が課題として残っている．参考のため，実施されたモデル検査の数例について状態数等のデータを表 1に示す.

表 1　モデル検査における状態数

Table 1　Number of states in model checking

Model checking No.	Line No. of promela code	No. of flag variable	No. of State
1	2559	1	about 9,000
2	2572	5	about 900,000
3	2659	7	about 1,000,000
4	2653	8	about 2,000,000

人手の介在に関しては，マイコン特有の命令やアセンブリコードの修正，削除のみで C プログラムを解析にかけられ，利用者の負担を軽減することができた.

f) 自動判定率が高い，すなわち,「競合の可能性あり」と判定される項目数が少ないと，人手による確認工数を縮減できる.

g) 判定区間の途中までを割込み禁止命令で割込み禁止とし，残りの区間を割込み許可レベル設定命令で割込み禁止にしている場合等で，5.3.3 項のアルゴリズムは誤検出を起こす.

h) 残りの原因はポインタ解析の精度不足であった.

7. さいごに

本論文では，C プログラムを対象に静的コード解析手法を応用して割込み競合の発生位置を検出する手法を示した．抽象解釈を用いた割込み状態の特定により，低優先関数における共有変数へのアクセス保護の有無を精度良く判定し，無保護のアクセスに対しては，モデル検査を用いて割込み競合となる実行パスの存在性を検証する．本手法は数万行規模の C プログラムに対して適用可能であり，1 事例ながら性能面での実用性が確認された．評価事例で十分性能が確認できたため，別の複数事例で評価する予定である．

車載ソフトウェアの大規模化に伴い，ソフトウェアインスペクションやテスト等の高信頼性を確保するための工数が増加の一途を辿る中，当手法は開発期間の短縮に貢献する見通しである．今後は視野をやや広げ，割込み競合を含めたタイミングに起因する問題に対して設計手法と解析手法の両面から解決する枠組みについて研究していく予定である．

参考文献

1) 荒堀喜貴, 権藤克彦: Cプログラムの割込み競合の動的検出法, 情報処理学会論文誌, Vol.51, No.9, pp1816-1831 (2010).
2) Higashi, M., Yamamoto, T. et al.: An Effective Method to Control Interrupt Handler for Data Race De- tection, *Proceedings of AST'10*, pp.79-86 (2010).
3) Gay, D., Levis, P. et al.: The nesC Language: A Holistic Approach to Networked Embedded Systems, *Proceedings of PLDI* (2003).
4) Pratikakis, P., Foster, J. et al.: LOCKSMITH: Con- text-Sensitive Correlation Analysis for Race Detec- tion, *Proceedings of PLDI'06*, pp. 320-331 (2006).
5) Motor Industry Software Reliability Association (MISRA): Guidelines for the Use of the C Language in Critical Systems (2004).
6) Cousot, P. and Cousot, R: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Pro- grams by Construction or Approximation of Fix- points, *Proceedings of the 4th ACM POPL*, pp. 238-252 (1977).
7) Sekiguchi, T.: A Practical Pointer Analysis for the C language, *Computer Software*, Vol. 21, No. 6, pp. 34-49 (2004).
8) Holzmann, G. J.: *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley (2004).
9) Clarke, E., Grumberg, O. et al.: Counterexample- Guided Abstraction Refinement, *Proceedings of CAV'00*, Vol. 1855 of LNCS, pp. 154-169 (2000).