National Institute of
Advanced Industrial Science
and Technology
AIST

AIST-PS-2007-010

環境ドライバを用いたモデル検査による検証事例

高井 利憲* 古橋 隆宏** 尾崎 弘幸* 大崎人士*

* 独立行政法人 産業技術総合研究所
システム検証研究センター

** 矢崎総業株式会社 技術研究所

算譜科学研究速報
Programming Science
Technical Report

Research Center
for Verification and Semantics

独立行政法人 産業技術総合研究所 システム検証研究センター 発行
Published by Research Center for Verification and Semantics
National Institute of Advanced Industrial Science and Technology

環境ドライバを用いたモデル検査による検証事例

高井 利憲[**]　　古橋 隆宏[**†]　　尾崎 弘幸[*]　　大崎人士[*]

2007 年 10 月 1 日

概要

本稿では，組込みソフトウェアに対してモデル検査を用いた検証事例を紹介する．検査の対象は，周期駆動 (cyclic executive) 型の組込みシステムのソースコードである．コードは，MISRA-C のコーディング規約に準じて記述されている．検証実験では，既知のソフトウェアのバグを検出し，詳細原因を突き止めた．状態爆発を回避しながら，効率よいモデル化工程を実現するため，我々は，環境ドライバの概念にもとづく独自の手法を採用した．本稿で紹介する検証事例，考察および環境ドライバを用いたモデル化手法は，企業との共同研究を通じて得られたものである．

1　はじめに

組込みシステムが社会の至る所に入り込み，動作の安定が社会の安定を支えるようになるのに合わせて，組込みシステムの開発技術も急速に進歩してきた．近年では，計算機科学にもとづくシステム開発技術が発達し，長年の開発経験がなくても高度な設計技術を現場で活用することが可能になりつつある．しかし技術の進歩は，デバイスの集積化も加速させた．組込みシステムのソフトウェア規模が増大し，機能が複雑になることによって，職人的技術だけでは品質の確保が難しくなってきている．そこで，従来の開発技術を補う新技術として，形式的手法に対する期待が，現在急速に高まっている [5].

品質確保の問題だけではない．機能安全規格への対応という要因もある．国際規格 IEC61508[4] では，開発プロセス・ハードウェア・ソフトウェアの 3 つの観点から，対象製品の安全度水準 (SIL; Safety Integrity Level) を 4 つのレベルに分類する．最高レベルの SIL4 では 50 個を超える手法が「推奨」されており，形式的手法は「強く推奨」されている．ところが，実際の開発現場では未だ，形式手法が積極的に取り入れられているとは言い難い．テストやデバッグ工程の大部分は依然人手に頼っており，システム開発全体にかかるコストにも大きな影響を与えている．技術の進歩と現場との隔たりは，時間とともに少なくなりつつあるが，技術移転のための本質的な問題は置き去りにされたままという印象は否めない．

著者らの研究グループでは，共同研究のパートナー企業の協力のもと，組込みソフトウェアの開発現場に，モデル検査を中心とした数理的技法を導入するための実験と考察を繰り返してきた．本稿で紹介するモデル化技法は，限られた期間内でも効果的なモデル化工程を実現するための方法を模索する過程で得られた知見にもとづく．既存の開発工程を大幅に変更することなく，モデル検査をソフトウェア評価法として使うという趣旨でも，現場技術者のニーズに叶っている．

我々のモデル検査実験では，開発の上流工程および下流工程の両者への検査の適用を試みて，効果を比較検討した．その結果，ソースコードを検証対象とするモデル検査が，コストの点でも品質向上の点でも，最も効

[*][*](独) 産業技術総合研究所システム検証研究センター
[†][**] 矢崎総業 (株) 技術研究所

果的であるという結論を得た．上流工程でモデル検査を導入すると，設計や分析に使用する従来のツールまでも変更するという導入コストの問題，仕様書からモデルを作成するためにかかる人的コストの問題を解決できないという理由も大きい．

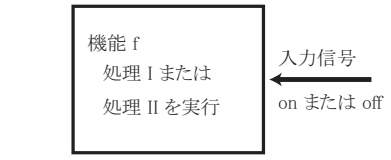
本稿で紹介する検証事例で採用した「環境ドライバ」を用いたモデル化手法は，コーディング規約 MISRA-C (Guidelines for the Use of the C Language in Vehicle Based Software) で開発される周期駆動型組込みシステムを対象としている．周期駆動型 (cyclic executive) 組込みシステム [1] とは，無限ループの構造を持つメインルーチンから，複数のタスクが定期的に呼び出されるようなシステムである．小型機器を制御するためのソフトウェアアーキテクチャとしてよく見られる．こうしたソフトウェアをコードレベルでモデル検査する際に問題となるのが，状態爆発である．従来のモデル化手法では，モデル化と抽象化を往復して状態爆発を回避しようとした．この方法では，検査可能なモデルを得るまでにかかる時間の見積もりが難しいため，開発に遅延をもたらすリスクが常ににつきまとう [12]．本手法では，現場の開発プロセスを乱すことなく効果的にモデル検査を行うことを目的とし，モデル化と抽象化の往復を極力減らすことを目指している．

ソースコードのモデル検査を行う際に，モデルの粒度にばらつきができるのは自然な現象である．逆に，対象全体をモデル化した後に，抽象化を施すと，モデル全体の粒度が下がるため，バグ原因が隠れてしまいやすくなる．また，必要部分のコードだけを抜き出してモデル検査することは難しく，必要部分に意味のある振る舞いをさせるための環境が不可欠である．我々のモデル化の方針では，詳細モデル (対象コードモデル) と抽象モデル (アーキテクチャモデル) を同時に作成して，組み合わせるというモデル化方針を採用している．結果的に，モデルの粒度にばらつきが生じる現象は著しくなる．また，この方針でモデル化を行うと，モデルは使い回しがきかない．したがって，まず検査項目を選別し，検査項目ごとにモデルの設計方針を検討する．一つのモデルであらゆる性質を調べるのではなく，検査したい機能やタスクだけを忠実にモデル化し，その他の外部要因は，検査対象タスクに影響がある副作用だけを「環境」としてモデル化する．検査対象タスク以外のものとしては，他のタスクや外部環境などがあり，これらのモデルを総合して，我々は**環境ドライバ**と呼ぶ．モデル検査が成功するか否かは，この環境ドライバの作り方に大きく依存する．

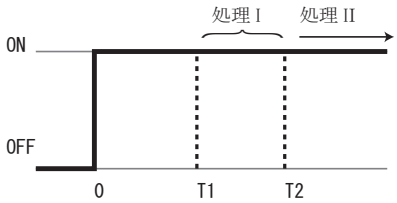
我々のモデル検査手順では，次の二段階のプロセスを踏む：(1) 検査対象タスクに対し，タスクが一回実行されたときの振る舞いを解析するための環境ドライバを作成して，検査を行う．ここでの検査は，**予備検査**と呼ばれる．この予備検査を通じて，タスク単体の機能の確からしさを検証するとともに，検査対象タスクのバグ原因の候補を列挙する．(2) 前段階で得られたバグ候補の真偽を確かめられる程度の環境ドライバを作成する．つまり最終的なモデルは，検査対象タスクの振る舞いについて，バグ候補を観察するのに必要最低限の機能だけを有する．この段階でモデルのサイズが大きすぎるようであれば，検査項目がモデル検査に向かないことになる．また，必要最低限の機能からなるモデルであれば，抽象化可能な部分はあまり多くない．

本稿で紹介する検査プロセスは，著者らが報告した別の事例論文で提案した周期タスク型モデル検査法 [6] と一致している．周期タスク型モデル検査法が，対象を特定して抽象化なども含めた詳細な方法論を展開しているのに対し，本稿では，より一般的なモデル化工程について述べる．また，周期タスク型モデル検査法においても二段階のモデル検査手順を踏むが，第一段階のモデル検査の役割として，本稿ではバグ候補の列挙を重視するのに対し，[6] の周期タスク型モデル検査法では，モデルの妥当性を確かめるための予備検査を行うためのモデルであることを強調している．著者らが報告している他の事例として，アセンブラ・プログラムへの適用例など [11, 12] があるが，本稿では，モデル検査プロセスを詳細に紹介し，環境ドライバという概念を導入する．

本稿の構成は以下の通りである．関連研究について解説した後，次節では，本稿で取り上げる検査対象の概要を説明する．第 3 節では，環境ドライバの概念とともに，我々が採用する検査手法について述べる．第 4 節



(1) 対象システム



(2) 入力信号に対する処理 I および II の実行タイミング

図 1　検証対象システムの概要

では，検査実験の一部について，その詳細を紹介する．本稿に載せた検査対象のソースコードなどは，実際の事例から変数名などを改変し，説明のため若干簡略化した．最後に考察を述べる．

関連研究

形式手法をソフトウェアの開発現場に適用した事例は他にもいくつか報告されているが，上流工程への適用が多い [2, 8, 9]．村石らの報告 [13] は，大規模な組込みソフトウェアに対するモデル検査事例である．テストでは発見困難と予想されるバグの検出にも成功している．ただし，もともとの開発工程で実行可能な UML が導入されており，モデル検査に使うモデルも UML から作成されている．自然言語で書かれた仕様書しか存在しない開発現場への適用は，対象外である．モデル検査をデバッグに応用する試みとして，篠崎らの報告 [10] がある．モデル化も含め，検証に要した作業時間の詳細な内訳も報告されている．本稿では，事例紹介に加え，状態爆発を避けるためのモデル化方針について，経験的な考察も述べる．

モデル検査の現場導入を考える際，モデル化以外にも，検査項目を LTL などの時相論理に翻訳する手続きを検討する必要がある．開発者らの心理的敷居は，モデル化よりも，むしろ LTL 検査式作成のほうが高く，モデル検査の現場導入に対する大きな阻害要因になっている．我々の研究グループでは，図示記法 [7] と呼ばれる LTL 式の図的な表現を独自に開発し，検査式作成の効率化も目指している．

2　検査対象

対象システムを A，検査対象機能である A の機能を f と呼ぶ．システム A の機能 f には既知のバグが含まれており，これを α と呼ぶことにする．システムの検証対象部分の要求仕様は以下の通りである: 機能 f は，一つの on/off の値をとる入力を持ち，連続した on の入力時間により，二つの処理 I と II を行う．具体的には，時間 T_1 以上 T_2 未満の間 on の値が連続すれば処理 I を行い，時間 T_2 以上 on が連続すれば処理 II を行う．ここで，実際の処理 I は，off になってからさらに時間 T_1 以上経過後に実行される．

システムはコーディング規約 MISRA-C を満たす C 言語で記述された組込みソフトウェアである．ソフトウェア・アーキテクチャは周期駆動型で，一定時間 (マイナサイクルと呼ぶ) ごとにいくつかのタスク呼び出

```
...
Task_f() {
  ...
}

main() {
  while (1) {
    Task_A();
    Task_B();
    ...
    Task_f();
    ...
  }
}
```

図 2　検査対象ソフトウェアの基本構造

しが行われる．今回検査対象とした機能 f に対応するタスクを Task_f とすると，ソフトウェアの基本的な構造は図 2 のように表せる．システム全体のソースコードは，数万行程度で，Task_f は，数百行程度である．ここで，マイナサイクルは，T_M とすると，各タスクに対応する呼出しは，T_M ごとに実行されるか否か決定される．タスクのスケジューリングはハードコーディングされているため，呼出しが実行されるタイミングが決まっており，整数型のカウンタ変数によって時間を扱える．例えば，入力信号が on になってから数え始めるカウンタ変数を用意することにより，その変数の値が T_1/T_M 回以上 T_2/T_M 回未満のときに入力信号が off になった場合に，処理 I を始めるということができる．

検査項目は，モデル検査実験時にはいくつかあったが，本項では以下の検査項目に対する検査プロセスを紹介する．

入力として T_1 以上 on が連続すると，
処理 I または処理 II のどちらか一方が
実行される　　(1)

3　検査手法

以降では，モデル検査ツールとして，Spin[3] を使用することを前提として，モデル検査の工程を説明する．我々が採用した検査工程は，これまでに繰り返し行われてきたモデル検査実験を通じて得られた知見を元に考案された．本稿で紹介するモデル検査実験の当初の目的は，その検査工程の有効性を確かめることであった．

3.1　環境ドライバ

モデル検査では，検査対象のシステムの振る舞いかたを網羅的に検証することが可能である．時間の経過とともに好ましくない振る舞いかたをするか，または想定外の状況に陥らないかについて調べることができる．このため，一部のソースコードだけを検査の対象とする場合でも，対象コード以外のシステムの部分もモデル化を行う必要がある．システムが，外部環境と情報をやりとりする場合には，外部環境のモデル化も行わなければならない (図 3 (a)) ．この際の外部環境は，忠実なモデルである必要はなく，例えば，取り得る値のすべてを非決定的に選択するようモデルであれば十分である．しかし，コードにあらわれる変数の取りうる値を忠実

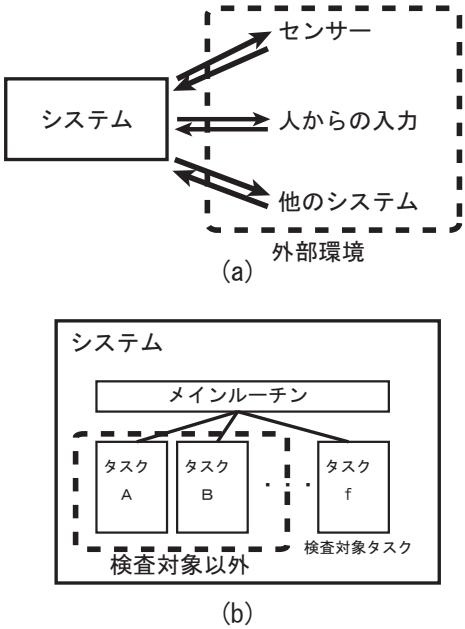


図 3　環境ドライバがモデル化する部分

にモデルとして表現すると，状態爆発が生じる．こうした問題を回避するためには，システムの振る舞いに影響を与える可能性のある値のみ (閾値) のみを，モデルの取りうる値とするなどの工夫を行う．また，検査対象部分を絞り込み，必要最小限の部分だけを取り上げる．(図 3 (b)) ．例えば，複数のタスクからなる組込みシステムの場合であれば，特定のタスクの性質のみを対象とし，他のタスクや，メインルーチンなどは，タスクに影響を及ぼす可能性のある副作用のみをモデル化する．

以上のように，外部環境のモデルや，検査対象以外のタスクのモデルを総合して「環境ドライバ」と呼ぶ．環境ドライバは，ソースコードに忠実なモデルである必要はなく，あくまで検査対象部分を動かすための環境にすぎない．検査対象部分と受け渡しする値は，モデル検査の網羅性を考慮した閾値周辺の値とし，大域変数への非決定的な代入としてモデル化する．取り得る値をすべて非決定的に選択可能にすると，偽反例が多くなり，検査の効率を著しく下げる．環境の振る舞いについては，検査に費やせるコストとのバランスを考えながら，受け渡す値がある一定時間同じであり続ける，疑似並行タスクの切り替えのタイミングを制限するなどの，想定 (シナリオ) を織り込む．常識的な制限をモデルに加えることにより，非常識な反例に煩わされる機会は減少するが，意外な反例は得られにくくなる．

環境ドライバの典型的な二つの例を図 4 に記す．いずれも，二値を取り得る外部信号が，大域変数 (`CHK_INPUT_ON`) の値 (`on` と `off`) として参照できる場合をモデルとして表現した環境ドライバである．ただし，(a) は一回だけの代入，(b) は任意のタイミングで任意回の代入を行う環境ドライバである．この入力信号を処理するタスクが `Task_f` の場合，メインルーチンのモデルは，例えば図 4 (a) に対しては図 5 (a)，図 4 (b) に対しては図 5 (b) のように記述する．`Task_f` 内部での入力信号の取り扱い方によっては，4 (a),(b) 以外の表現を取りうる場合もある．

状態数については，一般に，(a) の方が状態数は少なくなるが，外部環境からの入力が対象システムと独立であれば，(b) の方を採用する必要がある．今回の検査対象では，外部からの入力は，検査対象タスクが実行

```
unsigned CHK_INPUT_ON:uchar;
proctype drvINPUT () {
if
:: true -> CHK_INPUT_ON = 0
:: true -> CHK_INPUT_ON = 1
fi
}
```

(a)

```
unsigned CHK_INPUT_ON:uchar;
proctype drvINPUT () {
do
:: true -> CHK_INPUT_ON = 0
:: true -> CHK_INPUT_ON = 1
od
}
```

(b)

図 4　環境ドライバの例

する前に，すべて対応するポート (大域変数) に反映され，タスク実行中は，それらの変数の値が変わることがないという仕様であったため，図 4 (a) の環境ドライバを利用し，骨格構造としては，図 5 (a) に類する環境ドライバを利用した.

3.2　検査の流れ

本手法では，MISRA-C に準じた C 言語で記述された周期駆動型組込みシステムの特定のタスクの振る舞いを，モデル検査することを想定している．また，検査対象タスクは，C 言語上の関数 (呼出し) として表現されているものとする．以下，検査対象タスクに対応する関数を検査対象機能と呼ぶ.

本手法を用いた検査の流れは，以下の通りである:

1. 対象コードモデル (詳細モデル) の作成
2. ワンパスモデル用環境ドライバの作成
3. ワンパスモデルへのモデル検査適用
4. 周期モデル用環境ドライバの作成
5. 周期モデルへのモデル検査適用

1 の対象コードモデルの設計方針は，「ソースコードと同程度の精度のモデル」を目指す．同程度の精度のモデルとは，ソースコードの振舞いを保つように，C コードと Promela コードとの一対一の対応を保って変換することを意味する．ただし，検査対象タスクの振る舞いとは，変数同士の依存関係のないコードは，モデルの対象から除外できる．詳細について，以下順に説明する.

1. 検査対象機能を，忠実に対応する Promela 表現に変換する．MISRA-C に準じた C 言語は，Promela の制御構造と非常に似ており，文は代入，分岐，繰り返し，関数呼び出しからなる．つまり，C 言語の代入文は Promela の代入文に，分岐文は if 文に，ループ文は `do` 文にそれぞれ変換する．Promela では，関数呼出しを直接扱えないので，プロセスを用いて実現する．表 1 は，C 言語から Promela への変換例である.

```
active proctype main() {
  byte p;
  p = _nr_pr;
  run init(); p == _nr_pr;

  do
  :: true ->
     drv_INPUT(); p == _nr_pr;
     ...
     Task_f(); p == _nr_pr;
  od
}
```

(a)

```
active proctype main() {
  byte p;
  p = _nr_pr;
  run init(); p == _nr_pr;

  run drv_INPUT();
  do
  :: true ->
      ...
     Task_f(); p == _nr_pr;
  od
}
```

(b)

図 5　環境ドライバの使用例

両者を比較すると，プログラムの基本的な制御構造は同じであることが分かる．なお，この変換によって得られたモデルだけでは動作しないので，この時点では，シミュレーションによる動作確認はできない．

2. モデル検査を実施するコードに影響を及ぼす他のタスクや外部環境を抽出し，環境ドライバを作成する．1 で作成したタスクのモデルの基本的な振る舞いを見るために，タスクが一回だけ実行されるような環境ドライバを作成する．以下，他のタスクと外部環境をまとめて環境と呼ぶ．環境をモデル化する際，それらの処理過程をすべてモデル化する必要はない．出力として取り得る値を，非決定的に対応する大域変数に代入するモデルを作成する．

図 4 は，図 10 に示した Promela モデルのモデル検査を実施するための環境ドライバの一つである．変数 CHK_INPUT_ON は，入力信号の on/off のデータを格納する大域変数である．入力信号が時間とともに変化するのと同様に，格納されるデータも変化する．本来のシステムでは，入力信号に対して，ノイズ除去処理などを行い，その結果を変数 CHK_INPUT_ON に格納するが，今回の環境ドライバでは，そうした処理を省略し，単に 0 か 1 のどちらかが代入される仕様とした．代入される値は，モデル検査時には網羅的な組み合わせが選択される．また，シミュレーション時には，疑似非決定的に 0 か 1 が選択されるため，環境ドライバと検査対象機能のモデルを合わせることにより，ランダム・シミュレーション可能なモデルが得られる．以下では，

種類	C 言語	PROMELA
代入	`x = exp`	`x = exp`

分岐 1	`if (c) {`
	st1
	`} else {`
	st2
	`}`
		`if`
		`::c ->` *st1*
		`::else ->` *st2*
		`fi`

分岐 2	`if (c) {`
	st1
	`}`
		`if`
		`::c ->` *st1*
		`::else -> skip`
		`fi`

分岐 3	`switch (y) {`
	`case A:` *st1* `; break;`
	`case B:` *st2* `; break;`
	`...`
	`}`
		`if`
		`::(y==A) ->` *st1*
		`::else -> if ::(y==B) ->` *st2*
		`...`
		`fi;fi`

繰り返し 1	`for(;;) {`
	st
	`}`
		`do`
		`::true ->` *st*
		`od`

繰り返し 2	`for(`*exp1*`;c;`*exp2*`) {`
	st
	`}`
		exp1`;`
		`do`
		`::c ->` *st*`;` *exp2*
		`::else -> break`
		`od`

関数呼び出し	`foo()`	`run foo(); (p==_nr_pr) ->`

表 1　C 言語から PROMELA の変換例

このモデルをワンパスモデルと呼ぶ．ワンパスモデルのメインルーチンを図 6 に示す．初期化処理用プロセス (環境ドライバ) を `init`，その他の環境ドライバを `drv_A`, `drv_B`, ···，検査対象機能の詳細モデルを `Task_f` とする．

ワンパス用の環境ドライバは，検査対象タスクに影響を与える大域変数の値をすべて非決定的に選択できるよう作成する必要があるため，比較的作り込みには労力を要する．

3. 検査項目を LTL 論理式で記述し，2 で作成したワンパスモデルに対し，モデル検査を行う．ワンパスモデルでは扱えないシステムの振る舞いに関する検査項目もある．例えば，ループ一回の処理では完結しない応答性などの性質は，そのまま LTL 式で表したとしても，ワンパスモデルでは意図しない反例が出力される．こうした場合は，第 4 節で詳しく解説するが，検査式を工夫する必要がある．

ワンパスモデルのメインルーチンはループ構造でないことから，状態爆発は起きにくい．実行列も比較的単

```
active proctype main () {
  byte p;
  p = _nr_pr;
  /* initialize */
  run init (); (_nr_pr == p);
  /* drive environment drivers */
  run drv_A (); (_nr_pr == p);
  run drv_B (); (_nr_pr == p);
  ...

  /* run task */
  run Task_f (); (_nr_pr == p);
}
```

図 6　ワンパスモデルのメインルーチン

```
 1: active proctype main () {
 2:   byte p;
 3:   p = _nr_pr;
 4:   /* initialize */
 5:   run init (); (_nr_pr == p);
 6:   /* drive environment drivers */
 7:   run drv_A (); (_nr_pr == p);
 8:   run drv_B (); (_nr_pr == p);
 9:   ...
10:
11:   /* run task */
12:   do
13:   :: true ->
14:      /* drive environment drivers */
15:      run drv_a (); (_nr_pr == p);
16:      run drv_b (); (_nr_pr == p);
17:      ...
18:      /* run task */
19:      run Task_f (); (_nr_pr == p);
20:   od;
21: }
```

図 7　周期モデルのメインルーチン

純なため，検査結果を得やすく，投入する検査式の検討も容易である．ここで得られた反例は，変数に代入された値の組み合わせなど，実際には起こりえない状況を表している可能性があるが，設計者の想定から外れた状況によって発生するバグ候補が得られる機会でもある．ここで得られたバグ候補を列挙しておき，周期モデルでバグのあぶり出しを行う．

4. 検査対象タスクを周期的に起動する環境ドライバを作成する．ここで作成した環境ドライバを検査対象タスクのモデルをつなぎ合わせたものを周期モデルと呼ぶ．ここでは，3 で得られたバグ候補が，検査対象タスクが周期的に起動する場合でも起きうるか否かを確かめられる程度に詳細な環境ドライバを作成する．周期モデルをモデル検査すると，状態爆発が起こりやすいため，必要最小限のモデル化を行うために，ワンパスモデルに対するモデル検査で得られたバグ候補の情報を使うことが本手法の特徴である．周期モデルのメインルーチンを図 7 に示す．

5. 最後に，周期モデルに対して検査を行う．ワンパスモデルで反例が出たとしても，周期モデルで反例が出

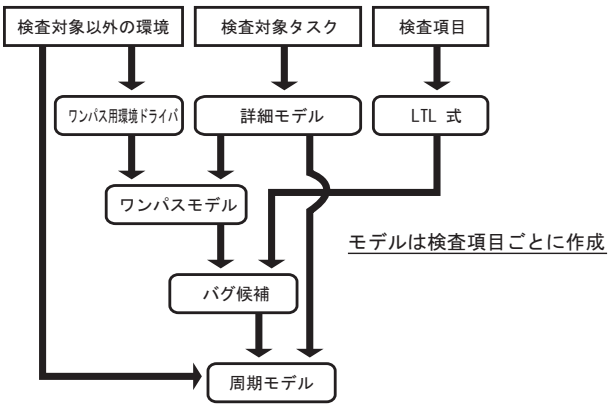


図 8　採用した検査プロセスの概要 (周期モデル作成まで)

るとは限らない．ワンパスモデルでは，大域変数の値をランダムに環境ドライバが格納する．したがって，繰り返しタスクを起動した場合には格納されないパターンで値を格納するかもしれない．すると，たとえワンパスモデルで反例が出たとしても，その状態が到達不能ならば，周期モデルでは反例が出ない．ゆえに，単純なワンパスモデルで反例が出た場合，その反例が周期モデルでも出るかどうかを確認することが重要である．また，ここで得られた反例もあくまでバグ候補であり，反例が表す実行パスを，実際のシステム上で実際に起きうるかどうかを判定する反例解析も行う．

4　検査事例

今回取り上げる検査対象システムのモデル化について説明する．前節で紹介した環境ドライバにもとづくモデル化手法に従い，検査対象機能 `Task_f` のモデル化を行う．関数 `Task_f` のコードの主要部分は，図 9 に示す．また，対応する部分を変換した PROMELA コード (詳細モデル) を図 10 に示す．この PROMELA コードは，表 1 の変換規則に従い，検査対象機能 `Task_f` の C ソースコードから得られたものである．

図 9 のコードをもとに検査対象機能 `Task_f` の機能説明を行う: 入力信号が on であるか調べ (1 行目)，on ならば 2 行目から 7 行目のコードが実行される．2 行目では，入力信号が on になってからの時間が T_2 を超えたか否かを調べ，超えていたら，変数 `FunctionFlags` に，処理 II を始めることを表すビットを立てる (5 行目) ．入力信号が off のときは (8 行目以降) ，まず入力信号が off になってからの時間を調べ，一定時間以上経過していなかったら (9 行目)，off になってからの時間を表すカウンタ `InputOffCnt` を増やす (10 行目) ．off になってから一定時間以上経過していた場合，on になってからの時間が T_1 を超えていたら (13 行目，17 行目) ，変数 `FunctionFlags` に処理 I を始めることを表すビットを立てる (14 行目，19 行目)．変数 `FunctionFlags` の初期値は 0，時間 T_1 に対応する定数は，`max_I_Cnt`，T_2 に対応する定数は，`max_II_Cnt` である．

実験の際に投入した検査項目のうち，本稿では，「入力信号が T_1 以上 on が続けば，処理 I または II のどちらかが必ず実行される」という検査項目についてのみを説明する．

検査の手順に従い，はじめに，ワンパスモデル用の環境ドライバを作成する．検査対象機能から変換されたプロセス `Task_f` の主要部分を参考にしながら，環境ドライバで非決定的な値の割り当てを行う変数を列挙す

```
 1: if (CHK_INPUT_ON) {
 2:   if (InputOnCnt < max_II_Cnt) {
 3:     InputOnCnt++;
 4:   } else {
 5:     FunctionFlags |= FUNC_II;
 6:   }
 7:   InputOffCnt = 0U;
 8: } else {
 9:   if (InputOffCnt < max_I_Cnt) {
10:     InputOffCnt++;
11:   } else {
12:     if(global_mode == MODE1) {
13:       if(InputOnCnt >= max_I_Cnt) {
14:         FunctionFlags |= FUNC_I;
15:       }
16:     } else {
17:       if ((max_I_Cnt <= InputOnCnt) &&
18:           (InputOnCnt < max_II_Cnt)) {
19:         FunctionFlags |= FUNC_I;
20:       }
21:     }
22:     InputOnCnt = 0U;
23:   }
24: }
```

図 9　検査対象機能 Task_f の主要部分

```
if
:: (CHK_INPUT_ON) ->
   if
   :: (InputOnCnt < max_II_Cnt) ->
      InputOnCnt++;
   :: else ->
      FunctionFlags = FunctionFlags | FUNC_II;
   fi;
   InputOffCnt = 0;
:: else ->
   if
   :: (InputOffCnt < max_I_Cnt) ->
      InputOffCnt++;
   :: else ->
      if
      :: (global_mode == MODE1) ->
         if
         :: (InputOnCnt >= max_I_Cnt) ->
            FunctionFlags = FunctionFlags | FUNC_I;
         :: else -> skip;
         fi;
       :: else ->
          if
          :: ((max_I_Cnt <= InputOnCnt) &&
              (InputOnCnt < max_II_Cnt)) ->
             FunctionFlags = FunctionFlags | FUNC_I;
          :: else -> skip;
          fi;
       fi;
       InputOnCnt = 0;
   fi;
fi;
```

図 10　Promela に変換された Task_f の主要部分

る．ここでは，以下の変数がそれに該当する:

```
CHK_INPUT_ON   global_mode
InputOnCnt   InputOffCnt
```

例えば，変数 CHK_INPUT_ON は，入力信号に対応するため，on,off の 2 値を非決定的に代入するプロセスを作成する．また，図 6 に示したメインルーチンを作成し，ワンパスモデルに対するモデル検査を実施する．ここでの検査項目は，次の通りである:

$$\begin{array}{l}\text{入力信号が } T1 \text{ 以上 on が続いた後}\\ \text{さらに } T_1 \text{ 以上 off が続けば,}\\ \text{必ず処理 I または II が実行される}\end{array} \tag{2}$$

本来，無限ループ構造をもつソフトウェアであっても，ワンパスモデルでは，1 回のループだけを再現する．このため，複数回以上のループ実行で真偽を判定する検査項目 (1) をそのまま検査に使用するのではなく，上記 (2) のように，タスクの入力時と出力時の変数の値に関する性質を問う検査項目に修正する．検査項目 (2) は，以下の検査式に翻訳される．

$$\Box(\mathtt{b} \rightarrow (\mathtt{p} \wedge \mathtt{q} \wedge \mathtt{s})) \rightarrow \Box(\mathtt{c} \rightarrow \mathtt{r})$$

ここで，各命題変数は以下のように定義されている．

```
#define b main@LABEL1  /* タスク f の実行前 */
#define s (CHK_INPUT_ON == false)
#define p (InputOnCnt >= max_I_Cnt)
#define q (InputOffCnt >= max_I_Cnt)
#define c main@LABEL2  /* タスク f の実行後 */
#define r (FunctionFlags != 0)
```

この検査式を用いて, モデル検査を実施すると, InputOnCnt がちょうど定数 max_II_Cnt で, CHK_INPUT_ON が off のとき，FunctionFlags が 0 のままである，という反例が得られる．すなわち，InputOnCnt が max_II_Cnt となるときが，バグ候補であると分かる．

次に，このバグ候補が，実際のソフトウェアで再現するかを調べるための周期モデル用環境ドライバを作成する．メインルーチンは図 7 の骨格を元にする．環境ドライバ内で，非決定的に値の割り当てを行う変数は，以下の 2 つだけである．

```
CHK_INPUT_ON   global_mode
```

変数 InputOnCnt および InputOffCnt の値は，初期化処理した後は，検査対象機能のモデルのなかで増減を行うため，環境ドライバ内で，値の操作を行う必要はない．

検査対象機能について要求仕様では，処理 I が，入力信号が T_1 以上 T_2 未満 on が続いた場合に実行され，処理 II が T_2 以上続いた場合に実行されると定めている．つまり，本来の検査項目 (1) は，要求仕様であると言える．この検査項目をモデル上の変数を用いた言葉で表すと，以下の通りである:

> 変数 InputOnCnt の値が，定数 max_I_Cnt 以上になれば，いつかは変数 FunctionFlags の値が 0 でなくなる

定数 InputOnCnt によって，入力信号が on になってからタスクが呼び出された回数が，経過時間に相当する実装になっているため，定数 max_I_Cnt は T_1 に対応する．また，処理 I または II は，変数 FunctionFlags

の特定のビットが 1 になっていることを参照して実行する．以上を考慮すると，検査項目は，以下の LTL 式に翻訳される:

$$\Box(\mathtt{p} \rightarrow \Diamond \mathtt{r})$$

ここで，p および r は次のように定義されている．

```
#define p InputOnCnt >= max_I_Cnt
#define r FunctionFlags != 0
```

上の検査式を用いた周期モデルのモデル検査では，ワンパスモデルでの検査と同様に，InputOnCnt がちょうど定数 max_II_Cnt のときに，FunctionFlags が 0 のままであるような反例が得られる．この反例を，実際のシステム上での振る舞いに対応させて考えると，図 9 をもとに，以下のような実行パスが考えられる: 入力信号が on の状態が，しばらく続き，InputOnCnt の値が max_II_Cnt となったとする．ここで，入力信号が off になった場合，Task_f が実行されると，4 行目で偽になるため 11 行目以降が実行さる．しかし，InputOnCnt の値が max_II_Cnt のときは，15 行目の条件が偽であれば，21 行目の条件を満たさないため，FunctionFlags は 0 のままとなる．

この問題を修正する方法として，図 9 の 21 行目の条件式が < ではなく，<= に変更すれば，この問題は回避されるが，この修正では他の部分で別のバグを引き起こすことも，モデル検査により確認されている．

実験ではその後，システム A の後継機種であるシステム B に対しても，本稿で説明した検査プロセスに従ってモデル検査を実施した．この検査では，上記検査項目では反例無しの結果を得た．

5 考察

今回の検証実験では，環境ドライバを用いたモデル化手法を採用することにより，既知のバグを検出した．実験前にはバグの詳細原因までは，知らされておらず，現象から推測してモデルの設計方針を立てた．

今回の検証実験には 2 名が参加し，それぞれワンパスモデルと周期モデルを別々に担当した．モデル検査の現場導入を想定した，モデル検査工程の分業化を試みるための実験的な措置である．本実験の後，さらに，環境ドライバ作成と対象コードモデル作成を，分業化する実験も行っている．

本実験の対象システムのソースコードは数万行あり，単純なモデル化では状態爆発を引き起こす．また，ソースコードの振る舞いに忠実なモデルを作り，次に抽象化を施すという従来のモデル化手法では，モデル化工程だけに長い時間を要するおそれがあり，モデル検査の実践的な利用には適さない．いっぽうで，本実験は，環境ドライバの着想を得てからの初の適用実験であったことや，分業化の実験を同時に行ったこともあり，本実験の結果からだけでは，モデル化作業が効率的だったとは言い難い．結果的に，実際にモデル化と検査に要した期間は，2 人×2 ヶ月程度であった．

しかし，検査工数は着実に短縮している．本実験の後，環境ドライバの概念を基にしたモデル化手法を用いた同規模のソフトウェアを対象としたモデル検査実験では，4 人×9 日を実現させた [6]．モデル検査工程で手に負えない状態爆発に至らず，着実にバグの検出が行えること，分業化と作業の効率化が行えることから考えて，環境ドライバを用いたモデル検査の有効性は，実験的に確かめられつつあると考えている．

実験初期に得られた考察として，以下のことが挙げられる．周期モデルの検査の際，ループ回数が大きくなるほど検査する状態空間が広くなり，検査が終わらない可能性が出てくると予想し，カウンタ変数を導入してループ回数を制限できるようモデル化していた．しかし，カウンタ変数を一つでも新しく導入してしまうと，

そのために状態爆発が起こりやすくなる事が検査過程で判明した．現在では，ループ回数の制限を行わずに周期モデルに対するモデル検査を行っている．

ワンパスモデルの役割についても，新たな考察が得られた．当初，ワンパスモデルは，周期モデルに対する補助的な役割のみを期待していた．しかし実験を重ねた結果，周期モデルでは状態爆発により検査できない場合でも，ワンパスモデルを用いることで，ある程度の検査結果が期待できることが分かった．周期タスク型システムをモデル検査する場合に，ソフトウェア・アーキテクチャに忠実にモデルを構成しようとして，状態爆発を制御できない場合がある．モデル化工程に，ワンパスモデルの作成を入れることで，モデル化の手間は増えるが，モデル検査で結果が得られる可能性は高くなる．

本稿で紹介したワンパスモデルおよび周期モデルによるモデル検査は，古橋らの別の事例紹介論文 [6] で提案されている「周期タスク型モデル検査法」におけるモデル単体テストおよび周期モデル検査とほぼ一致している．しかし，提案するモデル検査法での検査工程の目的が，それぞれやや異なる．周期タスク型モデル検査法では，モデル単体テストは，作成したモデルの妥当性検証のために行い，また，モデルを作成した後でモデルの抽象化を行うことにより状態爆発の回避を試みるのに対し，本稿で提案した手法では，ワンパスモデルにおいてもある程度の検証を行い，また，あらかじめ状態爆発が起きないようモデルの作成を進めること試みている．ソフトウェア開発現場のエンジニアの立場に立って考えたときに，最終的なモデルの正当性を優先するか，モデル検査の適用可能性を優先するかは，状況により異なる．したがって，それぞれのモデル検査工程が果たす役割や目的の相違点について，その是非を本論文では議論しないことにする．

謝辞. 本研究は，矢崎総業株式会社およびグループ各社と独立行政法人産業技術総合研究所システム検証研究センターによる共同研究「車載ソフトウェアのモデル検査に関する研究」(2005 年 4 月から 2007 年 3 月まで) にて実施された．共同研究に携わった各位に感謝する．

参考文献

[1] Baker, T. P. and Shaw, A. C.: The Cyclic Executive Model and Ada, Real-Time Systems, Vol. 1,No. 1(1989), pp. 7–25.

[2] 早水公二, 篠崎孝一, 高橋孝一, 渡邊宏: モデル検査器を用いた自動検針システムの仕様検証, 情報処理学会論文誌プログラミング, Vol. 44,No. SIG15(2003), pp. 67.

[3] Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2004.

[4] IEC/SC65A/WG14: Functional safety and IEC 61508, September 2005. `http://www.iec.ch/zone/fsafety/`.

[5] Katayama, T., Nakajima, T., Yuasa, T., Kishi, T., Nakajima, S., Oikawa, S., Yasugi, M., Aoki, T., Okazaki, M., and Umatani, S.: Highly Reliable Embedded Software Development Using Advanced Software Technologies, IEICE Transactions On Information and Systems, Vol. E88-D,No. 6(2005), pp. 1105–1116.

[6] 河本孝久, 小池憲史, 古橋隆宏, 鈴木伸一: 周期タスク型モデル検査法と車載組込みシステムへの適用事例, 組込みシステムシンポジウム (ESS2007), 東京, 情報処理学会, 2007.

[7] 小池憲史, 吉田聡, 大崎人士: LTL モデル検査の為の図示記法, 第 14 回ソフトウェア工学の基礎ワークショップ (FOSE2007), 下関, 日本ソフトウェア科学会, 2007 年 11 月.

[8] 丸山陽太郎, 岸知二, 片山卓也: 組込みソフトウェア設計へのモデル検査適用手法の提案と実験・評価, 組

込みソフトウェアシンポジウム (ESS2005), 東京, 情報処理学会, 2005, pp. 64–71.

[9] 水口大知, 渡邊宏: 組み込みソフトウェア開発におけるモデル検査の適用事例, コンピュータソフトウェア, Vol. 22,No. 1(2005), pp. 70–90.

[10] 篠崎孝一, 太田弘, 早水公二, 星野光勇: モデル検査のデバッグへの適用, ソフトウェアテストシンポジウム (JaSST'06 in Tokyo), 東京, 2006 年 1 月.

[11] 高橋孝一, 高井利憲: システム検証における数理的手法の紹介 – 組込みシステムへの適用事例–, システム/制御/情報, Vol. 51,No. 9(2007), pp. 393–398.

[12] 高井利憲, 吉田聡: アセンブラプログラムのモデル検査による検証事例, 第 5 回ディペンダブルシステムワークショップ (DSW2007), 函館, 日本ソフトウェア科学会, 2007 年 7 月.

[13] 村石理恵, 服部彰宏, 野村秀樹, 山本訓稔: Model Checking を適用した実践的非同期制御検証 – Go with the Early Bird –, ソフトウェアテストシンポジウム (JaSST'07 Tokyo), 東京, 2007 年 1 月.

環境ドライバを用いたモデル検査による検証事例
(算譜科学研究速報)
発行日 2007 年 10 月 1 日
編集・発行：独立行政法人産業技術総合研究所システム検証研究センター
同連絡先：〒563-8577 大阪府池田市緑丘 1-8-31
e-mail：informatics-inquiry@m.aist.go.jp

A case study on verification using model-checking and environmental drivers
Oct 1, 2007
Research Center for Verification and Semantics (CVS)
Ikeda Site
National Institute of Advanced Industrial Science and Technology (AIST)
1-8-31 Midorigaoka, Ikeda, Osaka, 563-8577, Japan
e-mail: informatics-inquiry@m.aist.go.jp

AIST01-J00022-73