自動車制御ソフトウェア開発プロセスへのモデル検査の適用

山口智也† 足立憲保†
加賀智之† 大桑芳宏†

近年、自動車制御システムに対する要求は益々高くなってきており、それを実現する制御ソフトウェアの大規模・複雑化が避けられない状況である。一方で、制御ソフトウェアの開発期間の短縮が望まれており、高い品質を維持しつつ、これに対応する為には、開発プロセスの効率化が必須である。筆者らはソフトウェア要求の検証にモデル検査を導入し、網羅的な検査を実施することで、開発プロセスを効率化することを目指している。

本稿では、自動車制御ソフトウェア開発プロセスへのモデル検査の適用に向けて、検証ツール開発の取り組み、モデル検査の自動車制御ソフトウェアへの適用事例とツールによるモデル検査の実施時間の低減効果について述べる。

A model checking application to software development of automobile control systems.

Tomoya Yamaguchi,† Noriyasu Adachi,† Tomoyuki Kaga† and Yoshihiro Ohkuwa†

Importance of software is increasing in automobile control systems and we expect software will be more massive and more complex, while shorter development period is desirable. Therefore, efficient software development becomes more important for keeping high quality control system. We improve quality of design specification by using model checking.

In this paper, we introduce a model checking support tool which we are developing, and we also show the result of a model checking application.

1. はじめに

近年、自動車制御システムは図 1 に示すように、エンジン制御、HV 制御、シャシ制御、ボディ制御、運転支援など、様々な機能や制御を求められ、著しく高度化、複雑化している。その制御システムの実現はハードウェアだけでなく、制御ソフトウェアによるところも大きく、自動車の制御システムを実現する上で制御ソフトウェアの重要性が増している。

図 2 に示すように、自動車の制御ソフトウェアの規模を表す一つの指標である ROM 容量の 1998 年から 2010 年までの推移は指数関数的な増加傾向にあり、今後も規模の増大が予想されている。

一方で、制御ソフトウェアは何重もの検査工程を経る必要があるため、更なる大規模化への対応には多大な工数が必要となることが予想される。現状の品質を確保しつつ、これに対応するためには、開発プロセス

図 1 自動車制御

の更なる効率化が必須である。筆者らはモデル検査技術を活用した新しい開発プロセスとそれを支援する検証ツールを開発し、品質を維持しつつ開発プロセスを効率化することを目指している。

2. 自動車制御ソフトウェア開発の現状と課題

2.1 自動車制御ソフトウェア開発プロセス

現状の自動車制御ソフトウェア開発プロセスの概要

† トヨタ自動車株式会社
Toyota Motor Corporation

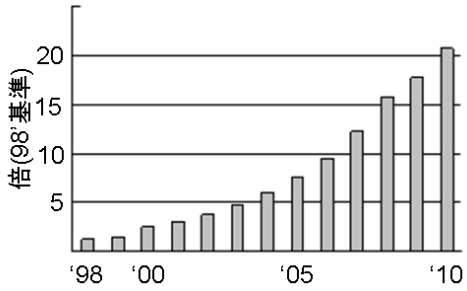

図 2 自動車 ECU の ROM 容量の推移

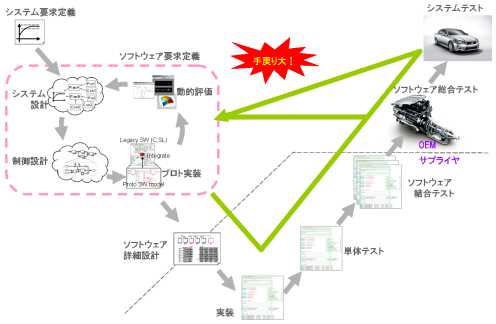

図 3 自動車制御ソフトウェア開発プロセス概要とその課題

を図 3 に示す。各プロセスの名称は独立行政法人 情報処理推進機構 作成の、組込みソフトウェア向け開発プロセスガイド[1] を参考に定義している。

自動車制御ソフトウェア開発プロセスは一般的なソフトウェア開発形態であるウォーターフォール型開発プロセスの 1 種である V 字開発プロセスをとっている。ただし、現実の物理現象を扱う制御ソフトウェアが開発対象であるので、試作システムを用いて、その仕様が実環境で動作することを確認しながら開発する、スパイラル型開発プロセスをソフトウェア要求定義工程に内包したものとなっている。

2.2 自動車制御ソフトウェア開発プロセスの課題

現状の開発プロセスは OEM、サプライヤにまたがる何十重もの検査工程により、システムテストまでに品質の確保を行っている。このプロセスの課題としては、図 3 のように手戻りの発生による開発効率の低下があげられる。この中で、手戻りの 8 割がソフトウェア要求定義を起因としたものであることが社内調査で判明しており、ソフトウェア要求の網羅的な検証が最も重点的に取り組むべき課題である。

2.3 ソフトウェア要求の網羅的検証への課題

自動車制御ソフトウェアの特徴は多数の入出力があり、条件分岐が多いことである。このような複雑なソフトウェアが要求を満たすことを網羅的に検査することが課題である。本稿ではこの網羅的な検査の為に、

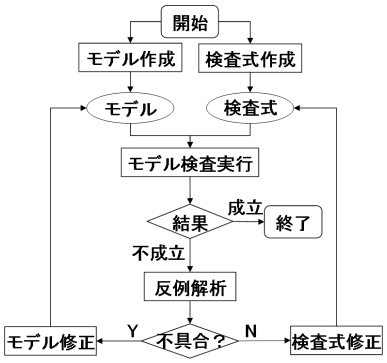

図 4 モデル検査作業フロー

近年、先進的検証技術として注目されているモデル検査を適用する。

3. モデル検査

3.1 モデル検査とは

モデル検査は形式的検証手法の一種であり[2)3)4)5)]、要求を論理式や時相論理式で表した検査式が、あるモデル上で成り立つことを網羅的探索により証明し、成り立たない場合はその事例の反例を示す。このように機能が満たすべき要求を検証することをプロパティ検証と呼び、モデル検査はこのプロパティ検証への有効な手法の一つである。この手法の最大の利点は網羅性である。検査式が証明された場合は、モデル上で検査式は静的に満たされることが保証されている。既にこの手法は半導体の論理回路の検証[6)] や高度なコーディングチェッカなどで適用事例がある。

実際にモデル検査を実行する場合の作業フローを図 4 に示す。これを実行するためには、まず、検査対象のモデルと満たすべき要求を論理式化した検査式が必要である。この二つを、モデル検査器に入力し検査結果を得る。結果が非成立の場合は、検査式が成り立たない反例が出力され、開発者は反例を解析し、それがモデル側の間違い、つまり不具合なのか、検査式が証明したい要求に対して不適切だったのかを判断し、そのどちらかを修正する。これを結果が成立、つまり要求が常にそのモデルに対して、成り立つことが証明されるまで繰り返す。

3.2 ソフトウェア要求工程へのモデル検査適用

モデル検査のソフトウェア要求工程への適用は、従来の開発プロセスに対して、図 5 のように、モデル検査を動的評価直前のプロト実装コードに適用することを検討している。

このモデル検査の開発プロセスへの適用の結果、以下の 2 つの効果が予想される。

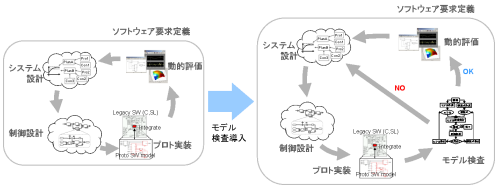

図 5 ソフトウェア要求工程へのモデル検査適用

・ソフトウェア要求定義の品質向上

不具合や仕様を規定した検査式を網羅的に検査することでソフトウェア要求定義の品質が向上する。

・ソフトウェア開発効率の向上

ソフトウェア要求工程内では、実施に時間のかかる動的検査の前に、網羅的な検査が可能なので、静的検査の段階で多くの不具合を検出することができ、工程内のサイクルの効率が向上する。また、ソフトウェア開発プロセス全体では、テスト工程を待たずにソフトウェア要求定義の品質向上が達成されるため、テスト工程からソフトウェア要求への手戻りが低減され、開発効率が向上する。

3.3 モデル検査の課題

モデル検査を開発プロセスに適用できれば前述のような効果が予想されるが、実際にはモデル検査作業フロー (図 4) 内の以下の部分で時間が必要であり、課題となっている。

(A) モデル作成

要件を検証するのに必要な情報を再現可能なモデルを作成することに課題がある。

(B) 検査式作成

開発者が検証したい要求を、適切に検査式にすることが本質的に難しく、課題がある。

(C) 反例解析

ソースコードが大きくなるほど反例は巨大になる。開発者がこのような巨大な反例を解析することに課題がある。

この 3 点の問題の中で、適切な検証式の設定が最も難しい課題であり重点的に取り組むべき課題である。筆者らは、このようなモデル検査の課題に対応するため要求定義検証ツールを開発中である。

4. 要求定義検証ツール

要求定義検証ツールは自動車制御ソフトウェア開発でのソフトウェア要求定義の品質向上のための静的解析ツールである。基本機能はモデル検査機能で、モデル検査の各プロセスを可視化技術や、構文解析技術を利用して支援する。このツールは、ソフトウェア要求定義内でのプロト実装コードを解析対象としている。

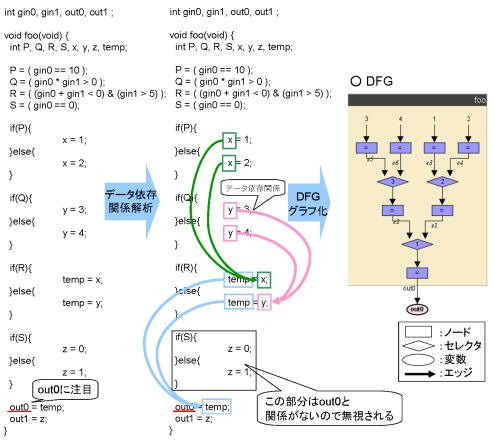

図 6 C ソースからの DFG 生成

4.1 ツール機能

4.1.1 モデル検査機能

C ソースからモデル検査を実施可能な CBMC[7] をモデル検査エンジンとして機能を実現している。

4.1.2 可視化機能

要求定義検証ツールは、モデル検査支援のためにコールグラフ、CFG(Control Flow Graph)、DFG(Data Flow Graph)[8] 等様々な可視化機能を有している。ここでは本ツールで特徴的な可視化形式である DFG について説明する。本ツールでの DFG は、ある一つの着目変数にデータ依存関係があるものをソースから抽出し、分岐を表すセレクタ、変数、関数、演算を表すノード、データ依存関係を表すエッジで構成されたグラフである (図 6)。

この形式の C ソースや CFG と比べ特徴的な点は、着目変数にデータ依存関係のあるもののみ抽出すること、グラフのエッジの意味がデータ依存関係であり、CFG の処理の順番を表すよりも、より本質的な意味をわかりやすく提示することである。

4.1.3 構文解析機能

C ソースコードを構文解析するのに一般的に用いられる、抽象構文木を内部で保持しており、これをもとに任意の構文を探索、追加、改変が可能である。

4.2 要求定義検証ツールのモデル検査支援

本ツールは前述の機能を応用して、モデル検査作業フローの支援する。概要を図 7 に示す。モデル作成と検査式作成では構文解析を用いて、反例解析では可視化機能を用いてモデル検査を支援する。また、定型的な検査については、要求定義検証ツールの様々な機能を組み合わせて、モデル作成と検査式作成を支援または、自動化する。

このツールのモデル検査作業フローへの適用を 5 章、

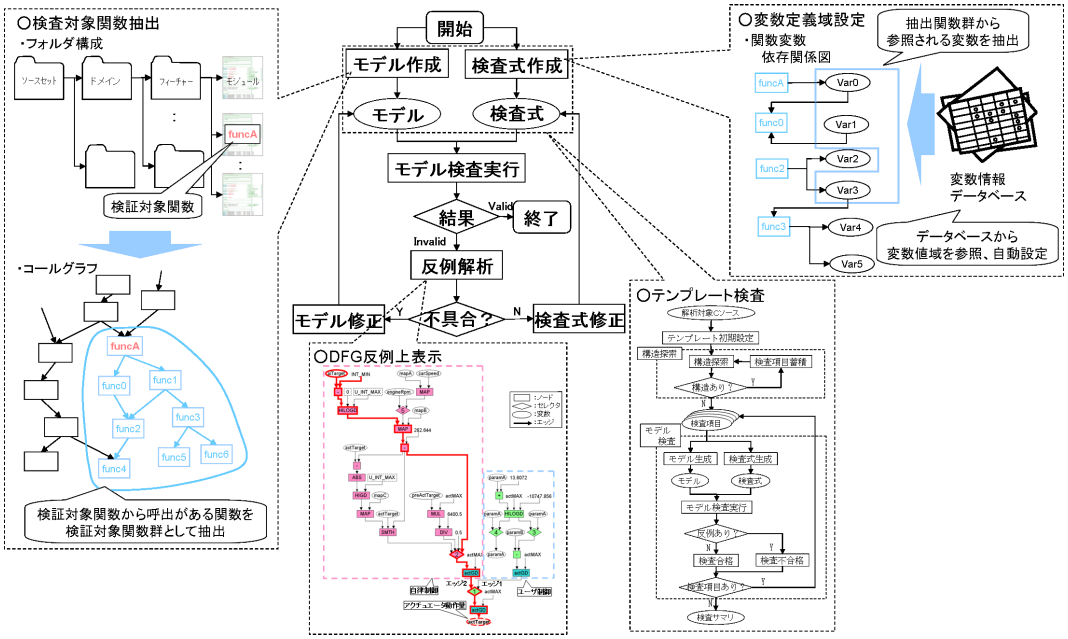

図 7　モデル検査作業フローへの要求定義検証ツールの支援

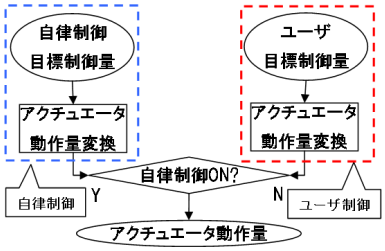

図 8　検証対象関数の概要

6 章にて実例を示しつつ述べる。

5.　要求定義検証ツールを利用したモデル検査

5.1　検証対象関数の説明

今回、具体例としてあげる関数は C 言語で実装されている自律制御に関する関数である。対象関数の概要を図 8 に示す。この関数はあるアクチュエータの動作量を算出する約 500 行規模の関数で、通常はユーザの指令より動作量を算出し、自律制御が実行されている間は、ここから要求される目標制御量より動作量を算出する。今回は、この関数に対して ”自律制御 ON のときアクチュエータが動作する ”ことを検証する。

5.2　検査対象関数抽出

～(A) モデル作成への支援～

モデル検査作業フロー (図 4) 中のモデル作成の効率化について述べる。モデルの作成については、前述のように CBMC を利用することによって、C ソースをモデルとして扱うことが可能なので、新たに検査用モデルを作成する必要がなく、既に従来に比べ作業が低減されている。ただし、システムの全ての C ソースを検証にかけることは、計算時間かかることや、反例が大規模になり解析が難しくなることから現実的でない。実際にモデル検査を実施するには、目的の検証を行うのに必要な最低限の C ソースを抽出する必要がある。この作業は以下の手順で実施する。

(1) 検証対象関数決定

検証対象となる関数を選択する。

(2) 検査対象関数群抽出

検証対象関数から呼出される関数を抽出する。

本ツールでは、(1) については開発者が判断することとし、(2) の部分を構文解析を用いて自動化しモデル作成時間を低減する。具体的には、図 7 の検査対象関数抽出のように、構文解析によりコールグラフを生成し、検証対象関数から呼び出される関数を自動で抽出することで検証作業を支援する。実際に開発者が手動でこの作業を実施することは、ソースファイルのフォルダ関係が関数の呼出関係と関連していないので、設定には時間が必要である。

5.3　定義域を設定すべき変数の抽出と自動設定

～(B) 検査式作成への支援～

モデル検査作業フロー (図 4) 中の検査式作成の効率化について述べる。適切な反例を得つつ、モデル検査を行う為には、検査式の一部にモデルに入力される変

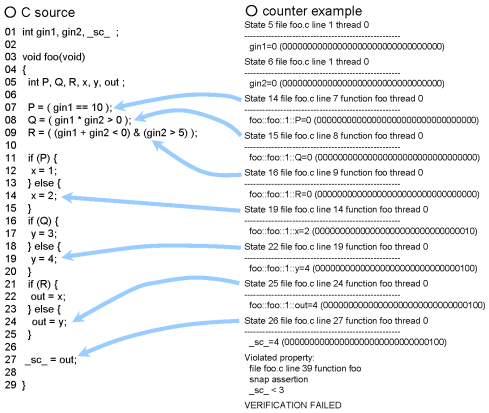

図 9　CBMC 出力の反例解析例

数の前提条件として定義域や入力変数間の拘束条件を検査式に与える必要がある。この作業は以下の手順で実施する。

(1) 検証要件検査式化

検証したい要件を検査式化する。

(2) 条件設定変数抽出

検証対象関数群から参照される変数を条件設定が必要な変数として抽出する。

(3) 変数条件設定

抽出された変数に対して、定義域と入力変数間の拘束条件を設定する。

本ツールでは、(1) については開発者が判断し作成することとしている。今回は ”自律制御 ON のときアクチュエータが動作する ”という要求から以下の検査式を作成した。

検査式： 自律制御フラグ=ON

$$\Rightarrow \text{アクチュエータ動作量} > 0$$

(2) に対しては定義域を設定すべき変数を自動でソースコード構文解析を用いて抽出し、(3) に対してはあらかじめ各変数の定義域を登録したデータベースから、定義域を自動で設定することで検証作業を支援することで検査式作成時間を低減する。この各変数の定義域等を記録するデータベースは、旧来の関数については仕様書から設定を抽出し構築し、新規関数については開発者が個別に変数情報を登録する。図 7 の変数定義域設定に概要を示す。

5.4　モデル検査実行と反例解析 ～(C) 反例解析への支援～

モデル検査作業フロー (図 4) 中の反例解析の効率化について述べる。先ほど作成したモデルと検査式をモデル検査器 CBMC に入力した。結果は検査式不成立であり、図 9 のように反例が出力された。反例は C

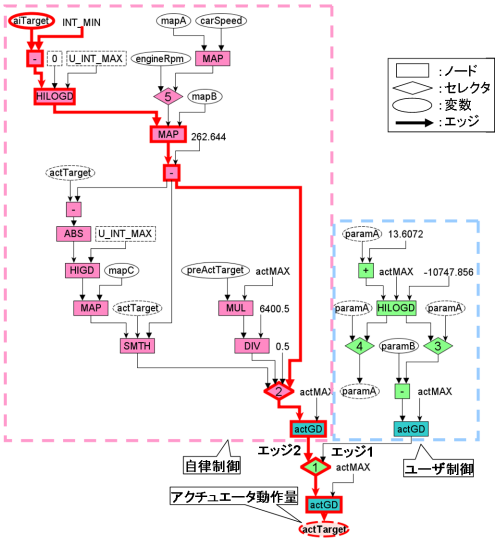

図 10　アクチュエータ動作量に着目した DFG と反例表示

ソースコード上での処理時の変数状態が行ごとにテキストで出力される。今回の検証対象の自律制御関数では、約 600 ステップ、2300 行という大きなテキストの反例が出力され、これを開発者がソースコードを追いつつ解析するには時間が必要である。

本ツールではこの対策として、ソースコード可視化機能の一つである、DFG と連携して反例を表示することが可能である。今回、着目すべき変数は検査式に含まれているアクチュエータ動作量であるので、これを着目変数とし DFG を生成した。さらに DFG 上に反例の実行経路と、各々の変数の値を表示したものを図 10 に示す。この DFG の利点は、ソフトウェアの複雑な処理を、着目変数に対するデータ依存関係の有無によって切り分け、開発者に理解しやすい形で提示することである。例えば、図 10 ではアクチュエータ動作量が、エッジ 1 からの経路であるユーザ制御側と、エッジ 2 からの経路である自律制御側を切り替えて算出していることが提示できている。さらにこの DFG 上で反例を表示させることで、今回の反例が自律制御側から現れ、この部分の修正が必要であることが短時間でわかり、モデル検査の効率的な実施が可能である。

5.5　低 減 効 果

要求定義検証ツールによるモデル検査機能を利用した場合と、このツールのモデル検査エンジンである CBMC を直接利用した場合の検査時間の比較を表 1 に示す。このツールの適用によってモデル検査作業フローの中で、モデル作成とモデル修正についてはコールグラフを用い検証対象関数群を自動で抽出すること

表 1 要求定義検証ツール利用によるプロパティ検証作業の低減時間

作業		CBMC 直接利用			要求定義検証ツール利用		
		実施方法	実施時間 (min)	実施回数 (回)	実施方法	実施時間 (min)	実施回数 (回)
モデル作成	検証対象関数決定	手動	5	1	手動	5	1
	検証対象関数群抽出	手動	25		自動	5	
検査式作成	検査要件検証式化	手動	4	1	手動	4	1
	条件設定変数抽出	手動	18		自動	3	
	変数条件設定	手動	18		自動	3	
モデル検査実行		自動	15	7	自動	15	5
反例解析	反例可視化	-	-	7	自動	5	5
	解析	手動	40		手動	10	
モデル修正	検証対象関数決定	手動	2	3	手動	2	2
	検証対象関数群抽出	手動	8		自動	5	
検査式修正	検査要件検証式化	手動	4	4	手動	4	3
	条件設定変数抽出	手動	8		自動	3	
	変数条件設定	手動	8		自動	3	
		565			214		

で効率化した。検証式作成と検証式修正については関数変数参照関係から拘束条件を設定すべき変数を、変数情報データベースから必要な定義域等の情報を引き出すことで、設定時間を低減した。反例解析については DFG 上に反例を表示することで、開発者の理解を促進し、解析を効率化した。これにより、全体の検査繰り返し回数も低減された。結果としてモデル検査作業フロー全体の実施時間が約 62%低減した。

6. 要求定義検証ツールによる定型的な検査の支援 ～(A) モデル作成と (B) 検査式作成への支援と自動化～

(A) モデル作成と (B) 検査式作成は、本ツールから前述の支援が可能であるが、要求定義検証ツールでは、さらに定型的なモデル検査について、あらかじめ適切な検査条件をテンプレートとしてツール側で用意し、その定型パターンに沿って検査を支援、または自動化することで、開発者のモデル作成や検査式作成の負担を軽減する。

6.1 テンプレート検査の概要

テンプレート検査の作業フローを図 11 に示す。この検査はあらかじめ決められた定型的な検査について、構文解析技術を用いて検査すべき構造を自動で探索、その後、その構造に対して、正常であることを証明可能な検査式を与えてモデル検査を実行するものである。

このような定型的な検査は、手動で行うと開発を重ねるごとに開発規模と検査項目が増大し検査に多大な時間を要する。テンプレート検査により一連の検証作業を支援または、自動化することで、高品質な制御ソフトウェアの開発を少ない工数で実現可能である。

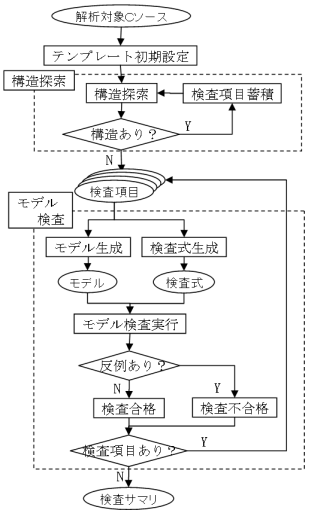

図 11 テンプレート検査作業フロー

6.2 テンプレート検査化検討項目

表 2 にこれまでの開発中の不具合を解析し、テンプレート検査化を検討している項目の一部を示す。この中で例として、上下限ガード値逆転防止検査と、デッドパス検査について以降に述べる。

6.3 上下限ガード値逆転防止テンプレート

上下限ガード値が逆転した場合、図 12 のように、開発者の予期しない動作が発生する。

表 2 テンプレート検査化検討項目の一部

No.	検査項目	検査概要
1	上下限ガード値逆転防止	上下限ガードの上下限値が逆転しないことを確認
2	デッドパス検査	デッドパスがないことを確認
3	関数出力変数範囲検査	関数の出力変数が、常に設定範囲内である
4	フラグ固着検査	全てのフラグ状態の成立性を証明
:	:	:

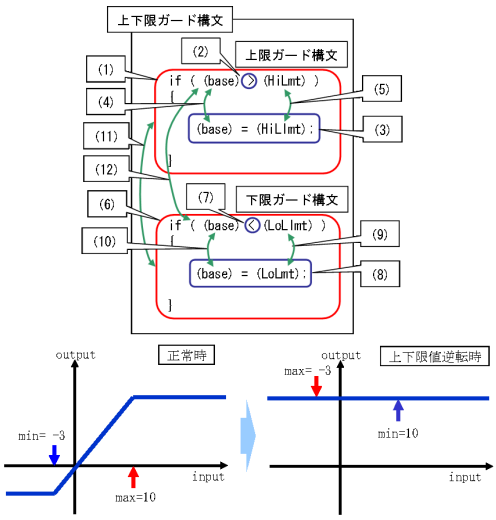

図 12 上下限ガード構文探索と
上下限ガード値逆転時のふるまい

このテンプレートは上限値と下限値の大小関係が逆転しないことを証明し、開発者の予期しない動作の発生を防止するものである。

このテンプレートの詳細を図 11 に基づき説明する。初めの構造探索では、不具合がある可能性がある上下限ガードを構文解析機能を利用し自動で抽出する。上下限ガードの構文抽出の手順をの条件を図 12 中の番号と共に以下に示す。

・上限ガード探索
(1) if 文のステートメント内である。
(2) if 文条件文が関係演算子 > である。
(3) if 文 then 命令文が代入演算子 = である。
(4) if 条件文左辺と if 文 then 文代入命令左辺が同一変数である。
(5) if 条件文右辺と if 文 then 文代入命令右辺が同一変数である。
・下限ガード探索
(6) if 文のステートメント内である。
(7) if 文条件文が関係演算子 < である。
(8) if 文 then 命令文が代入演算子 = である。
(9) if 条件文左辺と if 文 then 文代入命令左辺が同一変数である。
(10) if 条件文右辺と if 文 then 文代入命令右辺が同一変数である。
・上下限ガード探索
(11) 上限ガードと下限ガードの 2 つの構文が隣り合って存在する。
(12) 上限カード if 文条件文左辺と、下限ガード if 文条件文左辺が同一変数である。

次にモデル検査で、探索された上下限ガード全てに対して、常に上限値 ≥ 下限値が成り立つことを自動でモデルと検査式を自動で生成しつつ証明する。

実際にこのテンプレートを 5 章の自律制御関数に対して適用した。この結果、5 件の検査項目を抽出し、そのうちすべてが、常に上限値 ≥ 下限値が成り立つことが証明された。この検査を手動で実施した場合と、テンプレート検査を利用した場合の検査時間の比較を表 3 に示す。

テンプレート検査を利用した場合、上下限ガード抽出とその後の抽出された上下限ガードそれぞれに対する検証モデル作成を自動で生成することで実施時間が低減され、結果として手動での検査と比較して、約 80%の実施時間が低減した。

6.4 デッドパス検査テンプレート

デッドパス検査とはプログラムの構文上は存在するが、実際には通過することのないパスを検出するものである。開発者がそのデッドパスを意図せず作り出してしまっている場合は、確認項目として事前に検出されるべき項目である。しかし、ソースコード上のすべての実行パスを洗い出しデッドパス検査を行うことは、パス数が多く困難である。そこで、このデッドパス検査テンプレートでは、初めに全実行パスを洗い出すのではなく、本ツールの DFG 生成機能を利用して、比較的経路が似ている実行パス群ごとに分割して検査行う。

このテンプレートの詳細を図 11 に基づき説明する。まず、構造探索では、開発者に DFG 上の着目すべき変数を指定させ DFG を生成する。さらにこの DFG から、機能に対して重要な役割があり、状態を網羅的に分解して検査すべきセレクタを開発者に指定させ、そのセレクタを分解したサブ DFG を生成し、検査対象の実行パスとして取り出す。例として 5 章の図 10 で示した自律制御関数の DFG について、セレクタ 1、2、3 を分解して得られる、6 つのサブ DFG を図 13 に示す。

次にモデル検査では、ソースコード上の全ての分岐に対して、構文解析機能を用い、どちらのパスを通過したか判定する為の分岐判定フラグを挿入する (図

表 3 上下限ガード逆転防止テンプレート低減実施時間一覧

作業	手動検査 (min)	テンプレート利用 (min)
上下限ガード抽出	30	5
モデル作成	150	5
検査式設定	200	5
モデル検査実行	75	75
合計	455	90

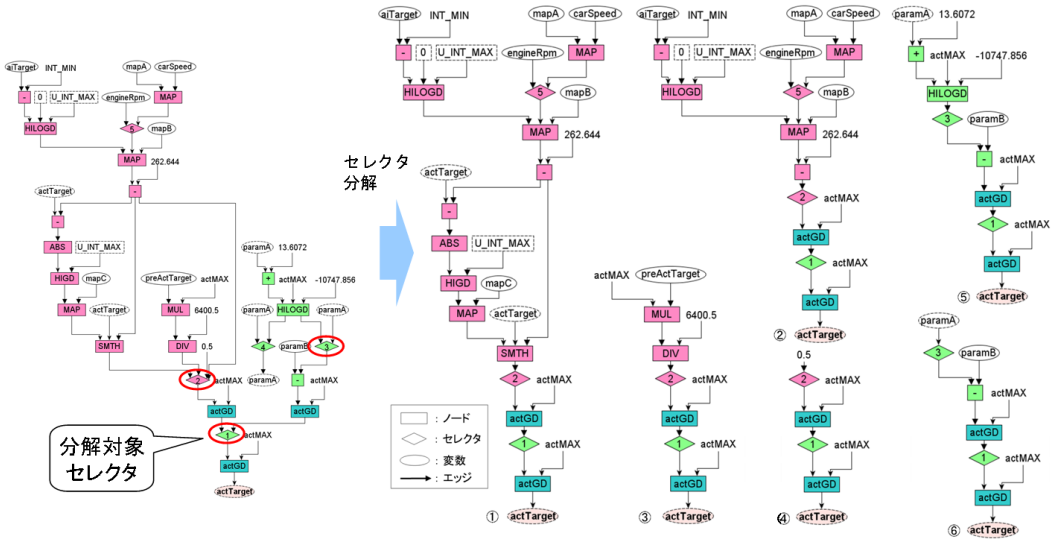

図 13　アクチュエータ動作量に着目した DFG のサブ DFG 分解

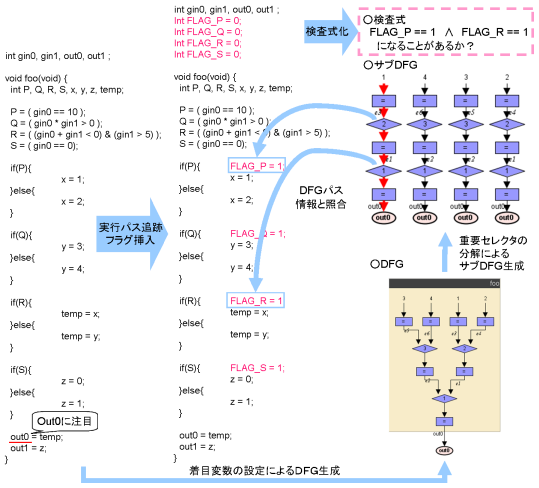

図 14　分岐へのフラグの挿入と検査式生成

14)。さらに、前述のサブ DFG からえた実行パスと対応する分岐判定フラグを抽出し、これが成立することを確認する検証式をモデル検査に与えて検査を実施する。以上の作業によって、開発者はこの観点を絞って生成した実行パスの中から成立しなかったものに対して、反例を重点的に仕様と確認することで効率的にデッドパスを検査を行うことが可能である。

実際に図 13 の自律制御関数に対してこの検査を行った結果、デッドパスとなるサブ DFG が 2 個見つかった。これについては仕様で問題がないことを確認した。この検査を手動で実施した場合と、テンプレート検査を利用した場合の検査時間の比較を表 4 に示す。

テンプレート検査を利用した場合、フラグのソースへの埋め込み作業や、その後のサブ DFG の実行パスに対応するフラグを検査式として自動で設定することで作業時間が低減され、結果として手動での検査と比較して、約 55%の実施時間が低減した。

7. ま と め

本稿では、今後の予想される自動車制御ソフトウェアの大規模化に対応するため、開発プロセスの効率化を行う必要を述べ、その達成にはソフトウェア要求の網羅的検証が最も重要であることを述べた。この検証の有効な技術としてモデル検査をあげ、その自動車制御ソフトウェアへの適用には、モデル検査作業フローのなかで、モデル作成、検査式作成、反例解析に課題があることを述べた。

それらの課題に対応する、筆者らが開発中の要求定義検証ツールを紹介し、そのプロパティ検証機能を用いることで、従来に比べて検査時間を約 62%低減可能であることを実例で示した。また、定型的な検査を支援、自動化するテンプレート検査を提案し、実施例として上下限ガード値逆転防止テンプレートとデッドパス検査テンプレートを紹介、前者については、従来の手動で行う検査と比べて、実施時間を約 80%、後者に

表 4　デッドパス検査テンプレート低減作業時間一覧

作業	手動検査 (min)	テンプレート利用 (min)
フラグ挿入	30	1
モデル作成	30	1
検査式作成	40	1
モデル検査実行	75	75
合計	175	78

ついては約 55%低減可能であることを示した。

これらのモデル検査作業フローの改善とテンプレート検査により、従来よりモデル検査の実施時間を低減したことで、自動車制御開発プロセスへのモデル検査適用が可能となった。

今後の課題は、モデル検査を実際に開発プロセスの適用し、開発プロセス全体でのモデル検査の適用効果を実証することである。

今後も、モデル検査等、先進的な検証技術を自動車制御ソフトウェア開発プロセスに適用することで、開発の効率化と品質確保の両立を進める。

参 考 文 献

1) 独立行政法人 情報処理推進機構 ソフトウェア・エンジニアリング・センター: 組込みソフトウェア向け開発プロセスガイド (2007)
2) 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 27(3), p24-35, (2010)
3) A.Biere, M.Heule, H.Van Maaren, T.Walsh: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications (2009).
4) 中島 震: SPIN モデル検査 検証モデリング技法, 近代科学社 (2008).
5) DR.GERARD J.HOLZMANN: The SPIN Model Checker, Addison-Wesley Professional(2003).
6) Limor Fix: Fifteen Years of Formal Property Verification in Intel:25 Years of Model Checking Lecture Notes in Computer Science (2008).
7) CBMC: http://www.cprover.org/cbmc
8) Tomoyuki Kaga, Masakazu Adachi, Ichiro Hosotani, Masaaki Konishi: Validation of Control Software Specification Using Design Interests Extraction and Model Checking, SAE 2012 World Congress & Exhibition(2012).